Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Telecommunications and Media Informatics

Enhancement of Lane Following Functionality With Temporal Information Integration into Deep Reinforcement Learning

Scientific Students' Association Report

Author:

Tibor Áron Tóth

Advisors:

Róbert Moni
Dr. Bálint Gyires-Tóth

2022

Contents

Kivonat

Napjainkban az autóipar egyik legtöbbet kutatott területe az autonóm vezetés. A technológiai fejlesztések fő célja a teljes automatizálás elérése, ezáltal a közúti balesetek számának és súlyosságának csökkentése, valamint a légkör szennyezésének csökkentése. Ezen célok elérése érdekében fejlett vezetéstámogató rendszereket szükséges kialakítani és fejleszteni.

A gépi tanulás, és különösen a mélytanulás, bizonyítottan hatékony megoldásnak mutatkozik a vezetést támogató funkciók megvalósítására. Ezt láthatjuk az ipari trendekben, akár a kamerák, radarok, lidarok jelfeldolgozását, akár a járművek döntéshozatalát és vezérlését vizsgáljuk. Kimondható, hogy jelenleg elképzelhetetlen lenne az önvezető járművek megvalósítása adatközpontú megoldások nélkül.

Az autonóm vezetésnél egyetlen mérés nem ad minden esetben kielégítő minőségű és mennyiségű információt, mivel figyelembe kell vennünk a járművek és a környezet dinamikáját is. Több mintavétel együttes figyelése közvetett dinamikai információt hordozhat, amely pontosabb döntésekhez vezetheti a modellt. A környezet ilyen állapotainak és az ezekből kiadott irányítási jeleknek a modellezésére szokványosan konvolúciós (CNN) és rekurrens neurális hálózatokat (RNN) alkalmaznak.

Ezen dolgozat a szekvenciamodellezési képességekkel rendelkező mélytanulási modellek teljesítményének vizsgálatáról és kiértékeléséről szól. Az összehasonlítás során különös figyelmet kap a különböző modellek azonos környezetben, azonos körülmények között történő tanítása és tesztelése. A modellek a sávkövetési feladatra tanítása mély megerősítéses tanulással történt, azon belül Proximal Policy Optimization algoritmussal. Az ágensek a sávkövetést egy szimulációban, pontosabban a Duckietown környezetben tanulták. A dolgozat további célja, hogy analizálja és kiértékelje az összehasonlítások eredményeit.

Abstract

One of the most researched areas of the automotive industry today is autonomous driving. The main purpose of technological developments is to achieve full automation, thereby reducing the number and severity of accidents on the road, as well as reducing pollution on the roads. To reach this goal, advanced driver-assistance systems (ADAS) must be created and developed.

Machine Learning, and specifically deep learning, has been proven to be an effective solution for implementing driver assistance functions. We can see this in industrial trends, whether we inspect the signal processing from cameras, radars, lidars or vehicle decision making and control. Nowadays it would be unthinkable to realize self-driving vehicles without data driven solutions.

In autonomous driving, it is ineffective to estimate from a single measurement since we must take into consideration the dynamics of vehicles and the surrounding. Considering multiple observations means more indirect perceived dynamics, which can lead the model to more thoughtful decisions. Modelling such states of the environment and the management signals issued from them is conventional to solved by convolutional (CNNs) and recurrent neural networks (RNNs).

This work focuses on investigating and evaluating the performance of Deep Learning models that have sequence modelling abilities, while adapting them to lane following using Deep Reinforcement Learning. During the comparison, particular attention is paid to learning and testing the different models under the same environment, in the same conditions. The models are trained using Deep Reinforcement Learning with Proximal Policy Optimization algorithm in a simulation entitled the Duckietown gym environment. A further aim of the paper is to analyse the results of the comparisons.

Chapter 1

Introduction

Nowadays, the most supported development trend in the automotive industry is to research and improve highly-automated or self-driving solutions. The goal is to reduce the role and responsibility of the driver in vehicle control and to replace it entirely by a machine eventually. It is known, that these technologies are still facing many challenges. The algorithms must make decisions quickly and confidently in dynamically changing environments at least as well or better than a human driver would. The solutions raise many legal and human responsibility issues which are still awaiting resolution. Rapid development is also made more difficult by the fact that these solutions are required to meet many safety-critical criteria. However, despite the difficulties, the benefits of these developments are quite positive. Autonomous features offer to reduce the number of road traffic injuries which is the leading cause of death in many countries. Also, offer an opportunity to reduce pollution by optimizing traveling routes and vehicle driving style through smarter algorithms or vehicle-to-vehicle and vehicle-to-infrastructure communications. One of the most important areas for development is the improvement of the lane following functionality, which probably offers the most benefits among all of the considered functions.

Lane following features are required in order to achieve SAE level four and five in autonomous driving. The functional and safety requirements at these levels are extremely high. The performance of the existing lane following functions mainly depends on the quality of the measured data provided by the sensors, which is strongly influenced by the weather conditions, visibility conditions and the state of sensor quality. A low amount or poor quality perception is directly impacts the performance of any lane detection algorithm. This research provides a solution for the highlighted problems and to make lane following functionality more robust and reliable with the technique of temporal information integration. Where temporal information integration means that instead of one, we consider and control based on a sequence of measurements taken from the surrounding environment. This method presumably process more information, selects adaptively the more important parts of incoming information and predicts with more accuracy and greater certainty.

Chapter 2

Background

2.1 Modeling temporal information

It can be seen that in most cases it is not sufficient to simply use a single observation to predict the control signal of an autonomous vehicle. Preferably, we have to take into consideration a collection of several past observations, in order to provide a more refined and safer driving. Collecting and storing the various past samples is straightforward, but processing them effectively and extract meaningful relationships from them is more difficult.

A good practice to solve such problems is to utilize sequence modelling with recurrent neural networks (RNNs) from the field of machine learning. Well known RNN architectures are for example, the long short-term memories (LSTMs), the gated recurrent units (GRUs) and the highway networks.[27] These models are able to learn and exploit temporal information from the past, using time-series of data. However, train these models are challenging and difficult, due to the problem of vanishing gradients.[14] Furthermore, they are only able to receive the input sequence content in consecutive manner, i.e. the sequence can not be processed in a single step, therefore can not be parallelized. An other interesting aspect of problem is that which collected samples to consider and what importance should we give to them to provide adequate prediction in the present. To take this into account, a so called attention mechanism have started to be used on sequence to sequence (seq2seq) models. With this extension, sequence modelling is able to consider past dependencies with different importance, regardless of the time distance from the event and the present prediction. Although recurrent networks have improved largely in terms of learning convergence and performance, they have not been able to abandon sequential processing[37].

Such considerations and problems led to the invention of a state-of-art architecture called the Transformer[37] in 2017, which is based on attention mechanisms and neglects previously used convolutions and recurrence from well-knwon convolutional neural networks (CNNs) and RNNs.

2.1.1 Recurrent Neural Networks

In case of standard feed-forward networks the prediction depends only on the current input and previous inputs are neglected. However, Recurrent Neural Networks (RNNs) are able to take into consideration a sequence of inputs by using feedback loops where information is flow back to the network again. This form a memory nature for the network. Due to this particular feature of the RNN models, they are commonly used for memory-intensive tasks as robot control, time series prediction and natural language processing.

2.1.1.1 Vanilla Recurrent Neural Network

The architecture of standard RNN also known as vanilla RNN is simple. It is built from three layers, namely the input, recurrent hidden and output layers. We can represent the model enrolled in the time, as on the figure (figure 2.1). We feed the network consecutively with $\mathbf{x}_t$ inputs. The model calculate its hidden state, based on weighted sum of the previous hidden state $\mathbf{h}_{t-1}$ and actual input $\mathbf{x}_t$. As the result, the model predict its output, what is actually a weighted hidden state. In the next iteration we add the next input part to the model, and it predicts the next output. We have to mention that the hidden state is recurrently fed back into the model from iteration to iteration, thus creating a memory nature to our network. The iterative formula of the hidden state and the output are the followings [32].

$$h_t = sigm(\mathbf{W}^{(hx)}\mathbf{x}_t + \mathbf{W}^{(hh)}\mathbf{h}_{t-1}) \tag{2.1}$$

$$\mathbf{y}_t = \mathbf{W}^{(yh)}\mathbf{h}_t \tag{2.2}$$

This vanilla RNN is a good example for the simplest recurrent network and for the general operation of recurrent models.

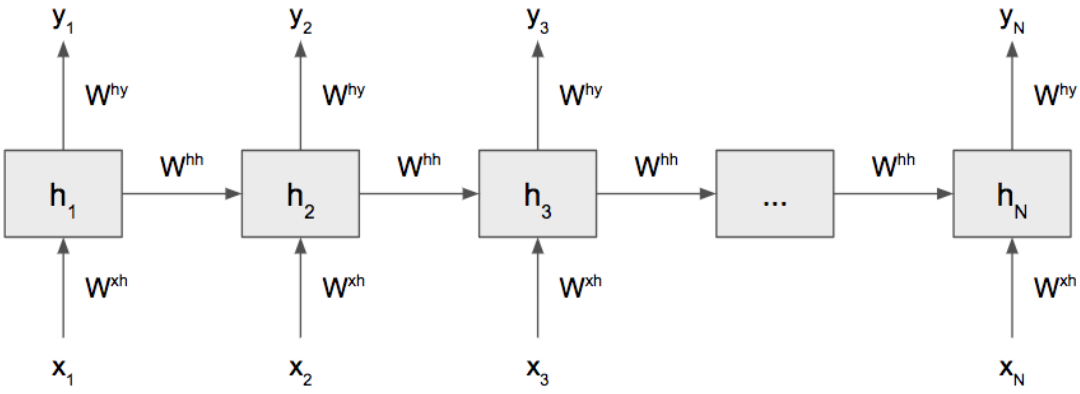


Figure 2.1: The Vanilla RNN architecture enrolled in time. (source: [10])

2.1.1.2 Long-Short Term Memory

The Long Short-Term Memory (LSTM) is an RNN model which makes much better use of its memory and can take into account longer sequences without significant forgetting behavior. This is achieved by introducing a new unit called memory cell figure 2.2.

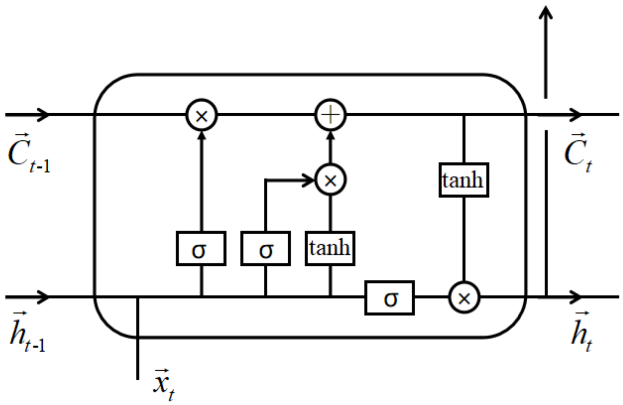


Figure 2.2: The architecture of a Long-short term memory cell. (source: [6])

The cell introduce two distinct state vector, namely the hidden state vector $\mathbf{h}_t \in (-1,1)^h$ and the cell state vector $\mathbf{C}_t \in \mathbb{R}^h$, where h is a hyperparameter, which refers to the arbitrary dimension of the hidden state. These vectors are transport the information through time, i.e. to the next iteration. The new information is added to the network with the input vector $\mathbf{x}_t \in \mathbb{R}^d$, where d is the length of the input vector. The memory cell also contains an input $\mathbf{i}_t$, a forget $\mathbf{f}_t$ and an output $\mathbf{o}_t$ gate, where $\mathbf{i}_t, \mathbf{f}_t, \mathbf{o}_t \in (0,1)^h$. [15] The basic concept can be likened to a conveyor belt, where long term information runs along a chain $\mathbf{C}_t$ with minor linear interventions (gating). The gates allow the LSTM to remove and add information to the data stream flowing on the conveyor. The gating operations are formulated as

$$\mathbf{f}_t = \sigma(\mathbf{W}_f \mathbf{x}_t + \mathbf{U}_f \mathbf{h}_{t-1} + \mathbf{b}_f) \tag{2.3}$$
$$\mathbf{i}_t = \sigma(\mathbf{W}_i \mathbf{x}_t + \mathbf{U}_i \mathbf{h}_{t-1} + \mathbf{b}_i) \tag{2.4}$$
$$\mathbf{o}_t = \sigma(\mathbf{W}_o \mathbf{x}_t + \mathbf{U}_o \mathbf{h}_{t-1} + \mathbf{b}_o) \tag{2.5}$$
$$\tilde{\mathbf{c}}_t = \tanh(\mathbf{W}_c \mathbf{x}_t + \mathbf{U}_c \mathbf{h}_{t-1} + \mathbf{b}_c) \tag{2.6}$$

where $W \in \mathbb{R}^{h \times d}$ is the weight matrix of the input, $U \in \mathbb{R}^{h \times h}$ is the weight matrix for the recurrent connections and $b \in \mathbb{R}^h$ is the bias vector. [15][36] From this the cell and hidden state vectors can be derived as

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t \tag{2.7}$$
$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t) \tag{2.8}$$

2.1.2 Sequence to sequence

Sequence to sequence (seq2seq) learning is a machine learning method where our so called seq2seq model take a sequence as an input and generate another sequence as an output, where the input and the output length can match or differ. It is typically applied for time series prediction[20], machine translation[35], text summarization[31] and so on. Since seq2seq models are invented in the field of language modeling, the best practice is to introduce them on language translation tasks. This does not mean that these solutions can not be used to implement self-driving functions, since instead of using word embeddings as inputs, we can simply use encoded information descriptor vectors that can carry arbitrary information, such as images from cameras, sensor measurements and so forth.

2.1.2.1 Trivial case

In the trivial case of seq2seq learing, the input $(\mathbf{x}_1, ..., \mathbf{x}_T)$ and the output $(\mathbf{y}_1, ..., \mathbf{y}'_T)$ sequence lengths are equal, i.e. $T = T'$. The model architecture for this scenario is very simple, since we have to use only one RNN basemodel, which is introduced before (figure 2.1). It can be seen that the length of the input and output can not change, since the input indicates the generation of the output in the same time step. It should be mentioned that in the field of machine translation, this seq2seq consideration is not quite applicable, since the sentence or word length can be different between the source and the target language.

2.1.2.2 General case

In some cases, we need different input $(\mathbf{x}_1, ..., \mathbf{x}_T)$ and output $(\mathbf{y}_1, ..., \mathbf{y}_{T'})$ length, e.i. $T \neq T'$. For this purposes the previous architecture has to be replaced. The new architecture is composed from three main, functionally different parts, namely an encoder, a context vector and a decoder (figure 2.3). To ensure the length difference between the ins and outs, the encoder and the decoder using two different model [32]. In our case, based on the original research [32] we will use two LSTMs.

The encoder's purpose is to consecutively get all parts of the input sequence into the model and map those to a fixed-size vector representation. Inside the model, the hidden states are being updated step-by-step through iterations and the final hidden state is derived what is actually the mentioned fixed-size v context vector. Based on this context vector and the previous predicted outputs, the decoder predicts the next output. Lets note, that the decoder only starts to working after the encoder finished, thus the decoder does not use encoder's hidden state except the context vector [32].

