Incremental Transfer Learning for Reinforcement Learning in a Realistic Environment

D506e19

Autumn 2019
Project report P5

Aalborg Universitet
Selma Lagerløfs Vej 300
DK-9220 Aalborg Øst

House of Computer Science
Selma Lagerløfs Vej 300
DK-9220 Aalborg Øst
http://aau.dk

AALBORG UNIVERSITY

STUDENT REPORT

Title:
Incremental Transfer Learning for Reinforcement Learning in a Realistic Environment

Project Period:
5. semester

Project Group:
D506e19

Participants:
Jesper Adriaan van Diepen
Niki Ewald Zakariassen
Patrick Kampdal Petersen
Sebastian Lund
Simon Steiner

Supervisor:
Razvan-Gabriel Cirstea

Page Numbers: 93
Standard pages: 46 (Excl. Appendices)

Date of Completion: 19/12 - 2019

Abstract:

The problem of applying transfer learning methods to realistic reinforcement learning problems, is of interest due to the shown usefulness of transfer in neural networks and the recent developments in reinforcement learning. We have built incrementally more difficult racing tracks to be able to show how the transfer of knowledge can affect training time and performance. The methods used for transfer involved two approaches, direct and indirect transfer of knowledge respectively transferring weights between agents and learning by imitating the actions of another expert agent. We found that the indirect approach generally works best, specifically the variant where the transferred agent is used as the expert for the next track i.e. continuous track transfer, suggesting that it might learn some general features of racing on the prior tracks that it can maintain in the model throughout the transfer and training process.

Table Of Contents

Preface

The repository used for management of the code throughout the project can be found at the the link: `https://github.com/Reniets/RLTL_Carla`

Aalborg University - December 19, 2019

Jesper Adriaan van Diepen
<jvandi15@student.aau.dk>

Niki Ewald Zakariassen
<nzakar17@student.aau.dk>

Patrick Kampdal Petersen
<pkpe16@student.aau.dk>

Sebastian Lund
<slund17@student.aau.dk>

Simon Steiner
<ssimon17@student.aau.dk>

Glossary

Baseline Agent: An agent trained on a track used to compare results

Source Task Agent (STA): The agent from which knowledge is being transferred. Also known as the director in relation to imitation

Target Task Agent (TTA): The agent to which knowledge is being transferred.

Full Transfer: The transfer of all weights and biases from one agent to another

Selective Transfer: The transfer of weights and biases from selected layers from one agent to another

Imitation Transfer: The indirect transfer of knowledge by asking an already trained agent which action to take when uncertain

Single Track Transfer: When transfer between tracks always use the baseline as the source task agent

Continuous Track Transfer: When transfer between tracks always use the resulting agent from the previous transfer as the source task agent

Polling Rate: A probability value of how likely the TTA is to poll the director for an imitation action

Uncertainty: A value describing for a given state how uncertain the TTA is of how to act

1. Introduction

When humans are to learn complicated tasks they often identify easier sub-tasks and learn those individually instead. Nobody would attempt to drive on the highway before learning the basics of shifting gears. Mostly due to the inherent monetary, and non-monetary, costs of crashing a car but also because the task of driving on the highway with no prior knowledge is very overwhelming. Humans seem to learn well by isolating and improving on specific parts of multifaceted problems, as a way of making them manageable.

Humans carry a lot of previous knowledge into new tasks but computational agents often start *tabula rasa*, as an empty slate with no prior knowledge of the problem domain. The field of *Transfer Learning* seeks to improve learning in similar tasks through transfer of knowledge from a related task that has already been learned [1]. This also encompasses the 'human' approach of learning easier sub-tasks before attempting more complex tasks. Transfer learning has successfully been applied to machine learning tasks involving *natural language processing*, *image processing* and various other domains [2].

Reinforcement Learning is a field that also attempts to mimic how humans learn. With reinforcement learning computational agents learn how to solve problems of maximizing rewards. To this end agents attempt different strategies in dynamic environments, improving them through trial and error [3]. This is akin to how humans learn from their past experiences but also to evolution in general. Evolution could be considered an inefficient optimizer to the reinforcement learning problem of survival in the dynamic environment of Earth.

Part I

Problem

2. Initial Problem

Autonomous driving is an actively researched real world problem relying on machine learning methods. Driving is a complicated task that relies on a lot of previous knowledge that humans may take for granted when first learning to drive. Knowing what a road is, knowing how pressing the throttle will affect the car, and knowing not to drive into walls are simple examples of tasks that, for humans, are already known from previous experiences when we step into the car the first time. One could imagine solving less complicated driving tasks such as simply recognizing where the road is or learning how to drive straight. For a human it would be second nature to use this experience when later driving in complicated road systems.

It would therefore be of interest to combine the method of learning and transferring knowledge into intelligent agents. The task of teaching a reinforcement learning agent to navigate a car is difficult even with large simplifications, as it initially has no knowledge. It has to explore its options and figure out what works in a vast and complex environment. But a human would not throw themselves onto city roads straight away. They would practice on closed areas and rural roads first. This is an approach that could be employed by intelligent agents too, if a way can be found for determining which knowledge to carry over. With that in focus, the goal is not to solve autonomous driving, but rather to explore the potential of transfer learning's application to reinforcement learning agents. This leads to the initial problem:

"Could transfer and reinforcement learning be applied to autonomous driving, to show generalizable transfer of knowledge?"