The model works with a conditional probability $p(\mathbf{y}_1, ..., \mathbf{y}_{T'} | \mathbf{x}_1, ..., \mathbf{x}_T)$, what describes the probability of a possible output sequence in the case of the input sequence

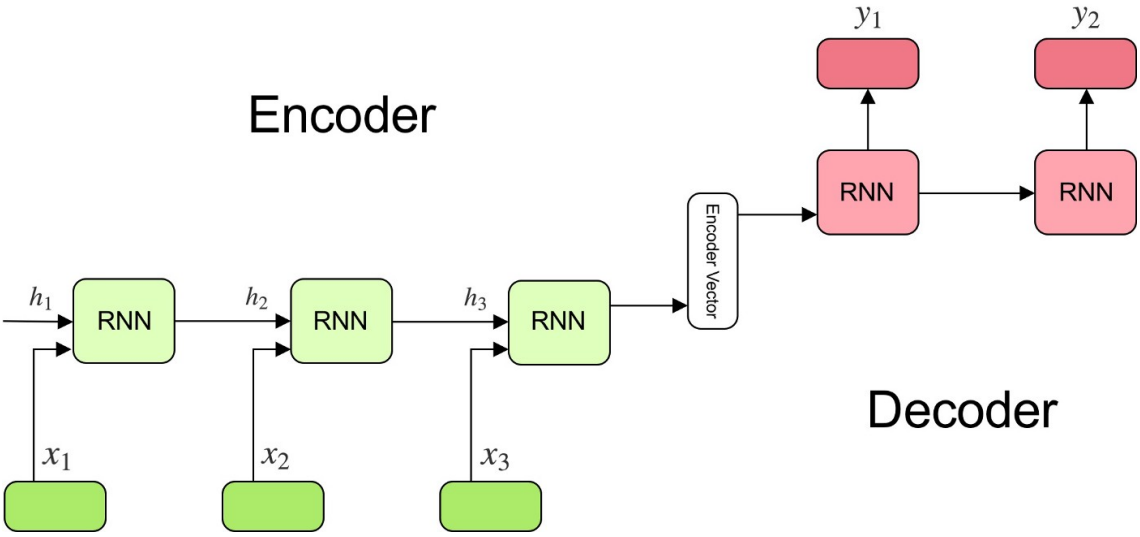


Figure 2.3: Seq2seq model architecture for general case, where input and output sequence length can be different. (source: [18])

is given. This probability can be described with the law of total probabilities:

$$p(\mathbf{y_1}, ..., \mathbf{y}_{T'} | \mathbf{x}_1, ..., \mathbf{x}_T) = \prod_{t=1}^{T'} p(\mathbf{y}_t | \mathbf{v}, \mathbf{y}_1, .., \mathbf{y}_{t-1}) \tag{2.9}$$

This equation in a standard LSTM-Language Model (LSTM-LM) formulation means, that using an LSTM with a softmax activation will predict the next most probable word given the previous predicted words and the context vector[32].

The original seq2seq with LSTM research [32] provides an input reversing trick. On these basis, if we reverse the order of the input, the first estimated word is fall close to the first input word in the processing chain. For example for *(I, trust, you)* the translation would be *(Én, bízom, benned)* in Hungarian. If we reverse the input *(you, trust, I)* the last hidden states (include context vector) will be calculated from the previous hidden state and the actual *(I)* input. To predict the right Hungarian word *(Én)*, it is practical that the most relevant word is processed right before the prediction, hence it cannot be "forgotten" by the updates of the hidden state.

We can see on the previous example, that the sequential modeling has a demand for knowing what to pay attention to within the input sequence. This is the problem what the attention mechanism will relieve.

2.1.3 Attention

In the field of machine learning, the attention mechanism is familiar to a simple everyday phenomenon, such as a person's visual perception is focusing on the interesting parts of a sight or people are paying more attention to individual words in a sentence.[38] In human life these are biological endowments, but helps us to focus strongly on the relevant and to neglect the unnecessary information. The ad-

vantages are conspicuous, we are able to understand and analyze the information more efficiently without getting overwhelmed by its amount. These benefits can be superior in the field of machine learning, where we have a hardware with limited computing performance. Our agents can be learn to solve tasks faster, to focus on valuable instances of the input sequence and to describe the association between them.

In the original study [3] the attention mechanism was demonstrated on a seq2seq model, which using a bidirectional RNN as a decoder and a standard RNN as decoder (figure 2.4). The bidirectional RNN has forward and backward hidden states, which are concatenated to a common hidden state as $\mathbf{h}_i = \left[\overleftarrow{\mathbf{h}_i}; \overrightarrow{\mathbf{h}_i}\right]$. The advantage of using hidden states processed opposite directions on the input sequence is that the encoder's state will get information from states before and after itself [38].

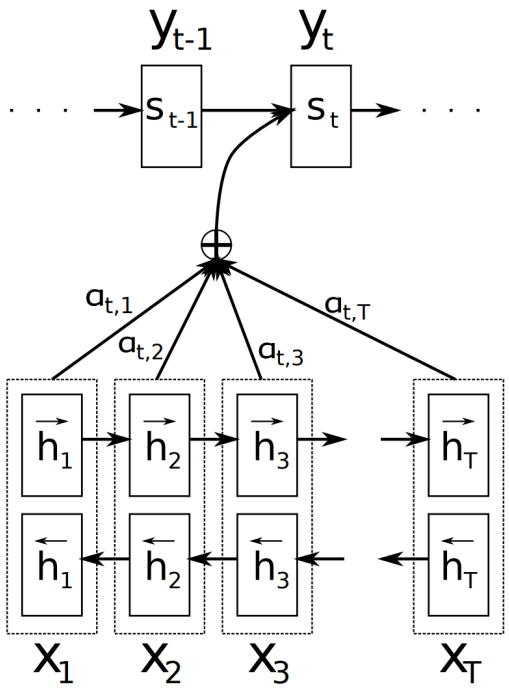


Figure 2.4: Attention mechanism. (source: [3])

We are still looking for a mapping between an input and an output sequence, but now we are indexing the encoder's positions with i and decoder positions with t, as follows.

$$\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_i, ..., \mathbf{x}_T] \tag{2.10}$$

$$\mathbf{Y} = [\mathbf{y}_1, \mathbf{y}_2, ..., \mathbf{y}_t, ..., \mathbf{y}_{T'}] \tag{2.11}$$

At the attention mechanism, the derivation of the decoder's hidden state has to be modified. We still take into consideration the previously predicted outputs, the previous hidden state, but now we generate individual $\mathbf{c}_t$ context vector for each decoder position. Previously at seq2seq-LSTM models we used only one fixed-length $\mathbf{v}$ context vector for the whole decoding process. The formula of the decoder's

network hidden state [38]:

$$\mathbf{s}_t = f(\mathbf{s}_{t-1}, \mathbf{y}_{t-1}, \mathbf{c}_t) \tag{2.12}$$

This time the $\mathbf{c}_t$ context vector is calculated (for each decoder position) from the weighted sum of all encoder hidden state, where the weights are the so called $\alpha_{t,i}$ attention weights [5]. The iterative equations for the context vector and the attention scores:

$$\mathbf{c}_t = \sum_{i=1}^{N} \alpha_{t,i} \mathbf{h}_i \tag{2.13}$$

$$\alpha_{t,i} = attention(\mathbf{x}_i, \mathbf{y}_t) \tag{2.14}$$

$$\alpha_{t,i} = p(e_{t,i}) \tag{2.15}$$

$$e_{t,i} = align(\mathbf{s}_{t-1}, \mathbf{h}_i) \tag{2.16}$$

The equation (2.8) is the formal representation of the method. The equation (2.9) and (2.10) describes the details. The attention function estimates the score, what indicates how well the compared input and output pair $(\mathbf{y}_t, \mathbf{x}_i)$ are match with each other. The collection of $[\alpha_{1,i}, \alpha_{2,i}, ..., \alpha_{M,i}]$ controls how much the decoder takes each encoder state into account [5].

The alignment scores (2.10) are learned by an additional feed forward neural network, which built into the architecture. Its input is the encoder-decoder hidden state pair. The align function i.e. the neural network, outputs an $e_{t,i}$ energy score. This is passed through the p distribution function (usually a softmax function), which converts energy score into attention weights. This neural network with the other recurrent parts of the model is differentiable, thus it can be trained jointly with backpropagation algorithm [5].

The relation between $(\mathbf{y}_t, \mathbf{x}_i)$ is described with the use of $(\mathbf{s}_{t-1}, \mathbf{h}_i)$. It is interesting that the formula uses $\mathbf{s}_{t-1}$, rather $\mathbf{s}_t$. This is because in order to calculate decoder hidden state in the actual position (2.6), we need the actual $\mathbf{c}_t$, but at this step only the $\mathbf{s}_{t-1}$ hidden state is available. This solution does not cause an information flow issue, since next state $\mathbf{s}_t$ provides the $\mathbf{y}_t$ output, based on the $\mathbf{s}_{t-1}$ previous state [3].

2.1.3.1 Generalize attention mechanism

For generalize the attention mechanisms, the following conventions are introduced. The attention mechanism on the seq2seq architecture (figure 2.4) can be treat as a key-value-query mapping process. The sequence of $\mathbf{K}$ keys (for each encoder hidden states) are mapped (2.11) to attention weights regarding to $\mathbf{q}$ queries (single decoder hidden state). Some solutions do not use or not make difference between keys and $\mathbf{V}$ values [5]. In this formulation we use $\mathbf{V}$ values as additional inputs, but it is denoting the same encoder hidden state, as the keys [3]. The importance of the

$(\mathbf{K}, \mathbf{V})$ pair will rise at transformer models, which will described later.

$$A(\mathbf{q}, \mathbf{K}, \mathbf{V}) = \sum_{i}^{N} p(align(\mathbf{k}_i, \mathbf{q})) * \mathbf{v}_i \tag{2.17}$$

The capitalness of the $\mathbf{K}$ keys and $\mathbf{V}$ values, means there are multiple encoder hidden states and inputs are mapped to a single $\mathbf{q}$ query or can referred as decoder hidden sate.

2.1.3.2 Alignment score functions

Based on how we determine the alignment score, we can differentiate quite a few attention mechanisms. For example, there are dot-product[21], general[21], additive[3], location-based[21], scaled dot-product[37] attentions and so forth. Their alignment functions [5][38]:

Type	Equation
dot-product	$align(\mathbf{k}_i, \mathbf{q}) = \mathbf{q}^T \mathbf{k}_i$
general	$align(\mathbf{k}_i, \mathbf{q}) = \mathbf{q}^T \mathbf{W} k_i$
additive/concat	$align(\mathbf{k}_i, \mathbf{q}) = \mathbf{V}_a^T tanh(\mathbf{W}_a[\mathbf{q}; \mathbf{k}_i])$
location-based	$align(\mathbf{k}_i, \mathbf{q}) = align(\mathbf{q})$
scaled dot-product	$align(\mathbf{k}_i, \mathbf{q}) = \mathbf{q}^T \mathbf{k}_i / \sqrt{N}$

Table 2.1: Examples of different alignment functions for attention.

The dot-product alignment calculates the dot-product between keys and query (encoder hidden states and decoder hidden state). The general method introduce a learnable $\mathbf{W}$ matrix, what maps the query to keys. The additive or can referred as concat method is simply concatenate the hidden states together and adds $\mathbf{V}_a, \mathbf{W}_a$ learnable weights with *tanh* activation function. The scaled dot-product is extended version of the basic dot-product, where the values normalized with the root of the keys length. This last method helps to overcome the problems of the long input sequences, since they product small gradients with softmax activation functions. We will find this alignment function at transformer architectures, probably because transformers are applicable for long input sequences [5].

2.1.3.3 Categories of attention mechanisms

Attention mechanisms can be differentiated based on their number of sequences, abstractions, positions and representations [5]. We should consider these categories as a continuous spectrum of possible solutions, most of the time there are no strict borders between each interdisciplinary solution. The most commonly distinguished types are the self-, soft-, hard-, local- and global-attentions [38]. The categories that include these more common types are discussed and elaborated below.

We allocate distinctive-attention, co-attention and self-attention by the number of different sequence used to the mechanism. At the distinctive-attention, the key and

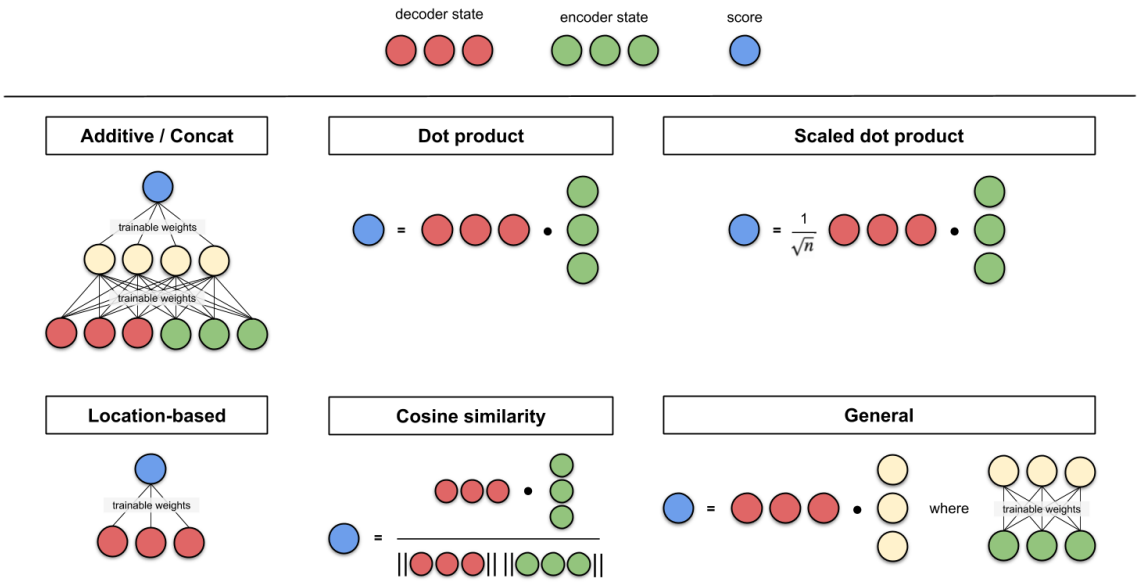


Figure 2.5: Illustrated alignment functions. (source: [17])

query states are from two different sequences. Until now we used only this type, since we had one input sequence and a related output sequence. The co-attention works with multiple input data and evaluate relations with scores between all input sequences as well [5]. The self-attention is a well known mechanism where only the input sequence is processed, in such a way that it links different positions of the input sequence to each other, thereby it computes a representation of itself. This way, self-attention finds influences and dependencies between input sequence positions.

The following example (figure 2.6) shows how self-attention works on an example sequence. In this case the items of the input are the words and the sequence is the sentence. The red color denotes the current word and the blue color refers to the level of memory activation or attention weight of a word [7]. As usual at the RNNs architecture, the mechanism is getting inputs sequentially one by one. The self-attention mechanism learns the association between the actual input word and the past words of the sentence. We can see that when the mechanism predicts the (*chasing*) word, it pays more attention to related words like (FBI) and (is).

Based on the number of the considered positions for attention function, we can distinguish soft-, hard-, local- and global-attentions.

At the soft-attention, the mechanism takes into consideration all the hidden states of the input sequence for determine context vector. Thus, it considers the whole input sequence e.g. the whole sentence or the whole picture at vision transformers. This soft weighting method makes the network differentiable, thereby for trainable with backpropagation, but debits high computational cost. The attention of the seq2seq model discussed so far, have used this soft-attention mechanism. The hard-attention uses only a part of the input, which is selected by stochastic sampling. This means that only a separated group considered in the attention function, e.g. some words

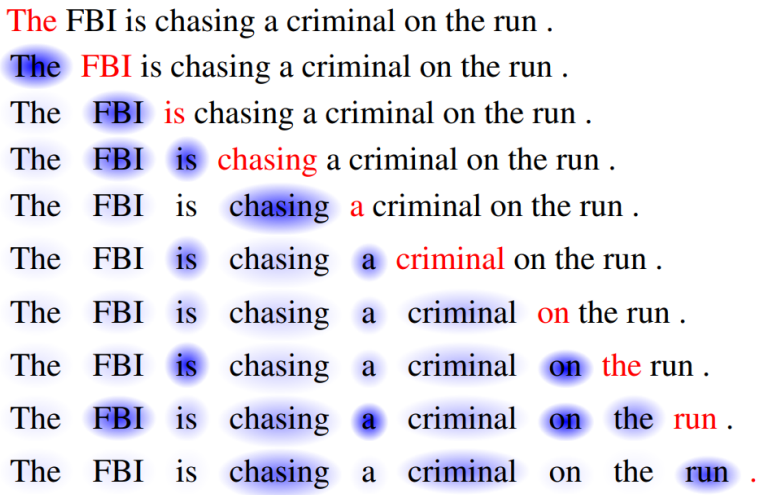


Figure 2.6: Self-attention applied on an example input sentence. (source: [7])

in a sentence or a patch in a photo. This method decreases computational cost, but makes the network non-differentiable what can lead into training difficulties [5].

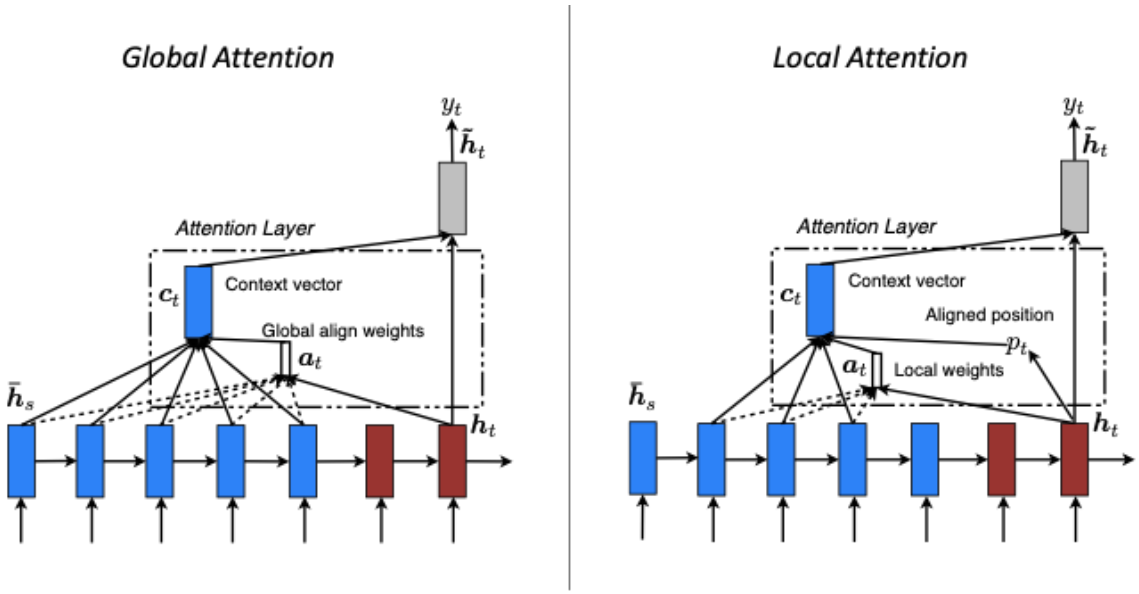


Figure 2.7: Global and local attention mechanism. (source: [21])

The local-attention combines the advantages of soft- and hard-attention, because it detects an attention position or a point on the input and then places a window around it, where it performs the attention mechanism. This results differentiability and lower computational cost for the network [5]. As shown in the figure (figure 2.7), the local-attention takes only a collection of $\mathbf{h}_s$ encoder hidden states and one $\mathbf{h}_t$ decoder hidden state to determine $\mathbf{a}_t$ local alignment weights. The global-attention is really similar to soft attention and their names are mostly used interchangeable in the field of machine learning. We can see (figure 2.7) that soft/global attention,

rather than takes a limited collection of $\mathbf{h}_s$ encoder hidden states, it takes all of them for the mechanism [5].

2.1.4 Transformer

The original state-of-art transformer architecture [37] proposed in 2017. It brought change in the way that it is able to neglect the use of complex recurrence or convolution of the previous sequence modelling architectures and use solely attention mechanism with simple feed forward network modules (figure 2.8). This model still has an encoder-decoder architecture, but it does not need to process the input sequence necessarily in order, thus accelerated hardware capabilities can be exploited. It is able to learn sequence processing tasks with less complexity, while enhancing long-time dependency acquisition easily [37].

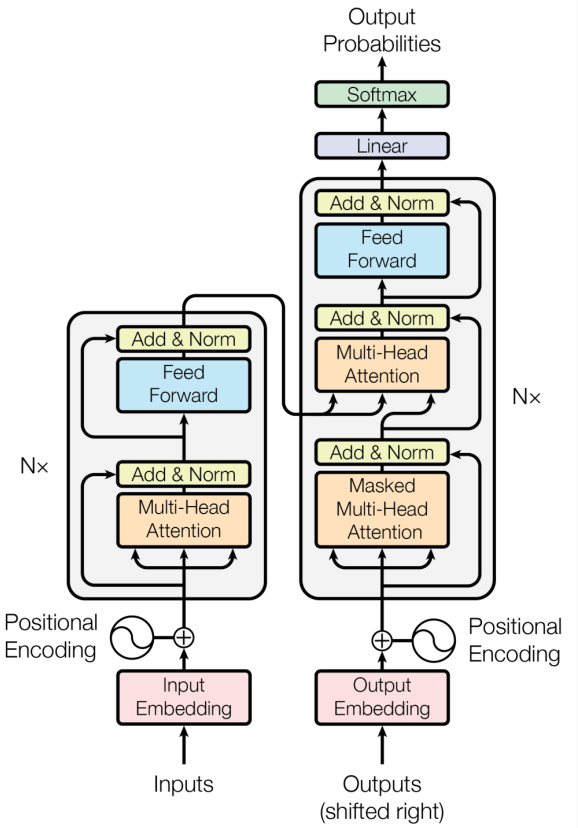


Figure 2.8: Transformer architecture. (source: [37])

The architecture (figure 2.8) is built from a similar, but a bit different encoder (left main blocks) and decoder parts (right main blocks). The decoder processes the inputs into attention-based representation and identifies the information from input sequence, which are worth to pay attention to. The decoder enquires the information from the encoder part [39]. These parts are performed N times (i.e. on all position of the sequence) in each prediction. This is where the transformer really shines over the RNNs, since parallelization is possible here. As we can see, there are smaller building blocks of the network, such as input embedding, positional

encoding, multi-head attention, masked multi-head attention, add&norm layer, feed forward network, linear network and softmax activation [37].

In the case of natural language processing (NLP) the input embedding block transforms the words into a representation that can be understood by computers. The words are usually described with a single vector in the so called embedding space, in a way that the words are close to each other in the meaning are placed close to each other in the embedding space. This approach is appropriate, however due to the permutation-invariant property of the attention mechanism, the model lacks of the information about the order of the input sequence. This is causing a problem, because words may have different meanings when they are in different parts of the sentence. To solve this issue, the transformer architecture proposes a positional encoding block, which modifies the values of the embeddings by the adding different constant values to each position's vector [37] [39]. In some cases, to use the transformer on other types of input data, we can map the input to a vector (as an embedding) with different techniques or networks (e.g. CNNs, auto encoders).

The multi-head attention (figure 2.9) is a key building block for transformer models. It performs self-attention mechanisms multiple h times on the projected version of the input in a stacked manner. The research proven that execute multiple stacked attention on a reduced dimensional input is more beneficial, than one self-attention mechanism on the high dimensional original input [37].

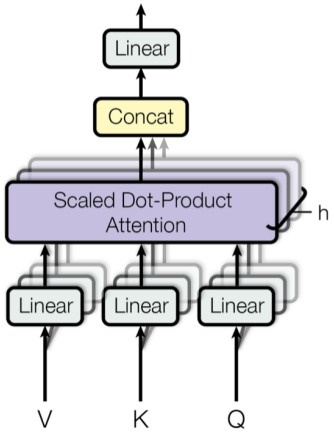


Figure 2.9: Multi-head attention. (source: [37])

At this original transformer, the set of $(\mathbf{V}, \mathbf{K}, \mathbf{Q})$ vectors are collected into matrices and scaled dot-product alignment function-based attention performed on them [37].

$$\text{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Concat}(\text{head}_1, ..., \text{head}_h)\mathbf{W}^O \tag{2.18}$$

$$\text{,where head}_i = \text{attention}(\mathbf{Q}\mathbf{W}_i^Q, \mathbf{K}\mathbf{W}_i^K, \mathbf{V}\mathbf{W}_i^V) \tag{2.19}$$

$$\text{,where attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}})\mathbf{V} \tag{2.20}$$

Encoder's attention mechanism takes all of $(\mathbf{V}, \mathbf{K}, \mathbf{Q})$ matrices form the previous layer, but decoder's attention takes the query from previous decoder layer and the values, keys from the encoder output [37].

The masked version of multi-head attention is different only in its self-attention process. The unit masks the part of the input sequence what is coming up later than the actual position, to hide the future information from decoder. Note, that the self-attention at RNNs (figure 2.6) is looked similar as the masked self-attention at transformers, what is caused by the consecutive input feeding of RNN. In the case of transformer model, the self-attention calculates weights for all positions of the input sequence, the masked unit only for a part of the sequence [37].

The residual connections in the network transfer earlier representations into later parts with addition and normalization. The feed forward block is a point-wise fully connected network, thus it make the same projection on all attention vectors, which are relate to each position.

Despite the fact that this first transformer model was a breakthrough in the field of sequence modelling and especially for NLP uses, it was not capable to perform on wide spectrum of use cases, such as image processing, reinforcement learning and so on. Several new solutions proposed since then, such as Transformer-XL [9] to solve base model's limited attention-span, Image Transformer [24] to make it applicable for images and Gated Transformer-XL [23] to use for reinforcement learning (RL) scenarios.

2.1.4.1 Gated Transformer-XL

The use and train of the transformer as a decent DRL memory was not achievable for a long time. A new state-of-art research [23] has shown that a specific variant of transformer, called Gated Transformer-XL (GTrXL) is able to learn in deep reinforcement learning situations. Furthermore, in some cases, especially where longer memory requiring environments are used, it can outperform LSTM models.

The GTrXL is built from several stacked blocks similarly as at the original transformer model. These blocks are apply self-attention mehcanisms on a recurrent basis. Each block consists of a multi-head attention unit and a position-wise multi layer perceptron (MLP). The additional features are the followings. The return side of the residual connections are replaced with gating units and the normalization layers are relocated. The intuition behind these changes is that multiplicative interactions are more successfully stabilizing learning for different architectures is proven [23].

We have L total number of transformer units, which are stacked. Each l-th unit got two input source. One is an embedding $E^{(l-1)}$ from the previous unit, except the first layer's input which can be word embedding for NLP uses or a sampled observation for RL uses. The other is a memory tensor what stores the previous inputs and does not allows to gradients flow backward in it. These two input is processed by a normalization and a multi-head attention, followed by a gating layer. The solution proposes multiple variants for the gating layer, but the most successful is the GRU unit-based one. The information flow at the second half of the unit is

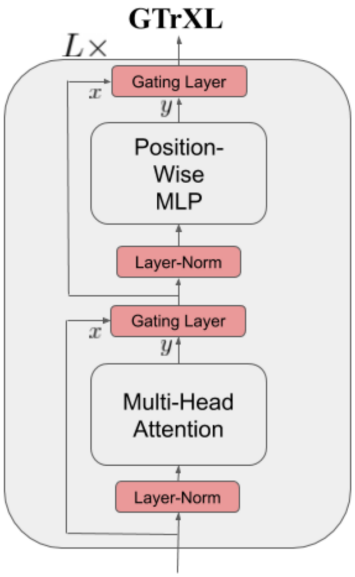


Figure 2.10: The GTrXL architecture. (source: [23])

exactly the same, but with a position-wise MLP [23].

$$\overline{\mathbf{Y}}^{(l)} = \text{RelativeMHA}(\text{LayerNorm}([\text{StopGrad}(\mathbf{M}^{(l-1)}), \mathbf{E}^{(l-1)}])) \tag{2.21}$$

$$\mathbf{Y}^{(l)} = g_{\text{MHA}}^{(l)}(\mathbf{E}^{(l-1)}, \text{ReLU}(\overline{\mathbf{Y}}^{(l)})) \tag{2.22}$$

$$\overline{\mathbf{E}}^{(l)} = f^{(l)}(\text{LayerNorm}(\mathbf{Y}^{(l)})) \tag{2.23}$$

$$\mathbf{E}^{(l)} = g_{\text{MLP}}^{(l)}(\mathbf{Y}^{(l)}, \text{ReLU}(\overline{\mathbf{E}}^{(l)})) \tag{2.24}$$

These prudent changes have made GTrXL capable of stabilize and speed up teaching for reinforcement learning applications. Among the transformers the GTrXL in general has longer-term attention span, less computational consumption and less memory needs [39].

2.2 Reinforcement Learning

In the field of machine learning we distinguish three main approach of learning, namely: supervised, unsupervised and reinforcement learning. Each of these has advantages and disadvantages over the others. The characteristics of the problem to be solved and the availability of the data will determine which approach is the most appropriate for a certain problem.

In the case of supervised learning, the algorithm is trained on fully labeled dataset. Fully labeled means that all sample from the dataset are already paired with the information that the algorithm needs to estimate. This way, we can examine, compare the algorithm's predictions and the ground-truth labels, which information can be used to improve our machine learning model's problem solving ability for the given task. Supervised learning offer solution for classification and regression type of problems. At classification the input samples are grouped into different classes. As simple as it sounds, it covers a wide range of modern problems, such as object detection, semantic segmentation, and so on. On the other hand regression gives solution to analyse relationships between dependent and independent variables. Related example problems can be temperature, salary and price predictions. Well known models which are usually deployed for supervised learning without being exhaustive can be support-vector machine (SVM), decision-tree, k-nearest neighbors (K-NN) and neural network (NN).

Unsupervised learning algorithms cluster and analyze datasets which are unlabeled, thus these algorithms are able to learn information and derive consequences without any feedback given. These algorithms are also appealing because they are not influenced by human prior knowledge like in the case of other methods. Main problems can be clustering (samples are grouped based on their similarities and differences), association (find hidden relations between samples) and dimension reduction (shrinks the data size while maintain its information). Some popular algorithms which are usable for unsupervised learning are autoencoder (AE), principal component analysis (PCA), K-means and isolation forest.

Reinforcement learning (RL) algorithms solve problems in a trial and error manner. In general, these algorithms take action on the basis of their experiences and based on the outcome of the taken action (which is given to the algorithm as reward or penalty) they change their way of thinking and behavior. Unlike at other approaches, RL algorithms do not need a classical dataset collection, since they learn and perform in dynamic environments. This area of machine learning is still in intense research, as it provides a solution to many industrial, robotics and control tasks without needing to formulate how the problem should be solved. An RL agent can be practically any machine learning model, but the realisation and formulation of the dynamic environment and the algorithm which makes the agent learn can be quite diverse.