3. Related Work

This chapter will discuss some of the previous research done on reinforcement learning and transfer learning that relates to this project. The chapter will only briefly cover subjects, and not go into the details of the individual methods. For more details on methods relevant to the project see Part III.

3.1 Background

Reinforcement- and transfer learning are each long studied topics within the field of machine learning. The view of states and actions that is typically employed in machine learning is covered in detail in Chapter **??**.
All relevant information about an environment, and an agent's place therein, constitutes a state. A state contains the necessary information about the environment for the agent to decide on an action to take. The state space is defined as the set of all possible states.
Similar to the state space, an agent also has the action space, which covers all possible actions available to the agent.
When transferring knowledge from one environment to another, there needs to be consistency between the environments, in terms of the utilized state variables and actions. If they are not the same, they at least need to be mapped to logical equivalents. This is task mapping.

3.2 Common Use Cases For Transfer Learning Today

A common use case for transfer learning today is image recognition. The task of detecting general features used for recognition of images, is very similar across different image recognition problems. Image recognition is therefore a great candidate for transfer learning. It is a common approach to use transfer learning as a way to initialize the parameters of an image recognition model [4].

3.3 Transfer Learning And Reinforcement Learning

In 2009 Matthew E. Taylor and Peter Stone wrote the paper: "Transfer Learning for Reinforcement Learning Domains: A Survey". This paper covers multiple research papers, mainly from between 2000-2008, and tries to create an overview of the research between transfer learning and reinforcement learning. The paper[5] will therefore be the main resource for Section 3.3 and 3.3.1.

Reinforcement learning is a popular method for solving complex tasks that have very large state spaces and limited environmental feedback. In recent years, reinforcement learning techniques have achieved notable success solving problems other machine learning methods struggle at.

By 2009, using transfer learning in the context of reinforcement learning was still considered a recent development. The core idea is similar to transfer learning used on images, but instead of learning general features, the algorithms learn a policy, which is a function that maps a state to an action, for a given task.
The hope was that earlier layers of a model learned useful general low level features of problem solving, that could help speed up training for a new different, but related task. A category of transfer problems related to the initial problem of the project was identified in the paper as - **same state variables and actions**.

3.3.1 Same State Variables And Actions

This type of transfer problem expects the source and target tasks to have the same state and action space, and are some of the more simple transfer problems - Since no task mapping is required.

In one of the earlier works on transfer learning with reinforcement it has been shown, that the idea of learning from "easier tasks" can improve training time [6].

One of the drawbacks with learning from easier tasks, is that they require human intervention and human knowledge, to design and define these easier tasks for the agent to learn.

4. Problem Statement

Previous chapters have identified an area of study that has yet to make real world use of knowledge transferring between intelligent agents. The domain of autonomous driving is a realistic and relevant problem domain to examine the potential of agents using knowledge transferring to mimic human learning methods with increasingly difficult tasks.

The goal of such an agent would be to be able to generalize the knowledge from one task to another. The actions and states available would not change as the task will always be to drive the car within the boundaries of the road. The aim of the knowledge transferring would be to outperform another agent, that starts with no prior knowledge, on the same task. Outperforming could entail either getting better faster or ultimately becoming better. This results in the question of:

"Can an autonomous driving reinforcement learning agent be used to improve generalizable performance by transferring knowledge through increasingly difficult tasks?"

Part II

Model Design

5. Markov Decision Process

Reinforcement learning problems build upon the theory of Markov Decision Processes (MDP). An MDP is applicable to decision problems with *discrete time*, which refers to scenarios where all variables are constant between each sampling of the environment, or where that assumption is made.
An MDP can be illustrated as in Figure 5.1 where it is applied to a basic driving scenario. The possible states are being on straight road, being on curved road, or being on the grass and the possible actions in each state are "Throttle" and "Steer". Given the model is in the state "Straight road" and it performs the action "Throttle" there is an 80% probability that it will stay in the same state, while there is a 20% probability of it transitioning to the state "Curved Road". If instead it performs a turn it has a high probability of transitioning to grass which would result a reward of -4. At any given time the probability of transitioning from one state to another remains the same. The probabilities are not affected by the past iteration of states. This property is referred to as the Markov Property [7] which is fundamental to Markov Decision Processes.

This process is a way of representing decision-making problems. The process is defined as a 4-tuple (S, A, P, R) [8].

- S: A set of states
- A: A set of actions
- P: The probability of transitioning from one state to another with an action
- R: The reward obtained for transitioning from state s to state s' due to action a

A regular MDP assumes that the environment is fully observable to the agent so that it knows exactly which state it is in at all times. This is not the case for an environment in which the only means of observation for the agent is through a segmentation camera sensor with a restricted field

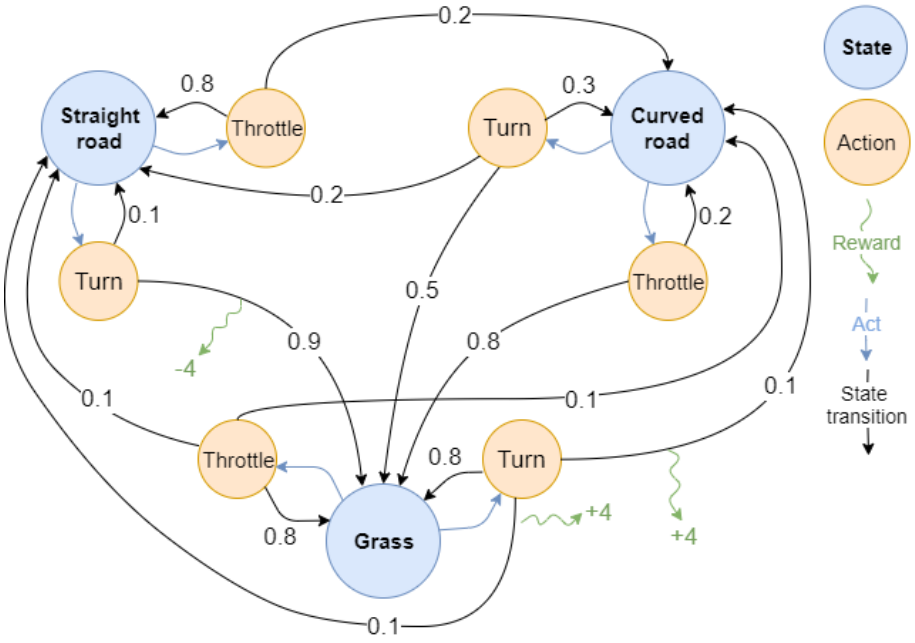


Figure 5.1: Example of an MDP in a simple driving scenario. State transitions with zero probability are not drawn

of view. This makes the environment partially observable to the agent as a full state representation might include things such as an overview of the entire track and lateral and longitudinal acceleration information, hence observation made by the agent is not equivalent to the state of the agent.

The following sections will put each of these parameters into more context in relation to the overall problem.

6. Agent Design

6.1 State Space

The position, interaction and other pieces of relevant information of an agent in an environment is encapsulated by its state, and the set of all possible states is defined as the state space. Due to the partially observable nature of the problem, the state space is not a point of focus, as the agent does not have access to all the information to distinguish all states. The focus is instead on how much of the information about the state is important, and possible, to represent in the observation.

6.2 Observation Space

An observation contains the information about the environment that is available to the agent, which it can use to decide which action to take. For an autonomous car it could be features such as speed, distance to curb and acceleration.

6.2.1 Observation Space Approach

This subsection will define the specific observation space for the agent. The set of observations that the agent can observe is defined by a semantic segmentation sensor, see Section 13.2.1.
As can be seen in Figure 6.1, the road is displayed as purple, the grass is displayed with a green color, and the red colored bar represents the speed of the car. The bar will extend based on the speed, where the leftmost equals zero and the rightmost equals 80 km/h.

The start observation is determined by a given starting point in the environment. The objective is for the agent to achieve the highest possible accumulated reward for traversing the track.

The autonomous car agent has a discrete observation space to navigate because there are a finite number of ways to arrange the pixels of an ob-

Figure 6.1: Semantic segmentation point of view

servation. An observation is a 50x50 pixel image, different observation sizes were tested and a larger resolution only increased training time and not performance. Even though the observation space is discrete the number of possible observations is still very large.

With the observation space covered, it is possible to analyze and design the action space.

6.3 Action Space

Similar to the obsercation space, an agent also has an action space, which covers all possible actions available to the agent. When training the RL agent of this project, it is desired to optimize some policy, which fundamentally is a function that maps observations to actions.

6.3.1 An Action

For simple driving problems, the possible actions might just be any one action from the set of directions {*THROTTLE*, *REVERSE*, *RIGHT*, *LEFT*}. This means that an agent cannot both press the throttle and turn left at the same time, or reverse and go right, but only one action at a time.

A more advanced version might allow the agent multiple actions at a time, such as going forward while turning right, or braking while turning right or even pressing the throttle, brake and turn left, which would probably translate to just going left. Such a model might not define a single action as any one of the possible combinations, but maybe instead as four

boolean variables, one for each option. A single action might be $\{1, 0, 1, 0\}$ which would translate into moving forward and turning right, but not holding down the brake and turning left.

In an even more advanced problem such as controlling a real world car, where an agent has to control multiple things at a time, such as the steering wheel and throttle, we might realize that simply pressing the throttle or not in a boolean fashion might be too rigid. The throttle might instead be modeled as any real value between 0 and 1, and similarly model the steering wheel as any real value between -1 and 1, where -1 is left and 1 is right. A single action could then be $\{0.523, -0.246\}$, which means that the agent wants to press the throttle half way down, while turning slightly to the left.

6.3.2 Action Space

An action space does not have to be finite, as presented in the previous examples.