2.2.1 Foundations

In reinforcement learning the agent is interacting with the environment. Simply, at every iteration the agent takes observation o_t from the environment state s_t and

based on it, takes an action a_t. The environment shifts its state as a result of the action taken. Based on how good is the new state to be in, the agent receives reward r_t. Based on the reward and the state the agent takes an action again. The main goal in RL is to maximize the cumulative reward (referred as return) collected by the agent through the iteration steps. [19][1]

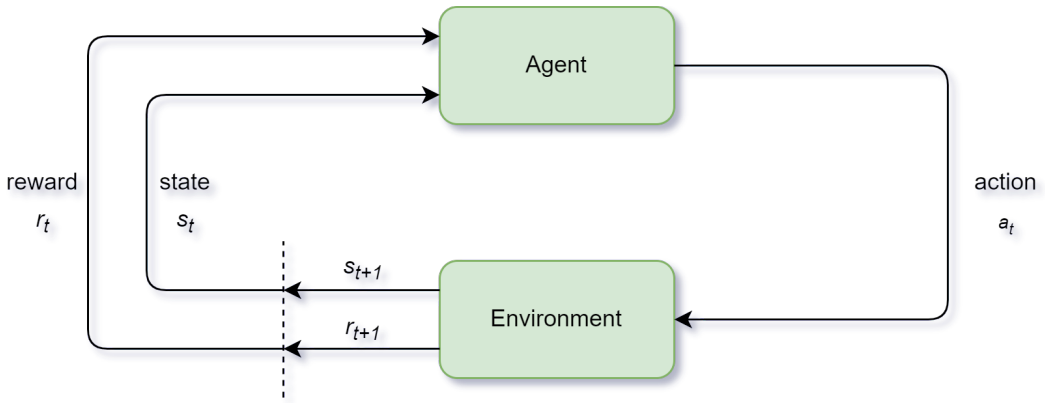


Figure 2.11: Overview of the agent-environment RL loop. (source (modified): [33])

The state is totally describe the environment, this means that the state includes all information about the world and there is no such information which is hidden from the state. The agent's observation is taken from the state, it can contain all or only the part of the information which described by the state. Based on these we can distinguish two environment types. If the agent is able to observe the entire state we talk about fully observed environment, but if only the the part of it, we refer to it as partially observed environment. [1]

The possible taken actions are determined by the different environments. There are discrete and continuous action spaces. At discrete action spaces the number of possible moves are finite, such as the possible moves of a chess piece. At continuous action spaces the available actions are described with real-valued vectors, for example steering wheel and gas pedal angle at car driving. [2]

The brain or the logic on the basis the agent makes decisions is called policy π in reinforcement learning context. The policy can be deterministic or stochastic. Since in my related work and realizations I only deal with stochastic policies, I will detail these in the following. The action is sampled from the stochastic policy as described:

$$a_t \sim \pi(\cdot|s_t) \tag{2.25}$$

It worth to mention that the system partially presented so far is described and modelled as a Markov Decision Process (MDP) [34]. The MDP model has the following elements:

- state space S - set of all valid states
- action space A - set of all valid actions
- initial state s_0 - a state sampled from start-state distribution $\rho_0(\cdot)$

- reward function R
- transition probability function P

With reward function we can derive the reward $r_t = R(s_t, a_t, s_{t+1})$. With the transition probability function we can calculate the probability $P(s'|s, a)$ of stepping into a state s' if the agent in the state s and take action a. Besides these, the MDP model complies with the markov property, which describes that the transition into a new state only depends on the previous state-action pair. [34]

As mentioned above, the only technical aim of reinforcement learning is to maximize the expected return, in order to achieve the desired agent behaviour. To formulate the problem we are examine a possible sequence of action-reward pairs, called trajectory $\tau = (s_0, a_0, s_1, a_1, ..., s_T, a_T)$. If state transitions and the policy are stochastic, the following equation is given for the probability of a trajectory:

$$P(\tau|\pi) = \rho_0(s_0) \prod_{t=0}^{T-1} P(s_{t+1}|s_t, a_t)\pi(a_t|s_t) \tag{2.26}$$

From this the expected return $J(\pi)$ can be calculated as:

$$J(\pi) = \int_{\tau} P(\tau|\pi)R(\tau) = \underset{\tau\sim\pi}{E}[R(\tau)] \tag{2.27}$$

where $R(\tau)$ is the return calculated for a single trajectory. Here we can differentiate two kind of return. In the case where only a fixed size window used to sum up the rewards, we call it finite-horizon undiscounted return [1]. Which can be described as:

$$R(t) = \sum_{t=0}^{T} r_t \tag{2.28}$$

In other case where we sum all rewards ever obtained and multiply each element with a discount factor, we call it infinite-horizon discounted reward. It's formula is the following:

$$R(t) = \sum_{t=0}^{\infty} \gamma^t r_t \tag{2.29}$$

where $\gamma \in (0, 1)$ is shrinking the importance of the more distant rewards. Intuitively, this adds a "better now than later" behaviour, which effect can optimized with the discount factor γ. [1] Finally, the optimization problem of RL can be formulated as:

$$\pi^* = \arg\max_{\pi} J(\pi) \tag{2.30}$$

Theoretically, if we achieve π^* policy we will have the agent which behave in the environment perfectly as we want it, perhaps if this desire coincides with our reward function. In practice, for complex environments and problems we tend to use deep learning models for RL, which together we call deep reinforcement learning. In these setups we train our DL model to approach as close as possible the theoretical π^* policy.

To better understand the algorithms that will be presented later, two function must introduced. The value-function $V(s)$ and advantage function $A(s,a)$. Value-functions are describe the expected return if the agent start act in a state or state-action pair and then act regarding to its actual policy forever. [19] The value function $V^{\pi}(s)$ and action-value function $Q^{\pi}(s,a)$ can be described as:

$$V^{\pi}(s) = \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)|s_0 = s] \tag{2.31}$$

$$Q^{\pi}(s,a) = \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)|s_0 = s, a_0 = a] \tag{2.32}$$

Note that in the case of value function only the state is given. At action-value function the first arbitrary action is given and its taken by the agent, then in both cases the agent acts according to the fixed policy. The other relevant function is the advantage function, what rates an action among other possible actions without describe its goodness in an absolute manner. [1] The advantage function for a given state and a possible action has the following form:

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s) \tag{2.33}$$

If we want to keep it simple, the advantage function returns the expected return difference if we deviate in the first step from the policy. In case we calculate the advantage in a given state for a several or for all possible actions, than we can rank and select the best action to take.

2.2.2 Algorithms

Choosing the proper algorithm is essential in reinforcement learning, since generally this is the main logic what trains our agent. The algorithm has important tasks as running the agent in the environment, updating agent's policy, leading the agent to the desired behaviour. In my work I deal with a model-free, on-policy, policy gradient method called Proximal Policy Optimization (PPO). To give a general overview, I will present the most common grouping and basic properties of these categorized RL algorithms. After that, I will present policy optimization methods in more detail.

In modern RL we can differentiate model-based and model-free algorithms, based on whether the agent uses a model of the environment or not. Using a model-based approach has the advantage that our agent is capable of select an action by examine all the possibilities and their outcomes by thinking ahead. The downside of this method that the environment model which properties identical to the original environment is usually not available and in these cases the agent has to learn the model by its experience. It is fair to say that model-based approaches are more sample efficient, but model-free approaches seem to be more simple to implement, easier to train and more likely to lead to working solutions. [13][1]

The on-policy property means that each update of the policy only relies on the information collected with the actual policy, later policies are not used by the algorithm.

Model-based approaches can be divided into two subcategory, which are policy optimization and Q-learning methods. I would like to introduce the policy optimization

methods which relates to my work. Policy optimization methods has the policy $\pi_\theta(a|s)$ defined in explicit form. The θ policy parametrization refers to that the policy function's output is a computable function which depends on a set of parameters, usually on the learnable parameters of a neural network. These methods are execute gradient ascent on the return function $J(\pi_\theta)$ or maximizing the approximation of the return function to optimize the policy. Moreover, policy optimization methods often learn an approximator for the value function. [1][8] Further details are described at each instance of policy gradient methods (2.2.2.1)(2.2.2.2)(2.2.2.3).

2.2.2.1 Vanilla Policy Gradient Optimization

The vanilla policy gradient algorithm approximates the gradient via construct an estimator for the policy gradient and then optimize it with a stochastic gradient descent (SGD) algorithm [1]. The pseudo code of the algorithm is the following.

Algorithm 1 Vanilla Policy Gradient Algorithm (based on [1])

1: Get initial policy parameters θ_0, initial value function parameters ϕ_0.
2: **for** k = 0,1,2,... **do**
3: Collect trajectories $\{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
4: Compute reward-to-go $\hat{R}_t$.
5: Compute advantage estimates $\hat{A}_t$, based on actual value function ϕ_k.
6: Approximate policy gradient with:

$$\hat{g}_k = \frac{1}{|\{\tau_i\}|} \sum_{\tau \in \{\tau_i\}} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)|_{\theta_k} \hat{A}_t \tag{2.34}$$

7: Compute policy update: $\theta_{k+1} = \theta_k + \alpha_k \hat{g}_k$
8: Update value function parameters by regression with MSE:

$$\phi_{k+1} = \underset{\phi}{\operatorname{argmin}} \frac{1}{|\{\tau_i\}|T} \sum_{\tau \in \{\tau_i\}} \sum_{t=0}^{T} (V_\phi(s_t) - \hat{R}_t)^2 \tag{2.35}$$

9: **end for**

Other policy gradient methods are developed from this algorithm, so they use this method analogy and alter from this via some modifications. In the following, we will see on the improved variations that introducing slight changes to the architecture can eliminate the problem of too big policy updates, which is the main drawback of vanilla policy gradient optimization.

2.2.2.2 Trust Region Policy Optimization

Trust Region Policy Optimization (TRPO) updates agent's policy with selecting the largest possible step to improve performance, but it constraints the difference between the old and the new policy with Kullback-Leibler (KL) divergence. The improvement over the vanilla policy gradient is that TRPO monitors and regularizes

the divergence between policy distributions, rather than regularizes the difference in parameter space. The underlying logic is that small changes in parameter space can cause huge differences in performance, thus controlling the policy on parameter space feature is not effective. [29][1]

The maximization of policy update and its constraint formed as

$$\underset{\theta}{\text{maximize}}\ \hat{\mathbb{E}}_t[r_t\hat{A}_t] \tag{2.36}$$

$$\text{subject to}\ \hat{\mathbb{E}}_t\left[\mathrm{KL}[\pi_{\theta_{old}}(.|s_t), \pi_\theta(.|s_t)]\right] \leq \delta \tag{2.37}$$

where $r_t(\theta) = \frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)}$ is the probability ratio of policy update. Note that $r_t(\theta_{old}) = 1$, i.e. the ratio is one if no changing in policy. In other conditions where the $r_t \neq 1$ the new policy is different from the old one, the magnitude of the difference form $r_t(\theta_{old})$ indicates how different the new policy is.

2.2.2.3 Proximal Policy Optimization

The Proximal Policy Optimization (PPO) algorithm updates the agent policy with a regularization, that prevents stepping too far from the previous policy. This is beneficial since high policy updates can lead the model to unwanted policy too fast, thus this effect can cause learning instabilities. Some algorithms use second-order derivatives (e.g. TRPO) while PPO use first-order derivatives with simple ideas as penalty or a use of clip function. The PPO provides same or grater results on several benchmarks [30] with much greater simplicity than vanilla, Policy Gradient, TRPO, and so on. [30]

As mentioned, one variant of PPO is using clip surrogate objective. This is the simplest and seems to be the best variant of PPO. It simply clips the $r_t(\theta) = \frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)}$ probability ratio if it would step outside the $[1-\epsilon, 1+\epsilon]$ region, where ϵ is a new hyperparameter. The introduced mechanism prevents too big changes on the agent's policy. The formula of the objective with clipping is the following:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}\left[\min\left(r_t(\theta)\hat{A}_t,\ \ \mathrm{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t\right)\right] \tag{2.38}$$

In order to demonstrate the operation of this loss, let's take a look on the formula in a given time step t. Let's assume that $r_t < 1+\epsilon$ and $r_t > 1-\epsilon$ is fulfilled, than the objective is remains untouched related to the L^{CPI}, i.e min function returns $r_t(\theta)\hat{A}_t$. If $r_t < 1+\epsilon$ is not satisfied, than min function returns $r(1+\epsilon)\hat{A}_t$. In the opposite side if $r_t > 1-\epsilon$ is not valid, than it returns $r(1-\epsilon)\hat{A}_t$. The loss can be visualized as the following, where we can differentiate two case based on the sign of the advantage $\hat{A}_t$, since it determines the slope of the loss function at the unclipped domain. [30]

The other variant of PPO is using penalty on KL-divergence with an adaptive penalty coefficient β [30]. The objective function takes the following form:

$$L^{KLPEN}(\theta) = \hat{\mathbb{E}}_t\left[r_t(\theta)\hat{A}_t - \beta\mathrm{KL}[\pi_{\theta_{old}}(.|s_t), \pi_\theta(.|s_t)]\right] \tag{2.39}$$

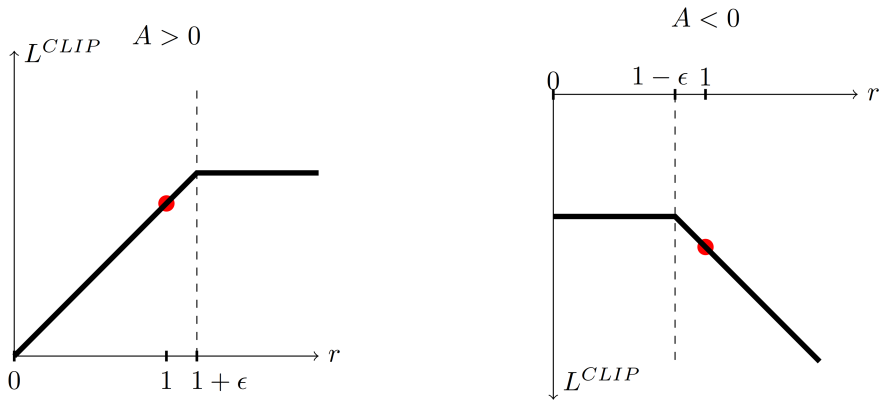


Figure 2.12: Objective function of PPO clip variant. (source: [30])

In every iteration of policy update we have a prior or in the case of first iteration an initial coefficient β. The algorithm runs several epochs of minibatch SGD to optimize the penalized objective function. Then calculates the new coefficient β for the next iteration, based on the following formula [30]:

$$\beta = \begin{cases} \beta/2 & \text{,if } d < d_{targ}/1.5 \\ 2\beta & \text{,if } d > 1.5 d_{targ} \end{cases} \tag{2.40}$$

,where:

$$d = \hat{\mathrm{E}}_t \left[\mathrm{KL}[\pi_{\theta_{old}}(.|s_t), \pi_\theta(.|s_t)] \right] \tag{2.41}$$

$$d_{targ} = \hat{\mathrm{E}}_t \left[\beta \mathrm{KL}[\pi_{\theta_{old}}(.|s_t), \pi_\theta(.|s_t)] \right] \tag{2.42}$$

2.3 Related work

Solutions can be found in the literature which have discovered that processing temporal information can improve performance in a variety of tasks. In this chapter, different solutions will be presented which try to achieve the same improvement as my solution, i.e. to increase the accuracy of the estimation by taking temporality into account.