In the simple driving problem case, the action space was defined as the given set of directions $\{THROTTLE, REVERSE, RIGHT, LEFT\}$, Such an action space is finite, but only 4 different actions are covered by the action space.

The more advanced driving problem's action space covers all possible combination of the 4 boolean variables. Such an action space is also finite, but $2^4 = 16$ different actions are covered by the action space.

Finally a real world car problem's action space covers all possible combination of real numbers between $[-1, 1]$, steering, and $[0, 1]$, throttle. Such an action space is infinite.

Transforming Continuous To Discrete

In the previous example of controlling a real world car, we had a continuous space represented by values in the range of $[0, 1]$ for the throttle and $[-1, 1]$ for the steering wheel. To convert this continuous action space into a discrete action space, it is possible to sample the continuous action space into intervals such as $0, 1$, which may define the discrete space for the throttle: *{0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}*. Using the same method for the steering wheel, the result will be 11 values for the throttle and 21 values for the steering wheel and a total of $11 * 21 = 231$ possible different actions instead of infinitely many, and thereby a discrete action space instead of a continuous.

6.3.3 Action Space Approach

This subsection will define the specific action space available for the agent. The agent has access to three different controls: Throttle, brake and steering wheel. CARLA defines the accepted value interval for instruments as depicted in Equations 6.1 to 6.3, which define a continuous action space. The agent can perform actions with a fixed time spacing of 100ms. The choice of this value is a trade-off between the agent decision frequency and the simulation time, where simulation time in some cases would affect training time.

Various test-runs with different action spaces have shown the discrete action space to be too crude for the agent to learn more than the bare basics. With the continuous action space the agents learned, but slowly, and often failed to obtain consistently good results. A Multi-discrete action space with a sparse selection for each action proved to add enough precision to avoid the problems faced with purely discrete actions, while avoiding the staggering amount of options for the agent to search in the continuous action space.

$$\text{Throttle} : [0, 1] \tag{6.1}$$

$$\text{Brake} : [0, 1] \tag{6.2}$$

$$\text{Steering} : [-1, 1] \tag{6.3}$$

Based on these observations, the agent will use a multi-discrete action space, where we transform each of the instrument intervals into discrete subsets. The agent's action space for each instrument is defined in Equations: 6.4 to 6.6

$$\text{Throttle} : \{0.0, 0.3, 0.6, 1.0\} \tag{6.4}$$

$$\text{Brake} : \{0.0, 0.5, 1.0\} \tag{6.5}$$

$$\text{Steering} : \{-1.0, -0.9, ..., 1.0\} \tag{6.6}$$

By using the discrete subsets for each instrument as possible values for the agent, the final multi-discrete action space is a tuple defined by Equation 6.7 with a total of $4 \cdot 3 \cdot 21 = 252$ different possible actions.

$$\text{Action} : (Throttle, Brake, Steering) \tag{6.7}$$

7. Environment Design

7.1 Environment Modelling

With transfer learning in mind the project group has devised six different tracks, each with different learning goals, that the agent should learn from. The objective is to transfer knowledge from track to track on incrementally harder tracks, hopefully achieving better performances than when no knowledge transfer is done.

The first track is a straight road where the agent has to learn to not drive off the edge. The agent should learn to follow the road. The track is depicted in Figure 7.1a.

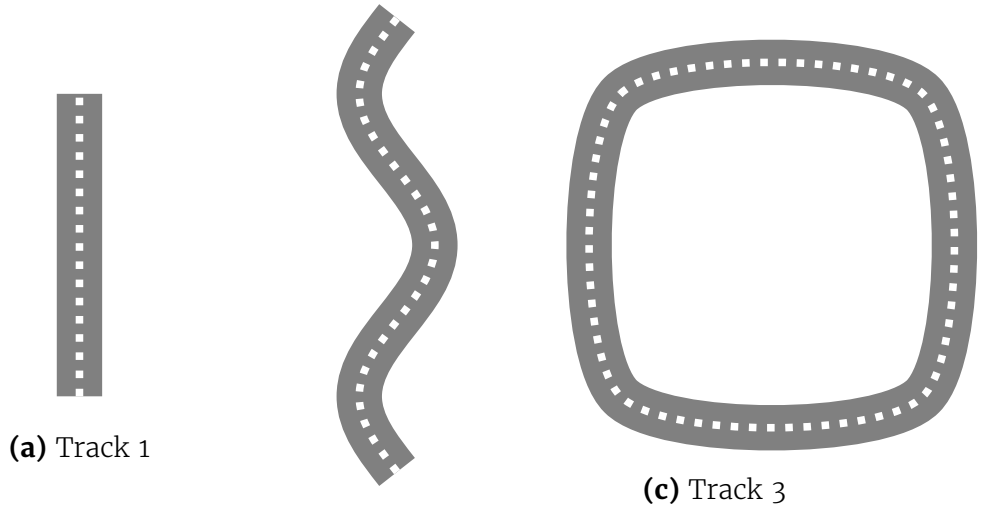
The second track is a sine wave track like the one depicted in Figure 7.1b. This track should teach the agent that both left and right turns exist.

The third track involves sharp turns at the end of a long straight line. The goal is that the agent learns it should still drive fast on long stretches, but should slow down to be able to turn safely at the corners. It should by this track also be able to handle 90 degree turns. The track is depicted in Figure 7.1c.

The fourth track is a more complicated shape: An L-shape. The shape should teach the agent that all straight stretches are not the same length. The track is depicted in Figure 7.2a.

The fifth track has an intersection at its center. The goal is that the agent learns it should drive straight at intersections and not try to cheat. The track is depicted in Figure 7.2b.

Finally the sixth track is a track where the roads are very close together probably confusing the agent. The agent must learn that tracks close together are not connected and to not drive over the small patches of grass between the roads. The track is depicted in Figure 7.2c.



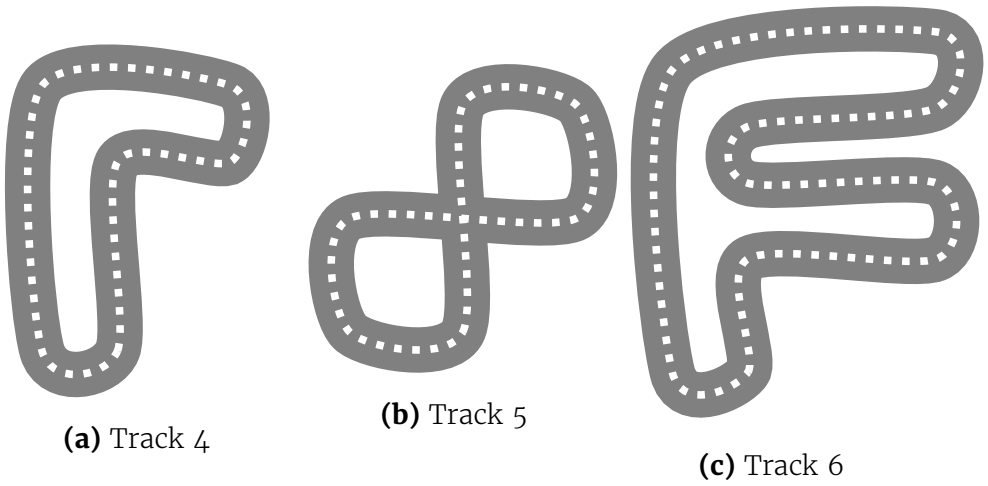
(a) Track 1

(b) Track 2

(c) Track 3

Figure 7.1: The first three tracks that the agent should learn from. It should after training be able to drive straight and turn.

For the purpose of evaluating general performance of agents, there will be slightly altered evaluation variants of the tracks. This approach is uncommon in reinforcement learning, where it often happens that training and testing is performed on the exact same level and task [9]. In this case it is beneficial to detect overfitting and evaluate general performance. Examples of these track variations are track 4 gaining larger, less uniform turns, and track 6 having more varied distances between the roads.



(a) Track 4

(b) Track 5

(c) Track 6

Figure 7.2: The latter three tracks. Contain varied twists and turns. The depicted sizes are not to scale, as track 6 is actually significantly smaller than the other two.

7.2 Reward Function Design

A reward function can allow an agent to evaluate the degree of success of their actions and the resulting states [10]. When designing reward functions, it is important to have a clear vision for what the agent is supposed to achieve. If the design is slightly off, the agent's understanding of the problem will also reflect that inconsistency [11].

It is important to take a look at how a reward function impacts an agent's ability to learn efficiently. For example, a sparse reward function will simply inform the agent when the task was achieved correctly, but would not help the agent get closer to a solution, nor find a better one, unless it randomly stumbles upon a goal state. Thus the agent will be extremely unlikely to learn any task blindly. To deal with this, it can instead be fitted with a shaped function, which can determine how well the agent is performing. Thereby, the reward function is able to reward the agent for any partly successful attempts. Determining what variables should or should not be used to shape this function is a central challenge when designing reward functions [11].

On the other hand, applying a penalty for doing something which is undesirable will teach the agent that the action taken in this state should be avoided. In small scale these penalties can nudge the agent away from behaviour the designer knows to be bad, while a huge penalty will mark that outcome as nearly unacceptable. Overdoing penalties may disincentivize exploring new solutions, or lead to inaction if carrying on is too likely to penalize [11].

7.2.1 Rewards In Practice

In Table 7.1 a set of sub-reward functions are defined and their purpose explained. The full reward function for the agent will be designed as a sum over a subset of these, which all either tries to reduce or induce certain behaviour. The sub reward functions are not an exhaustive list, but the ones identified for the project.

7.2.2 The Naive Reward Function

The first reward function used a combination of the **Stay On Road**, **Drive Fast** and **Avoid Grass** sub-reward functions. By using these the agent is rewarded for each tire on the road and for its speed, and penalized for each tire on the grass. The idea was that it should avoid the grass and thereby stay on the road, but we also wanted the agent to drive fast on

Name	Purpose	Type
Drive Far On Road	Points pr. meters driven pr. wheel on the road	Reward
Drive Fast	Points pr. km/h	Reward
Stay On Road	Points pr. tire on road	Reward
Return To Road	Points for each wheel that transitions to or from the road	Reward/ Penalty
Avoid Grass	Points pr. tire on grass	Penalty
Turn Sensitivity	Points pr. degree of rotation	Penalty
Drive Short On Grass	Points pr. meter driven pr. wheel on grass	Penalty
Do not Stand Still	Points pr. step below a triviality limit of velocity	Penalty
Follow spline	points pr. meter driven following spline	Reward

Table 7.1: Sub-reward functions, their purpose and reward/penalty type

the track, thereby rewarding it for speed.