A solution [16] for lane following uses frame stacking, which is a simple trick of stacking image-based observations along their channel dimension. The solution only provides temporal information indirectly, but offers promising results in the Duckietown environment without any recurrent network or sequence modelling. This clever workaround is worth comparing with the other sequence model-based solutions.

Dreaming with Transformers [40] proposes an effective world model-based architecture, what is able to learn directly from high-dimensional inputs with the help of latent imagination. Predictions of probable actions is projected into imagination space with GTrXL model. The solution uses actor-critic RL algorithm to train the network. The solution tested in DM Lab and VISTA driving simulator, making

this solution a good starting point for self-driving. The dreamer algorithm called dreamer because it still can learn representation while completely detached from source observation for a several timesteps [40]. This is an interesting advantage, which is achievable with taking temporal information into account.

There are many algorithms we can choose from to train our agent in reinforcement learning. Some solutions use the same model for acting in the environment and learning from experience, while other solutions use a distributed RL setting, where the previous tasks are divided between the learner and the actors. A common practice in the field of distributed RL to run the learner on accelerated hardware such as GPU and run the actors on unaccelerated hardware as CPUs [28]. The actor-learner distillation [22] research published in 2021 and proposes efficient way to do distributed RL training with transformers. The solution aids the problem of actor-latency, what is actually the unwanted time delay caused by the actor slowness on the unaccelerated hardware and the data acquisition speed from finished actors to learner [22]. The architecture uses a GTrXL for the learner model, to utilize transformer model's sample-efficiency and LSTMs for actors, to maintain LSTM's computational-efficiency. This layout of training is a forward-looking example to show the potentials of parallelization in the case of reinforcement learning and a possible way to make GTrXL training more efficient.

Chapter 3

Research objectives

The development of a lane following function raises many problems and challenges. In my work, I want to investigate the following concepts.

I would like to be able to train a deep neural network based agent for lane following behavior with deep reinforcement learning. I would like to confirm that Proximal Policy Optimization algorithm is appropriate for teaching agents in complex environments. The outcome of all these will be a tested infrastructure in which different solutions can be compared in a consistent and appropriate way.

If the training infrastructure works, I would like to answer the question of whether the temporal information integration promotes the lane following functionality. If it is possible to consider temporal information, I would like to examine the cases where it performs better and worse. To answer these question I would like implement four different deep neural network models and compare their abilities.

The comparison will be based on a CNN model that does not have sequence modelling capabilities in its current use. The purpose of this, is to examine how time-independent predictions perform. The other models will be hybrid models consisting of a CNN and a sequential model in a stacked manner. In this way, the four models intended for comparison will be a CNN, a CNN with frame stacking, a CNN with LSTM and a CNN with Transformer model. The comparison will be based on indicators of lane following performance, model complexity and computational demand.

Chapter 4

System design

Before presenting the experiments in details, I would like to describe the software and the hardware environment I used.

4.1 Duckietown platform

Duckietown is an open-source, cost-effective and flexible platform that allows participants to research and develop autonomous driving. It enables to pursue development in a fully fledged, scaled-down, cheaper world while retaining the features and effects of real world driving scenarios. These properties make it possible to observe and practice the whole process of development in an affordable way for instructors, students, professionals and researchers [25].

The platform consists of small vehicles (Duckiebots), cities (Duckietowns) and citizens (duckies). Duckiebots are three-wheeled, DC motor-driven robots with a monocular camera. The most recent DB21 duckiebots are equipped with Nvidia Jetson Nano developer kit, which even compatible for deep neural network deployment with low power consumption in a small unit. A duckietown is easily customizable and reproducible, since it built of modular road tiles, traffic lights and obstacles.

Universities around the world use the platform to develop and test their best and most reliable self-driving algorithms against others. An annual competition called Artificial Intelligence Driving Olympics (AI-DO) is offering different challenges for participants in the duckietown system. The competition consists of a pre-selection and a final. In the pre-selection phase the developers with the best submitted solutions are picked based on duckietown's official unified metrics. At the final phase the qualifiers have another chance to submit their best solutions, which are evaluated in real world environment called duckietown autolab. This kind of autolabs, sometimes called robotariums can be found at Swiss Federal Institute of Technology in Zürich (ETHZ) and at Toyota Technological Institute at Chicago (TTIC). The winners and rankings are determined on the robot's performance, which is described

Figure 4.1: Duckiebot and Duckietown real environment. (source: [12])

with above mentioned official duckietown metrics. The most important metrics are the followings [1].

- ***Survival time:*** The time spent on the road. If the robot runs off the road the simulation stops.
- ***Major infractions:*** The time spent outside of the derivable zones.
- ***Traveled distance:*** The distance traveled in lane.
- ***Lateral deviation:*** The lateral deviation from right lane center.

Figure 4.2: Duckietown autolab at Zürich. (source: [11])

Competitors can compete in quite a few different challenges, where the most important are the followings.

- ***LF - Lane following:*** Following the lane without any obstacles.

[1] `https://challenges.duckietown.org/v4/humans/challenges/aido-LF-full-sim-validation`

- ***LFP – Lane Following with Pedestrians:*** Lane following with avoiding duckie pedestrians.
- ***LFI – Lane Following with Intersections:*** Lane following with intersections which must be crossed by the agent.
- ***LFV_multi - Lane Following with dynamic Vehicles:*** The agent is running on multiple duckiebot and executing lane following with the presence of other vehicles.
- ***LFVI_multi - Lane Following with dynamic Vehicles and Intersections:*** The agent is running on multiple duckiebot and executing lane following with the presence of other vehicles and intersections.

These challenges can be evaluated in 3 different way. One of them is the validation in simulation, where the map and output of the simulation with additional evaluation metrics are published. An other is the test in simulation, where only the simulation evaluation metrics are shown, the maps are hidden from participants. The more difficult evaluations are realized in the real world at robotariums. Those who have done well in the simulation can have their solutions evaluated in real world. Here, the maps and the metrics are both visible.

The platform is perfect for implementing and developing agents for autonomous driving with reinforcement learning algorithms, since as mentioned, the duckietown environment is constructed in simulation and also in the real world. The training and the early testing stage of the agent is usually realized in simulation, since this way it takes less time, money cost and effort. The real-world testing is generally the final stage of the development, to test the promising agents in action.

4.2 Software and hardware environment

To train the hybrid model in a reinforcement learning setup I used the open-source Ray framework[2]. The framework consists of several sub-libraries, such as Ray Tune (hyperparameter search library) or Ray Train (distributed deep learning library). One of them is Ray RLlib, as its name suggests it is a reinforcement learning library, which offers predefined RL algorithms, distributed RL capabilities, simple API and much more. Fortunately, it easily handles custom environment and gym environments, such as duckietown simulator. Since my solution required custom model build and fully adjustable CNN architecture to investigate the agent performance in case of architectural changes, I implemented a custom CNN model as a part of the architecture in PyTorch framework with the help of RLlib's TorchV2 model class.

I ran all of my experiments on an Nvidia DGX station which has Ubuntu 18.04 LTS operation system, 4 Tesla V100 32GB GPUs, 40 Intel Xeon CPUs. I tried extensive hand-tuned hyperparameter configurations with one GPU and ten CPU per agents.

[2] `https://docs.ray.io/en/latest/index.html`

For hyperparameter search I ran 2 simultaneous agent threads per experiment with a reduced number of CPUs (5 per agents) for better paralellization.

For my experiments I used two different logger framework. Since the default one of the Ray framework is Tensorboard, I used it from the very beginning of my works. The disadvantages (e.g. local running on the host machine, difficult to categorize runs) led me to use Weights & Biases (WandB) framework, which is intuitive, cloud-based logger framework with includes hyperparameter optimization (called sweep) and report creation tools.

Chapter 5

Methods and implementation

I trained various models using reinforcement learning in the Duckietown environment. The aim was to get the best possible lane following behavior from each model, which was achieved by significant amount of hyperparameter optimization with the help of Ray and WandB frameworks. I evaluated and compared the performance of the best trained models in the Duckietown simulator. Based on the results, I have drawn the consequences about the applicability of each model for lane-following functionality.

5.1 Models to compare

The basic concept of comparing model architectures with different sequence modelling capabilities are came from the requirement that we may need to reasoning over a time span to enhance more performance and safety for autonomous driving. I chose the different architectures to cover as much of the range of temporal information processing capability as possible, according to the literature.

One of the difficulty of the problem is that most of the sequence modelling solutions are capable for supervised learning, but they are not applicable for reinforcement learning. The other complication is that sequence to sequence models are mostly used in the field of NLP, thus they destine sequence of embedding, which are vectors or scalars, but in our case the observations are tensors of camera frames captured by the duckiebot.

5.1.1 Convolutional Neural Network (CNN) based model

I would like to introduce the architecture of the CNN model, which has no sequence modelling capabilities in this setup. The used Proximal Policy Optimization required to define the model (figure 5.1) in an actor-critic structure. I implemented it as a custom torch model into RLlib to serve as a baseline, while preserving its modularity for later architectural modifications.

I built a custom torch CNN (figure 5.1) as mentioned in an actor-critic or sometime referred as policy-value architecture. It built from two models, which are actually

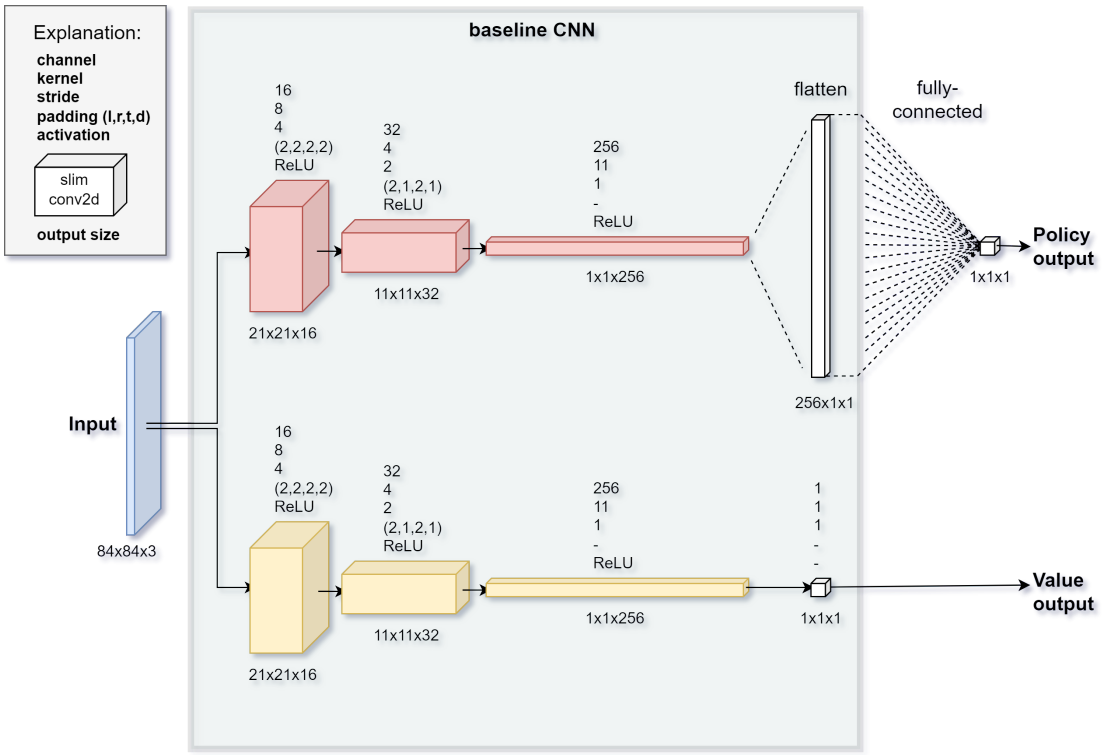


Figure 5.1: The baseline CNN model in detail.

defined as two separate branch with common input, but with distinct weights and outputs. Both branch are built from slimConv[26] convolution layers, which are boost performance of the network by decreasing channel redundancy.

This architecture is created based on the default RLlib vision model, which is a generally appropriate network for various partially observable environments with visual observations. I find it an interesting and useful practice to not to use an MLP head on the top of the branches, rather applying a convolution with a kernel size identical to the dimensions of the previous layer's channels. This way we can achieve the same result in dimension reduction, but depends on the situation we can have less learnable parameters for our network.

The framed "baseline CNN" component provides the foundation for all models which I will use in for the comparison. Later, only the input and the post-processing (fully-connected layer in this case) will be changed to fit in to the new architectures.

5.1.2 CNN with frame stacking model

This model is using the same architecture as CNN model (5.1.1), except that it receives 5 stacked images at its input. The 5 images with dimension of $(84 \times 84 \times 3)$ are stacked over the channel dimension, i.e. the input dimension is $(84 \times 84 \times 15)$.

This is a one step ahead addition to reach dynamics and temporal information. However, this workaround does not considered as sequence modelling. It may receive information about the past, but it cannot treat the input as a time-dependent sequence, since the stacked observations are fed to the network all together without

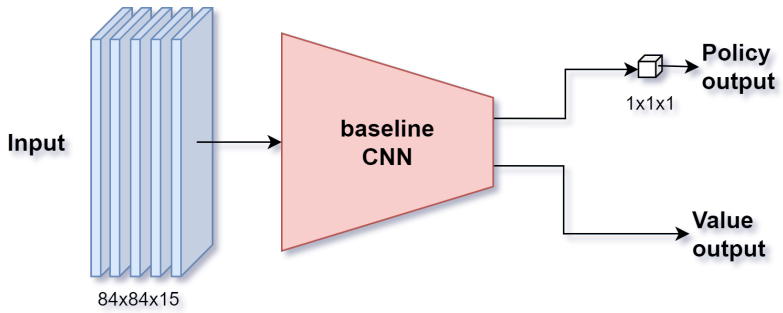


Figure 5.2: The data flow of frame stacking with CNN.

distinction over time. This means that the network may experience that all the pictures were taken at one particular moment. Furthermore, the number of frames to be stacked while still converging is limited to a few, while sequence models are capable to consider longer spans (like in my experiments 64 observations or way more above it).

5.1.3 CNN with LSTM model

Motivated on several sequence model aided RL solutions [4][40], I applied the baseline CNN as an encoder unit on the incoming camera frames. The CNN encoder in my solution maps each camera image to a latent vector. The encoder part does not provide temporal information, it only transforms our input to a more manageable representation, while adds more complexity and capability to the network via its learnable weights, see on figure 5.4.

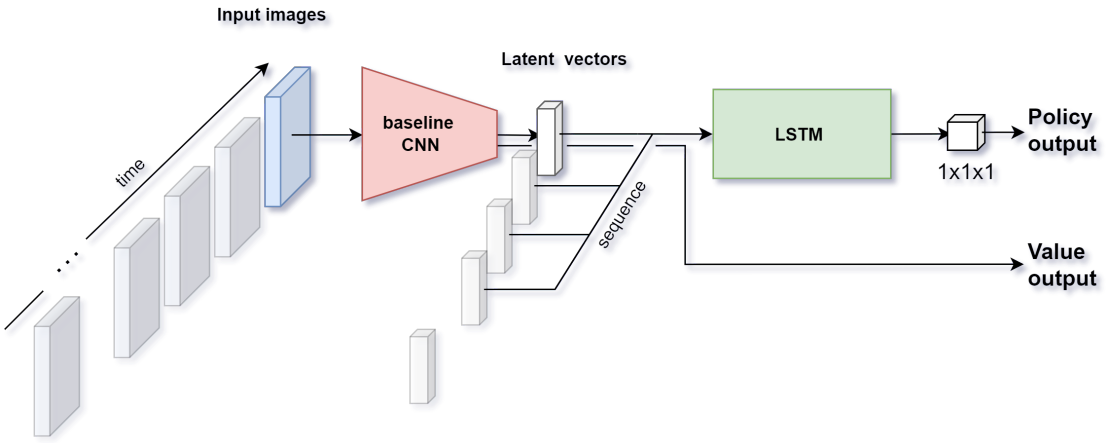


Figure 5.3: The CNN-LSTM hybrid model architecture.