Agent Training

Throughout the training phase the agent displayed different approaches to this problem. While learning basic controls the agent snailed its way forward but usually ended up in the grass where it got stuck and could not return to the road - This approach gave a large penalty.

The next approach after ending up in the grass was to simply drive straight on an otherwise curved road, to increase its speed and thereby minimize the total penalty of driving on the grass - This approach gave less total penalty.

The final approach that the agent settled for, before it was stopped training, was to simply do nothing and stay still. The risk of driving on to the grass was too great according to its policy, and the passive reward for simply staying on the road was satisfactory - This approach gave only **Stay On Road** rewards.

The Takeaways

The takeaways from this experiment is that the agent can grow content with simple passive rewards, if the risk of penalties is too large. Furthermore a distinction between on/off road for rewards was needed, because simply giving rewards for driving fast is not a true representation of the

problem we want to model, we want the agent to drive fast *on the road*, and not just anywhere.

Finally a key takeaway is that the agent really did improve its performance on the problem defined by the reward function - Just not how we expected it, which is a sign that the problem we want the agent to solve, is not the same problem described by the reward function given to the agent.

7.2.3 Final Reward Function Design

In order to make the agent drive along an arbitrary road it is important to reward the agent when making progress along the road. Therefore the agent is rewarded or penalized for each action.

The only reward given to the agent is the sub-reward function **Follow spline**. Where a spline is a line following the center of the road which is invisible to the agent. The reward is dependant on the distance traveled along the spline. This calculation is formalized in Equation 7.1:

$$DistanceMovedAlongSpline * c_1 \tag{7.1}$$

To ensure the agent would not skip entire parts of the track in order to maximize spline distance travelled, it will not be rewarded if the spline travelling speed exceeds 300 kilometers per hour, since such speeds can only be obtained by cutting parts of the track. The speed is not limited to the top speed of the car, since the fastest legal route around a track would deviate from the spline at the center of the road.

In addition to the rewards, the agent is also penalized by the **Avoid Grass** sub-reward function, which gives a penalty for driving off the road with a penalty at each time step based on the number of wheels driving in grass. The way this is calculated appears from Equation 7.2:

$$-(WheelsOnGrass) * c_2 \tag{7.2}$$

To avoid the agent swaying more than necessary on the road the sub-reward function **Turn Sensitivity** is introduced to give a penalty for each degree the car turns since last time step. The degree is calculated from the environment as the difference in the car's bearing between two ticks. This is formalized in Equation 7.3:

$$-(|(|PrevAngle| - |CurAngle|)| * WheelsOnRoad) * c_3 \tag{7.3}$$

Lastly, to avoid the agent from standing still for any reason, we introduce the sub-reward function **Do not Stand Still**, which is formalized in Equation 7.4:

$$\begin{cases} -c_4, & \text{if } velocity < 0.5 \\ 0, & \text{otherwise} \end{cases} \tag{7.4}$$

The choice of constants for each of these rewards is based on trial and error testing. The values that were used were: $c_1 = 4, c_2 = 2, c_3 = 0.15, c_4 = 4$.

The design choices for this reward function should result in the agent trying to follow the road for as long as possible without unnecessary swirling. If the agent interacts with something else than the road, it will either take actions to avoid going off-road or come to a sudden stop in order to minimize overall penalty of driving on grass.

8. Overview Of Design Decisions

This section will serve as a summary of the design decisions made throughout this part.

It was decided that the observations should be represented by semantically segmented image data from the point of view of the car. Different resolutions of the image were experimented with resulting in a resolution of 50x50 pixels.

In Section 6.3.3 multiple options for defining the action space were tried, resulting in the choice of a multi-discrete action space.
The different environments that the agent should train on were decided to be 6 distinct tracks that should each teach the agent something different. Based on these tracks an appropriate reward function was devised, attempting to give continuous feedback by rewarding driving along the road and punishing driving off-road. For this purpose the agent should be rewarded for following the road, and penalized for wheels being off the road, standing still, and turning more than necessary.

Now that the problem has been modelled it is possible to explore how to solve the problem with transfer and reinforcement learning in the Learning Methods part.

Part III

Learning Methods

9. Transfer Learning

This chapter will give an overview of the general theory of transfer learning and introduce some evaluation metrics and simple approaches to transfer.

9.1 Overview

In Transfer Learning there is a set of general measurement options which are listed below and visualized in Figure 9.1 [5].