I picked the LSTM for sequence modelling, since it is a widely used architecture for this purposes. I choose the input sequence length to be 64 for the LSTM, thus this much past observation are considered by the model. In the case of LSTM, this is done so that the Backpropagation Through Time (BPTT) process only performed on the previous 64 states, the older ones are truncated.

5.1.4 CNN with Transformer model

The final architecture utilizes a state-of-art model GTrXL with attention-mechanism and sequence modelling abilities. Based on previous researches [4][40][22], my choice fell on the GTrXL model, since this is the only transformer model which can stabilize learning in RL setup.

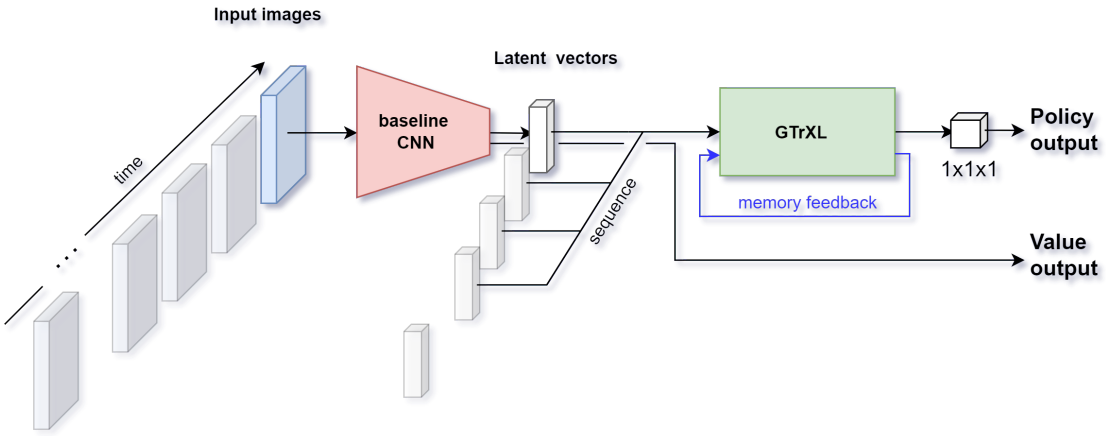


Figure 5.4: The CNN-GTrXL hybrid model architecture.

The pipeline is same as at the CNN-LSTM model, but the LSTM is changed with GTrXL model (figure 5.4). The input sequence length is also 64. Additionally, the model applies a memory feedback which stacks the past observations and adds them back to the input.

5.2 Training

I trained all the agents in the Duckietown environment. I have endeavoured to provide identical conditions in all aspect for each model. The simulation environment proprieties, the software and the hardware configurations were same for all trainings. Only the PPO specific and model specific parameters were optimized to achieve the best lane following behavior with each model.

The map set I used consists of the following eight tracks.

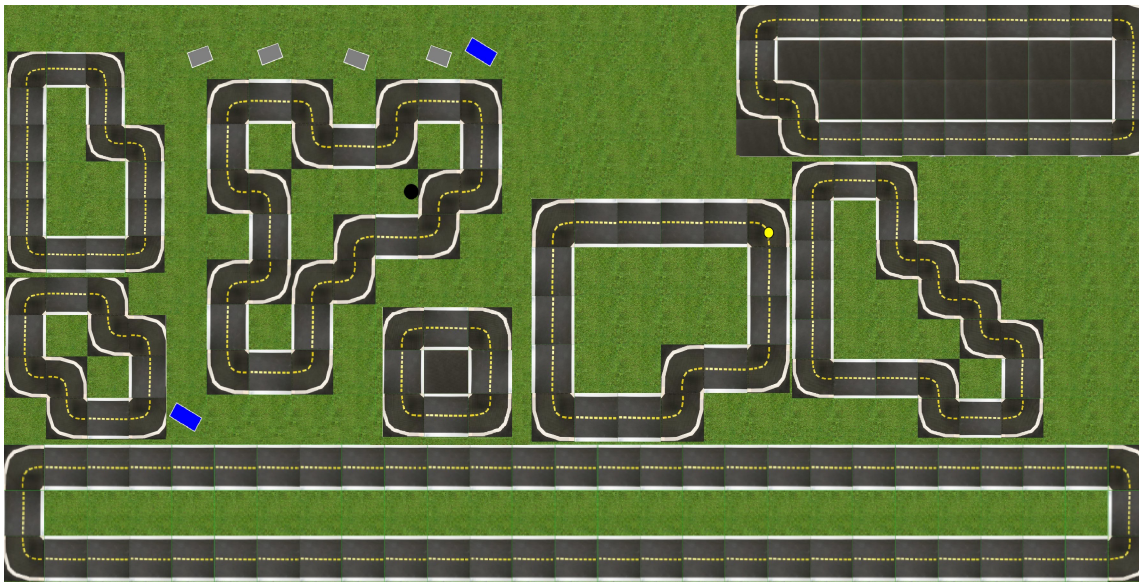
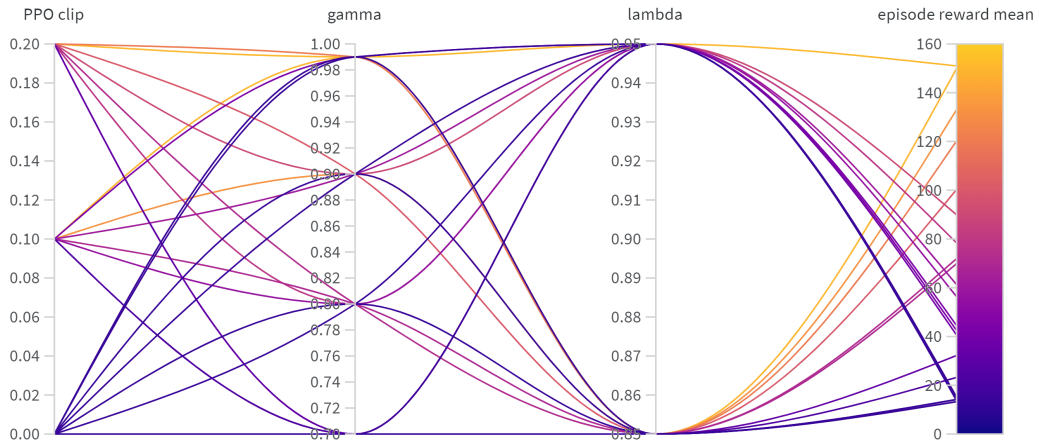


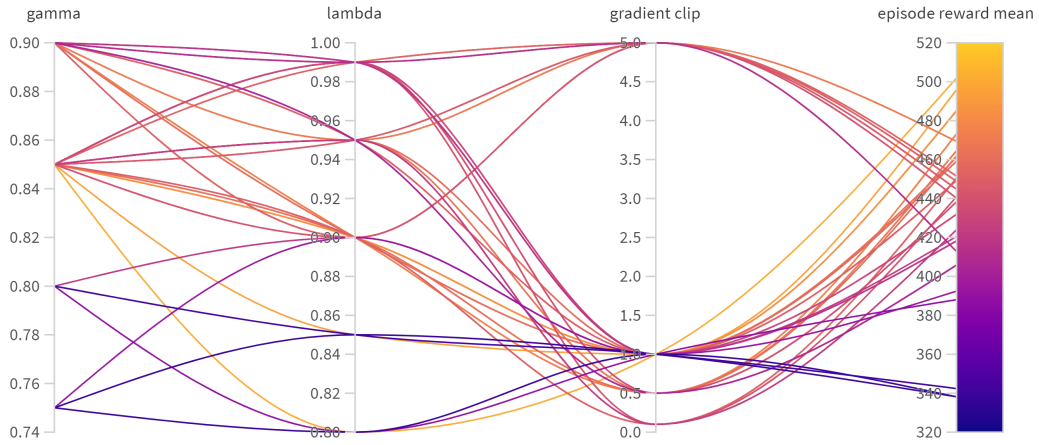
Figure 5.5: The top-down view of all maps, which are used for training.

The Duckietown simulator implemented as a gym environment, thus I could define and manipulate its behavior via wrappers. Such wrappers can be defined for observations, actions and rewards. In my solution, I did not use the original wrappers of Duckietown environment, but an improved version of them from an open-source solution [16] of my university colleague. In my solution the used observation wrappers are preprocessing the frames taken by the agent with image cropping, image resizing and image normalization. The action wrappers are transform the model output to wheel velocities of the duckiebot agent. While reward wrappers are giving rewards based on agent heading angle and velocity.

I started trainings with my best prior guess for initial hyperparameters and I only performed a rough learning rate optimization with deploying some single runs for all models. As the next step, I optimized RL and PPO specific hyperparameters with grid search on every model to reach the maximum episode reward mean. The hyperparameter space is stretched by the variations of the following parameters: ppo clip, gamma, lambda and gradient clip. I would like to introduce the results of the CNN and CNN-GTrXL models.



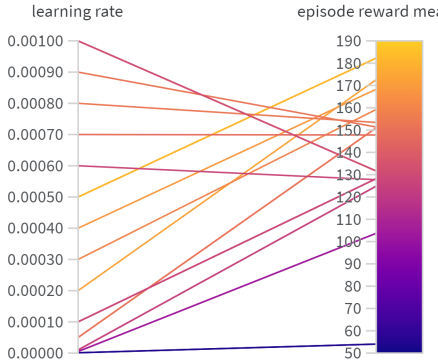
(a) CNN



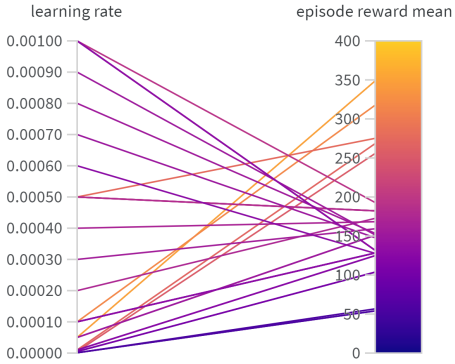
(b) CNN with GTrXL

Figure 5.6: Hyperparameter optimization results for RL and PPO specific parameters. In this case, the parallel coordinates plot shows, how much episode reward mean is achieved for the end of the training for each individual hyperparameter selection.

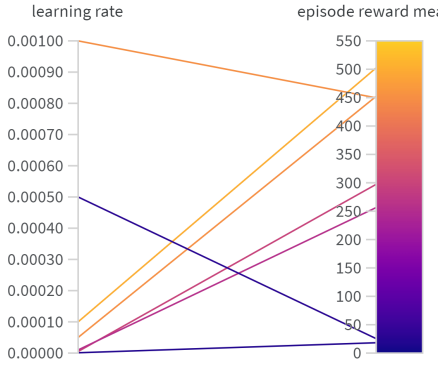
As figure 5.6 shows, the GTrXL model has learned better with parameters that are not yet completely tuned. The experiment was also performed for the CNN with frame stacking and for the CNN with LSTM model. I found that the baseline CNN and the frame stacked version needed similar RL and PPO specific parameters. Also, the LSTM and GTrXL versions needed the same parameters. Lastly, I performed fine learning rate tuning for each model.



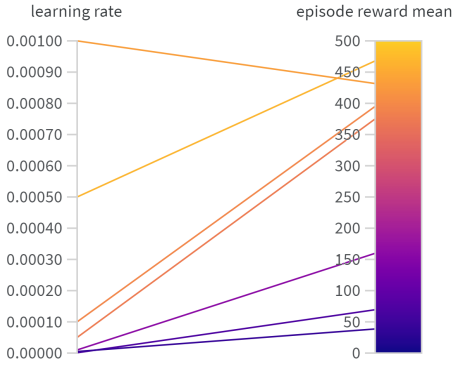
(a) CNN



(b) CNN with frame stacking



(c) CNN with LSTM



(d) CNN with GTrXL

Figure 5.7: The results of fine tuning learning rate.

With the best hyperparameters the final models are trained for 2 million steps to ensure that we give the agents enough steps to learn and that we do not stop trainings too soon. We continuously checkpoint the model states with the maximum episode reward mean and at the end of the training we restore the best checkpoints. These models are used for the the evaluation and comparison.

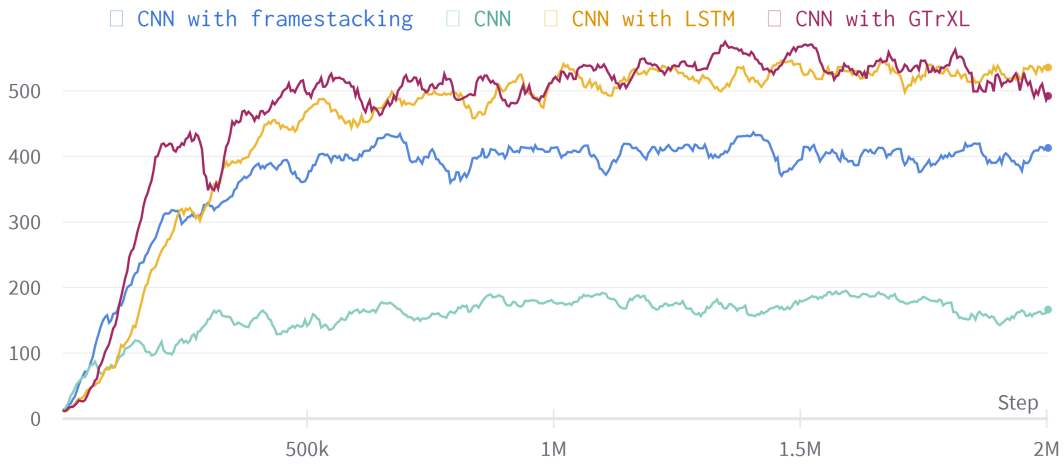


Figure 5.8: The final trainings with the best hyperparameters for 2 million steps.

Chapter 6

Evaluation and results

I evaluated the performance of the four different models from several perspectives such as model complexity, driving performance and computational demand. The evaluations were done on two self-made test track that the agents had never seen during the training process.

6.1 Model complexity

The number of total and learnable parameter is a common used indicator for model complexity. In this work the CNN has the fewest and hybrid sequence models have the most parameters. I have chosen the LSTM and GTrXL model settings to have the same number of parameters so we can compare them more accurately.

Table 6.1: The number of parameters of different models.