- Jump start: Performance of target task increased without any training
- Quicker performance improvement: Improves more in less time
- Higher asymptotic performance: Target task reaches an overall higher performance than the source task ever does

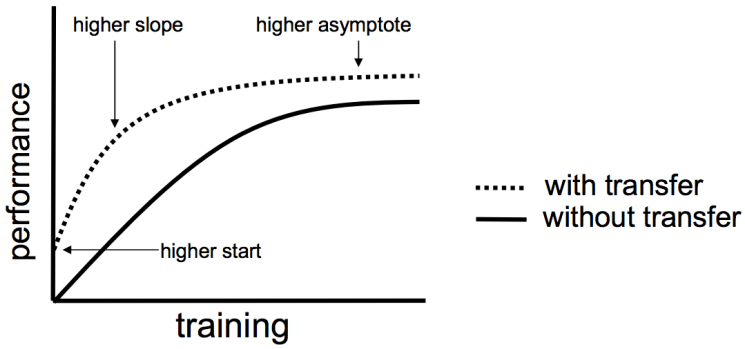


Figure 9.1: Transfer learning improvement measurements [1]

The difficulties of Transfer Learning can be categorized into three general questions [12]:

1. What to transfer?

2. How to transfer?
3. When to transfer?

Asking "**What** to transfer?" examines which parts of knowledge should be transferred between tasks. This question deals with identifying which pieces of knowledge are generalizable and applicable to a related task.

After having identified this knowledge, a method for transferring it into another agent needs to be developed in order to determine **how** to correctly transfer.

Lastly, is the decision of **when** the target agent should receive the knowledge. Some pieces of information might be used to enhance the jump start measurement by initializing the target agent with some knowledge, whereas others might enhance the overall training speed and performance by being fed information during training. Another question to ask regarding "When" is whether or not knowledge should be transferred at all. In some cases it may cause "Negative Transfer" which is the concept of transferred information hurting the performance in the target task. [12]

9.2 Transfer Learning For Reinforcement Learning Domains

The previous sections covered transfer learning on a theoretical level. This section is concerned with applying transfer learning to reinforcement learning agents and discussing the specific approaches that could be applied.

An RL agent can observe the current *state* of the environment and perform *actions* that change the state and prompt *rewards*. The goal of the RL agent is to learn some *policy* for gaining the maximum possible reward. At a simple level this is what makes up an RL agent. Three basic method types can be used: starting point methods, Imitation methods, and Hierarchical methods [1].

9.2.1 Starting Point Methods

The starting point methods make use of the fact that when RL agents are initialized, they start from zero and pick actions randomly. To allow an agent to start much closer to a good solution the agent is initialized with knowledge from the source task agent. This would primarily result in a

jump start, as was described in Section 9.1. The primary downsides of this approach is that bad habits will also be transferred.

9.2.2 Imitation Methods

An imitation method tries to further decrease the training time of the target-task agent. In contrast to the starting point methods, imitation attempts to incrementally help the agent during its training, instead of initializing the target-task agent with source-task knowledge. One way of doing this is to use the policy of the source-task agent to make decisions guiding the target-task agent at some chosen time steps. Intuitively this should result in a quicker performance improvement and might have some benefits over using starting point methods, if identifying the specific situations in which the guidance of the source-task agent could be beneficial is possible.

10. Proximal Policy Optimization

To show transfer of knowledge, a model needs some meaningful knowledge of the problem that can be transferred. The problem of driving a car in a realistic environment is a difficult one, and to obtain a model as quickly as possible, a state-of-the-art method was chosen. Proximal Policy Optimization (PPO) was developed by OpenAI as a simple alternative to other modern methods which are difficult to implement and tune, and has been proved effective in complex environments [13].

PPO uses concepts from multiple different methods, but is fundamentally a policy gradient method that attempts to combine the ability to train an agent from multiple actors while trying to ensure steady improvement of the policy by using a clipping-function as explained in Section 10.3.1. The following subsections will shortly introduce background of each of these methods.

10.1 Policy Gradient

A policy is a probability distribution of actions, for a given state. The goal of the policy gradient is to increase the probability of taking a given action in a given state, whenever that produces a better result than previously predicted. This is done by changing the policy parameters θ, which is often the weights and biases of neural networks, by using a stochastic gradient ascent method. The gradient ascent tries to optimize the long-term cumulative reward [14].

10.2 Actor-Critic

This approach combines two overall approaches to reinforcement learning: Policy-based methods and Value-based methods. This is done by using two interacting components; the *actor* and *critic*. The actor decides how the agent behaves in a given state by looking at its policy, also known as the policy-based part. The critic estimates how good states are based on the actions taken and their associated reward, which is known as the

value-based part. To allow the critic to evaluate the actor it uses an advantage function. This describes the difference between the value that the actor received by taking an action in a given state, and the value that was expected from the state [14].