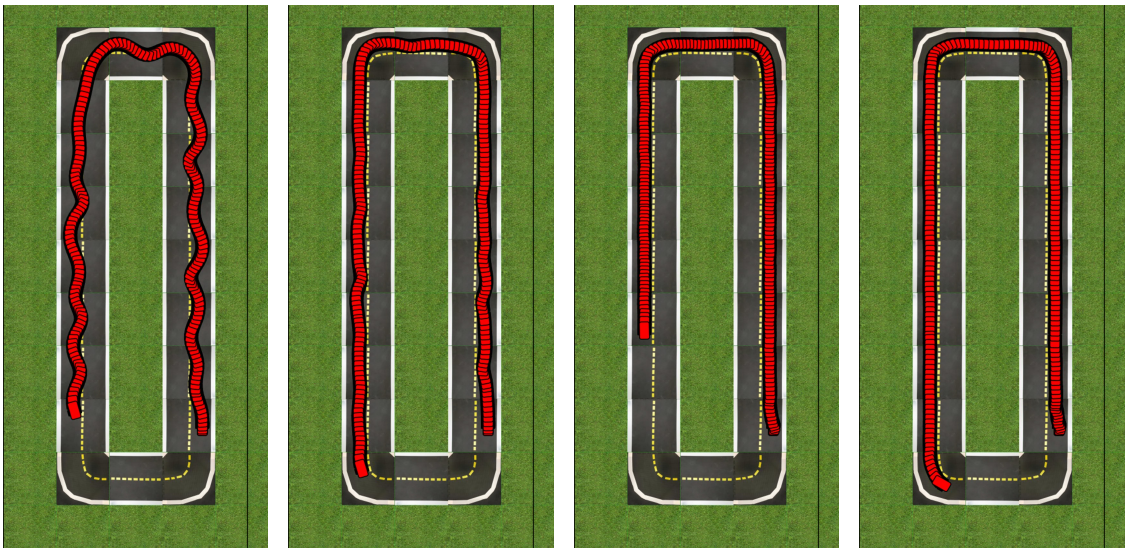
	CNN	CNN with frame stacking	CNN with LSTM	CNN with GTrXL
number of total parameters	2.006M	2.031M	6.641M	6.639M
number of learnable parameters	2.006M	2.031M	6.641M	6.639M

6.2 Driving performance

The driving performance comparison of the models was evaluated with the help of the official metrics of Duckietown platform and on some reinforcement learning specific indicators. I used two test tracks, one easier with mostly straight parts and a harder which has more curvy parts. All the agents started in the same position with the same pose and they got 20 seconds to act in the environment.

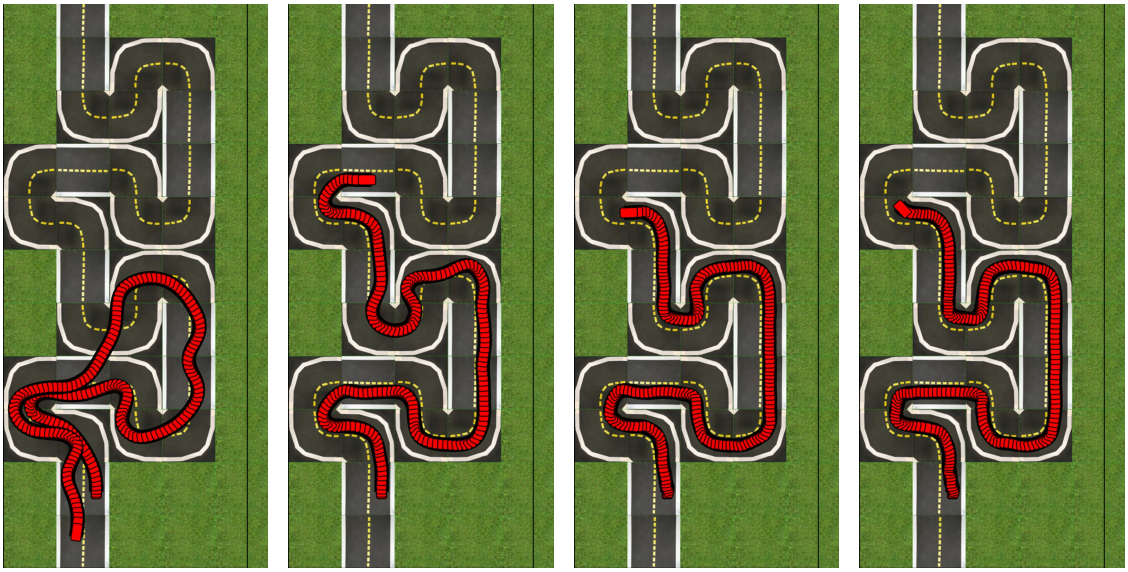
6.2.1 Agent trajectories

The top-down view visualization of the agent trajectories during the test simulations on the easy map can be seen on figure 6.1 and for the hard test map on figure 6.2. Also, a video is made from the related trials of the evaluation [1]. The agents are started the trials from the bottom-right and they were driving upward from the perspective of the diagram.



(a) CNN **(b)** CNN with frame stacking **(c)** CNN with LSTM **(d)** CNN with GTrXL

Figure 6.1: Agent trajectories[1] on the easy test track.



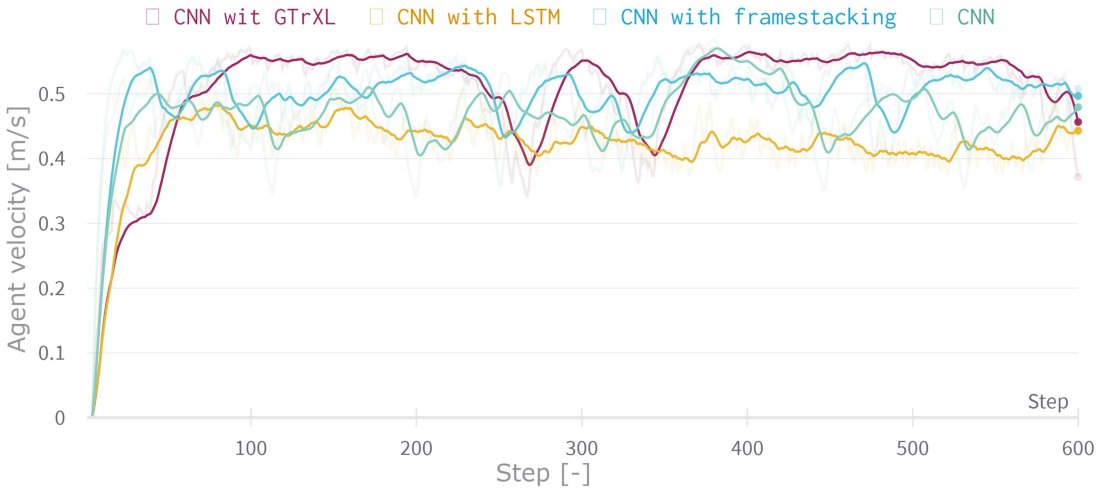
(a) CNN **(b)** CNN with frame stacking **(c)** CNN with LSTM **(d)** CNN with GTrXL

Figure 6.2: Agent trajectories[1] on the hard test track.

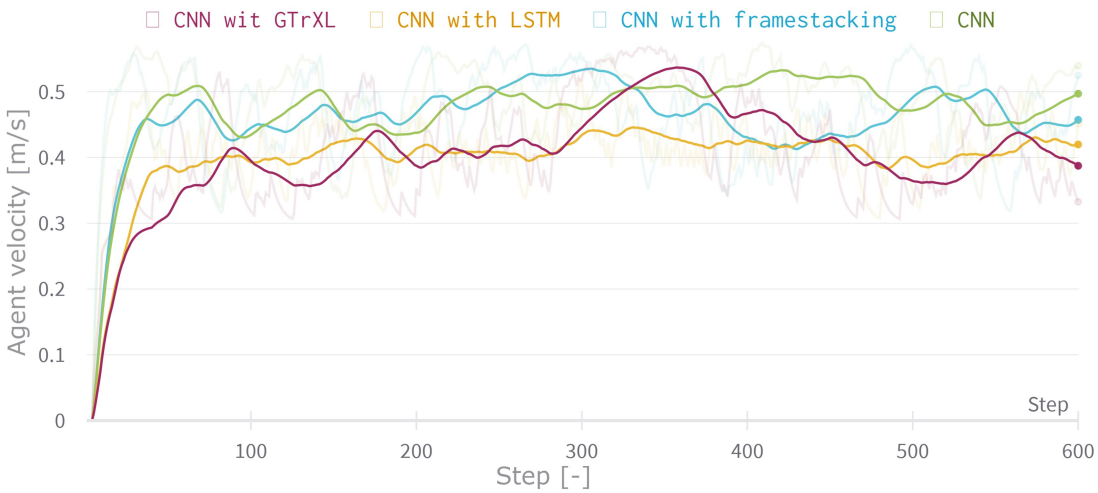
[1] https://youtu.be/xZV_Txv8nx8

6.2.2 Agent velocity

The agent velocity shows the different agent velocities during the evaluation. It is noticeable that the velocities are continuously changing and transients are also observable, this is because there is no fixed speed or only heading angle control, rather we control the two DC motor of the robots. In this way, the simulation models the motion by taking the dynamics into account.



(a) On easy map.



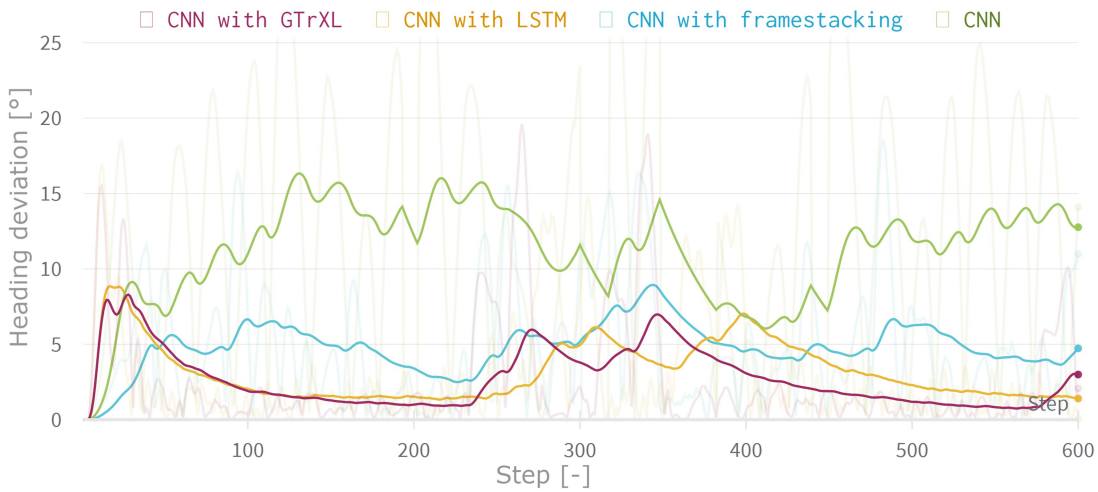
(b) On hard map.

Figure 6.3: Agent velocities during the evaluation.

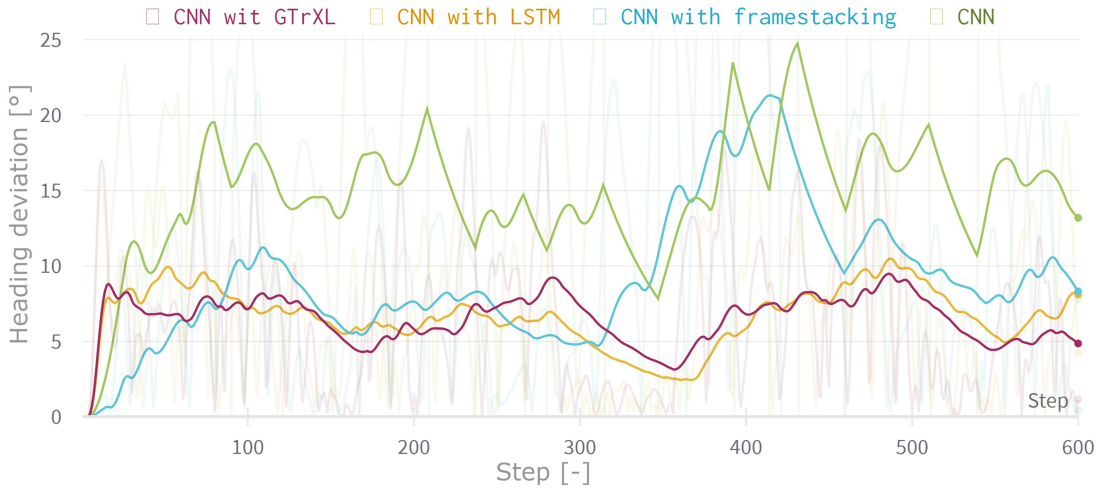
6.2.3 Accuracy

The lane following accuracy is measured in heading deviation and in lane center deviation directly. Also, have to mention that in later statistics, one part of the cumulative reward added from these properties indirectly.

The heading deviation from the correct direction in every timestamp indicates how well the agent aiming the right direction during the evaluation (figure 6.4). Furthermore, the centerline deviation shows the distance difference from the right lane center (figure 6.5).

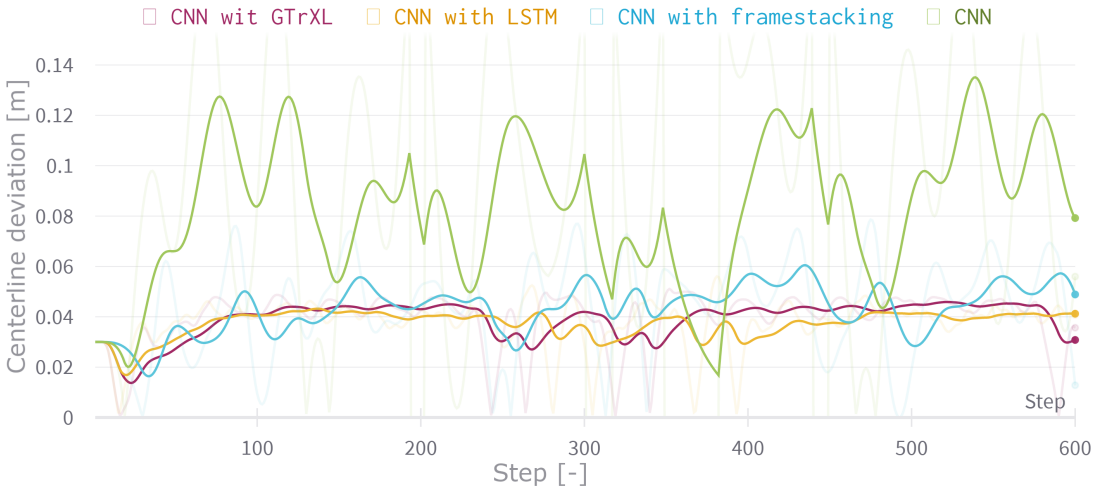


(a) On easy map.

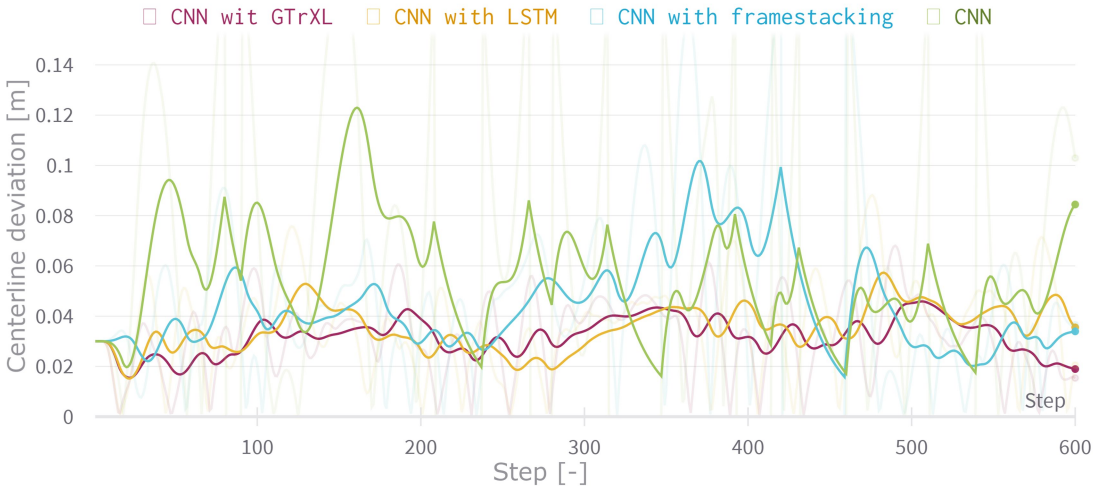


(b) On hard map.

Figure 6.4: Heading deviation of each agent during the evaluation.



(a) On easy map.



(b) On hard map.

Figure 6.5: Deviation from right lane center of each agent during the evaluation.

6.2.4 Comparison

As can be seen, the measurements shown above are the result of only one experiment. For the final analysis, I started each model from ten different locations on both the two test map and I recorded the metrics all along the way. The results of these measurements are summarized in table 6.2.

Table 6.2: Summary of driving performance indicators.
(The metrics are represented in mean±2·standard deviation format.)

		CNN	CNN with frame stacking	CNN with LSTM	CNN with GtrXL
cumulative reward[-]	easy	233.2 ± 24.4	441.7 ± 31.2	$\mathbf{473.6 \pm 12.8}$	461.4 ± 9.1
	hard	129.5 ± 159.7	406.9 ± 50.1	504.5 ± 19.1	500.8 ± 24.0
heading deviation[°]	easy	4.61 ± 0.57	2.18 ± 0.34	$\mathbf{1.05 \pm 0.08}$	1.06 ± 0.16
	hard	2.63 ± 3.24	3.36 ± 0.67	2.17 ± 0.18	$\mathbf{2.11 \pm 0.18}$
centerline deviation[m]	easy	1.62 ± 0.10	0.76 ± 0.14	0.76 ± 0.02	$\mathbf{0.76 \pm 0.02}$
	hard	0.80 ± 0.96	0.86 ± 0.16	0.71 ± 0.04	$\mathbf{0.62 \pm 0.02}$
speed[$\frac{\text{m}}{\text{s}}$]	easy	0.48 ± 0.01	$\mathbf{0.50 \pm 0.01}$	0.42 ± 0.01	0.49 ± 0.02
	hard	0.48 ± 0.01	0.46 ± 0.01	0.41 ± 0.01	0.42 ± 0.01
in wrong lane[s]	easy	2.06 ± 1.32	0.00 ± 0.00	$\mathbf{0.00 \pm 0.00}$	$\mathbf{0.00 \pm 0.00}$
	hard	2.22 ± 4.42	0.25 ± 1.00	$\mathbf{0.00 \pm 0.00}$	$\mathbf{0.00 \pm 0.00}$
driven distance[m]	easy	8.90 ± 1.43	10.02 ± 0.16	8.49 ± 0.13	9.95 ± 0.30
	hard	4.96 ± 6.00	8.97 ± 0.52	8.18 ± 0.13	8.36 ± 0.20
driven lane distance[m]	easy	8.19 ± 0.80	9.97 ± 0.13	8.47 ± 0.12	9.93 ± 0.30
	hard	3.98 ± 4.52	8.81 ± 0.54	8.14 ± 0.12	8.32 ± 0.18

The 20 seconds long measurement time with 30 step/sec frame rate means 600 maximum environment step for the evaluation. The cumulative reward (also called return) $R = \sum_{t=0}^{600} r_t$ is the summed up reward for all of these 600 environment step. This shows how well the agent performs in regards to reward functions, which

The mean speed, the in wrong lane and the driven distance indicators are straight forward, but worth to mention that the driven lane distance shows how far the agent has travelled in the direction of the lane. In other words, this is the effective travelled distance.

6.3 Computational demand

Most of the real-world applications require fast inference time from the algorithm. This applies even more to autonomous driving, where the decision must be made in a split of a second. Also high computational demand and inference time can cause anomalies and unreliable control in these systems. At machine learning models the inference time is meaning the time for the trained model to predict an output.

I measured the inference and the environment step time during each evaluation. Valuable information can be obtained by observing the mean of collected time measurements. The results are summarised in the following table 6.3.

Table 6.3: Summary of inference times.
(The metrics are represented in mean±2·standard deviation format.)

		CNN	CNN with frame stacking	CNN with LSTM	CNN with GtrXL
inference time [ms]	easy	5.95 ± 0.10	7.68 ± 0.12	6.86 ± 0.15	27.69 ± 0.20
	hard	5.98 ± 0.21	7.66 ± 0.15	6.87 ± 0.07	27.76 ± 0.83

Chapter 7

Conclusions

The statistics of the evaluation show that every investigated model has different lane following performance. The considered tasks, aspects and metrics are highlighted some relevant properties of each model. The outline of the most significant conclusion based on the comparison of the four different architectures:

- **Temporal information improves driving performance.** All models which are took into consideration more observation or states were driving more robust with higher lane following accuracy and earned twice or thrice as much cumulative reward as the simple CNN.
- **CNN with GTrXL model gives more performance especially in hard tasks.** On both easy and hard maps the GTrXL variant had the best heading deviation and centerline deviation results. Also, with even that good lane following accuracy it driven the longest absolute and effective distances and had higher mean robot speeds than LSTM variant. Also a logical behavior is noticable at this model, that its robot speed was dynamic, i.e. the agent accelerated at straight parts even more, see figure 6.3a.
- **The LSTM and GTrXL hybrid model variants are outperform the other variants,** in almost every metrics. The only aspect where other models performed better was the robot speed which meant sacrificing accuracy for them (table 6.2).
- **The CNN model without temporal information was not able to perform the tasks in an acceptable way.** The lost trajectory at figure 6.2a or low cumulative reward with high deviation at table 6.2 are indicates the unreliability of the model predicitions in a complex, dynamic dominated task as lane following. Also, the unintentionally spent time in the wrong lane indicator ought to be zero in autonomous driving.
- **Inference time of GTrXL variant is much higher.** The extra accuracy and learnability are required five times higher inference time. The good alternative can be to apply the LSTM variant, which has same 6.6M learnable parameters (table 6.1) and its inference time is only a few milisencond higher than simple CNN's (table 6.3).

Chapter 8

Summary

The aim of the work was to find proper solution for temporal information integration into deep reinforcement learning to improve lane following functionality.

The applicable model architectures, such as recurrent neural networks, sequence models, transformer models are investigated in the first part of the work. This is followed by a literature review of reinforcement learning, with a particular focus on Policy Gradient algorithms. During the experiments, there are four different architectures, such as CNN, CNN with frame stacking, CNN with LSTM and CNN with GTrXL were trained for lane following with reinforcement learning in the virtual environment of Duckietown platform. The sensitive reinforcement learning and model specific parameters were extensively searched via hyperparameter optimization. All the four trained policies evaluated and compared based on driving performance, model complexity and computational demand. The work showed that utilize memory behavior with hybrid sequence models can outperform feed-forward models with time-independent predictions in lane following task.

The work presented in this paper is complete. Although, like everything else, it can be further developed. My thoughts about the future works are the followings.

Adapting agents from simulation to real life can be challenge, where the diverse environment and the hardware capabilities are limiting the performance of the agents. It would be interesting to investigate how hybrid sequence models can adapt to real world with the techniques of domain randomisation, dynamic randomisation or model quantization.

Also, the Duckietown infrastructure itself offers many challenges with more complex tasks, such as lane following with other vehicles or lane following with intersection crossing task. Evaluate the performance of models with temporal information processing in more complex tasks would certainly beneficial.

A future plan can be to examine the behavior of recurrent and transformer models with even longer past sequences. In these setup, the effect of the attention-mechanism presumably would be much more significant.

Acknowledgements

I would like to thank both of my supervisors, Dr. Bálint Gyires-Tóth and Róbert Moni for their support during the research. I am grateful for Bálint's professional insights that helped me a great amount in creating better documentation and maintaining the progress of the work throughout the semester. I feel thankful to Róbert, who introduced me the world of reinforcement learning years ago. Since then, he supported this research and my way of progress with professional advices and kind personal help at any time.

Also, the research presented in this work has been supported by the PIA Project, a collaboration project between Budapest University of Technology and Economics and Continental AI Development Center with the goal of supporting students' research in the field of Deep Learning and Autonomous Driving.

Bibliography

[1] Joshua Achiam. Spinning Up in Deep Reinforcement Learning. 2018.

[2] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38, 2017.

[3] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[4] Andrea Banino, Adrià Puidomenech Badia, Jacob Walker, Tim Scholtes, Jovana Mitrovic, and Charles Blundell. Coberl: Contrastive bert for reinforcement learning. *arXiv preprint arXiv:2107.05431*, 2021.

[5] Sneha Chaudhari, Varun Mithal, Gungor Polatkan, and Rohan Ramanath. An attentive survey of attention models. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 12(5):1–32, 2021.

[6] Keqiao Chen. Apso-lstm: an improved lstm neural network model based on apso algorithm. In *Journal of Physics: Conference Series*, volume 1651, page 012151. IOP Publishing, 2020.

[7] Jianpeng Cheng, Li Dong, and Mirella Lapata. Long short-term memory-networks for machine reading. *arXiv preprint arXiv:1601.06733*, 2016.

[8] Po-Wei Chou, Daniel Maturana, and Sebastian Scherer. Improving stochastic policy gradients in continuous control with deep reinforcement learning using the beta distribution. In *International conference on machine learning*, pages 834–843. PMLR, 2017.

[9] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*, 2019.

[10] Mohit Deshpande. Recurrent neural networks for language modeling. URL `https://gamedevacademy.org/wp-content/uploads/2017/10/Unrolled-RNN.png.webp`. Accessed: October 29, 2022.

[11] Duckietown. Research using duckietown, . URL `https://www.duckietown.org/research/guide-for-researchers?doing_wp_cron=1667052787.5364470481872558593750`. Accessed: October 29, 2022.

[12] Duckietown. Duckietown and the ai driving olympics., . URL https://www.aeapolimi.it/en/duckietown/. Accessed: October 29, 2022.

[13] Jan Gläscher, Nathaniel Daw, Peter Dayan, and John P O'Doherty. States versus rewards: dissociable neural prediction error signals underlying model-based and model-free reinforcement learning. *Neuron*, 66(4):585–595, 2010.

[14] Alex Graves. Long short-term memory. *Supervised sequence labelling with recurrent neural networks*, pages 37–45, 2012.

[15] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[16] András Kalapos, Csaba Gór, Róbert Moni, and István Harmati. Sim-to-real reinforcement learning applied to end-to-end vehicle control. In *2020 23rd International Symposium on Measurement and Control in Robotics (ISMCR)*, pages 1–6. IEEE, 2020.

[17] Raimi Karim. Attn: Illustrated attention. URL https://towardsdatascience.com/attn-illustrated-attention-5ec4ad276ee3. Accessed: October 29, 2022.

[18] Simeon Kostadinov. Understanding encoder-decoder sequence to sequence model. URL https://towardsdatascience.com/understanding-encoder-decoder-sequence-to-sequence-model-679e04af4346. Accessed: October 29, 2022.

[19] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[20] Benjamin Lindemann, Timo Müller, Hannes Vietz, Nasser Jazdi, and Michael Weyrich. A survey on long short-term memory networks for time series prediction. *Procedia CIRP*, 99:650–655, 2021.

[21] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.

[22] Emilio Parisotto and Ruslan Salakhutdinov. Efficient transformers in reinforcement learning using actor-learner distillation. *arXiv preprint arXiv:2104.01655*, 2021.

[23] Emilio Parisotto, Francis Song, Jack Rae, Razvan Pascanu, Caglar Gulcehre, Siddhant Jayakumar, Max Jaderberg, Raphael Lopez Kaufman, Aidan Clark, Seb Noury, et al. Stabilizing transformers for reinforcement learning. In *International Conference on Machine Learning*, pages 7487–7498. PMLR, 2020.

[24] Niki Parmar, Ashish Vaswani, Jakob Uszkoreit, Lukasz Kaiser, Noam Shazeer, Alexander Ku, and Dustin Tran. Image transformer. In *International Conference on Machine Learning*, pages 4055–4064. PMLR, 2018.

[25] Liam Paull, Jacopo Tani, Heejin Ahn, Javier Alonso-Mora, Luca Carlone, Michal Cap, Yu Fan Chen, Changhyun Choi, Jeff Dusek, Yajun Fang, et al. Duckietown: an open, inexpensive and flexible platform for autonomy education and research. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1497–1504. IEEE, 2017.

[26] Jiaxiong Qiu, Cai Chen, Shuaicheng Liu, Heng-Yu Zhang, and Bing Zeng. Slimconv: Reducing channel redundancy in convolutional neural networks by features recombining. *IEEE Transactions on Image Processing*, 30:6434–6445, 2021.

[27] Hojjat Salehinejad, Sharan Sankar, Joseph Barfett, Errol Colak, and Shahrokh Valaee. Recent advances in recurrent neural networks. *arXiv preprint arXiv:1801.01078*, 2017.

[28] Mohammad Reza Samsami and Hossein Alimadad. Distributed deep reinforcement learning: An overview. *arXiv preprint arXiv:2011.11012*, 2020.

[29] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR, 2015.

[30] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[31] Tian Shi, Yaser Keneshloo, Naren Ramakrishnan, and Chandan K Reddy. Neural abstractive text summarization with sequence-to-sequence models. *ACM Transactions on Data Science*, 2(1):1–37, 2021.

[32] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 27, 2014.

[33] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[34] Richard S Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2):181–211, 1999.

[35] Gaurav Tiwari, Arushi Sharma, Aman Sahotra, and Rajiv Kapoor. English-hindi neural machine translation-lstm seq2seq and convs2s. In *2020 International Conference on Communication and Signal Processing (ICCSP)*, pages 871–875. IEEE, 2020.

[36] Savvas Varsamopoulos, Koen Bertels, Carmen G Almudever, et al. Designing neural network based decoders for surface codes. *arXiv preprint arXiv:1811.12456*, 2018.

[37] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[38] Lilian Weng. Attention? attention! *lilianweng.github.io*, 2018. URL `https://lilianweng.github.io/posts/2018-06-24-attention/`.

[39] Lilian Weng. The transformer family. *lilianweng.github.io*, 2020. URL `https://lilianweng.github.io/posts/2020-04-07-the-transformer-family/`.

[40] Catherine Zeng, Jordan Docter, Alexander Amini, Igor Gilitschenski, Ramin Hasani, and Daniela Rus. Dreaming with transformers. 2022.

Appendix

RL and model hyperparameters.				
	CNN	CNN with framestacking	CNN with LSTM	CNN with GtrXL
learning rate α	0.0001	0.0001	0.0005	0.0001
batch size	4096	4096	4096	4096
PPO clip parameter	0.2	0.2	0.1	0.1
gamma γ	0.99	0.99	0.85	0.85
lambda λ	0.95	0.95	0.8	0.8
gradient clip	0.5	0.5	1.0	1.0
nr. of SGD iteration	16	16	16	16
SGD mini batch size	128	128	512	512
entropy coefficient	0.0	0.0	0.0	0.0
frame stacking depth	-	5	-	-
maximum sequence length	-	-	64	64
LSTM cell size	-	-	954	-
nr. of transformer unit	-	-	-	8
attention dimension	-	-	-	128
nr. of heads in MultiHeadAttention	-	-	-	3
head dimension in MultiHeadAttention	-	-	-	128
attention memory size	-	-	-	64
attention MLP output dimension	-	-	-	64
attention GRU gate bias	-	-	-	2.0
nr. of used previous action	-	-	1	3
nr. of used previous reward	-	-	1	3

Duckietown environment parameters	
seed	1234
episode maximum step	500
domain randomisation	False
dynamic randomisation	False
frame skip number	1
frame rate	30
accepted start angle	4
camera distortion	True