10.3 Objective

The previous sections have introduced how policy gradient methods can be used to optimize policy parameters, how actor-critic combines value-based and policy-based methods, and how the clipping-function helps ensure a safe policy improvement. These make up the parts of the PPO method. The method assumes stochastic policies, i.e. a policy that outputs a probability distribution over actions. The PPO method updates policies with Equation 10.1:

$$\theta' = \arg\max_{\theta} L^{CLIP}(\theta) \tag{10.1}$$

A specific policy is described by its policy parameters θ. We seek to update these policy parameters, arriving at θ' which are the parameters that best maximize the objective. The objective L^{CLIP} is given by Equation 10.2: [13]

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t\left[\min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\hat{A}_t,\ \text{clip}\Big(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}, 1-\epsilon, 1+\epsilon\Big)\hat{A}_t\right)\right] \tag{10.2}$$

The term, which L^{CLIP} takes the expectation over, describes how well the potential new policy parameter θ compares to the current policy parameters θ_{old} on action a_t in state s_t. The objective takes the minimum of two terms. The first term is the ratio between the probability of taking action a_t under the potential new policy π_θ, and the probability of taking action a_t under the current policy $\pi_{\theta_{old}}$. This probability ratio is multiplied by the advantage at timestep t: $\hat{A}_t$ which is a measure of how much action a_t was a good or bad decision given the value of state s_t, in terms of expected reward.

10.3.1 Clipping Function

The objective function being used in Equation 10.2 needs some way of restricting how much the policy can change each update. Previous methods involved using the Kullback-Leibler divergence to look at how different the new policy is from the old and restricting how large this difference can be. PPO proposes the use of clipping to restrict this difference as is illustrated in Figure 10.1

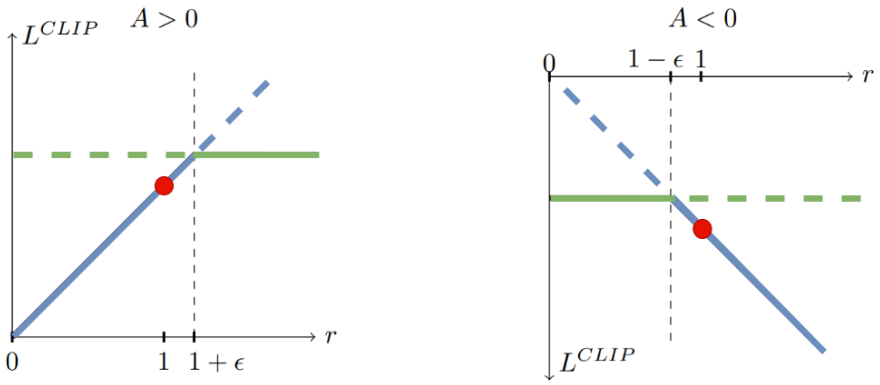


Figure 10.1: Depiction of clipping function [13]. Left plot showing clipping of the objective function when advantage is positive and right plot when advantage is negative. The blue lines refer to the left part of the minimum e.g. the unclipped part. The green lines refer to the right part of the minimum e.g. the clipped part.

r is the ratio between the old and the new policy, so when the advantage is positive we want this ratio to increase as this makes the given action more likely in the new policy in comparison to the old policy, and conversely when the advantage is negative we want the action to be less likely hence a lower *r*.

10.3.2 Advantage

PPO learns while online, which means that the agent learns while doing and does not remember its own past experiences. The paper showcases a method where the policy runs for T timesteps and the rewards from those timesteps are used to estimate the advantage for the update. Equation 10.3 shows their computation of the discounted advantage $\hat{A}_t$:

$$\hat{A}_t = -V(s_t) + r_t + \gamma^1 r_{t+1} + ... + \gamma^{T-t+1} r_{T-1} + \gamma^{T-t} V(s_T) \tag{10.3}$$

Where $V(s)$ is, usually an estimation of, the discounted expected reward from being in state s, independent of the action chosen. γ is the discount factor which is a number in the range $[0, 1]$ for how much future rewards should be discounted. r_t is the reward received at timestep t. The advantage gives a measure for how good an action was, i.e. was the reward from applying this action better or worse than what was expected from this state?

The second term that is minimized in Equation 10.2 also consist of a probability ratio between the potential new policy and the current one, as well as the advantage. The probability ratio is clipped between $1-\epsilon$ or $1+\epsilon$, this is done to ensure that the policy updates being made are reasonable, in

how much the proposed policy can deviate from the current policy [15].

The full objective additionally contains terms for the value function squared error loss L^{VF} used for estimation of the critic $V(s)$. There is also an entropy bonus $S[\pi_\theta]$ to incentivize exploration in the actor [13].

10.4 Component Overview

The particular implementation of PPO used by the group uses a combination of different types of neural networks to convert observations from the car's sensor to an optimal action. It accomplishes this with a number of CNN layers followed by a fully connected layer. A simplified overview of the component diagram is illustrated in Figure 10.2.

An observation from each car consists of a 50x50 array with 3 channels to account for rgb. Because this is an image, a convolutional neural network (CNN) is used because they are good at extracting features from images[16]. Each convolutional layer consists of weights and biases which are updated during training. Each of these weight and bias layers make up the knowledge of the agent which consequently can be a way of transferring knowledge. Multiple layers of convolution are used to extract multiple levels of details from the observation, which will also be further discussed in the following chapter. The fully connected layer after the CNN layers is used to learn combinations of the high-level features from the output of the CNN layers.

The policy network and value network are fully connected layers each with a specific purpose. The policy network tries to convert the features of the observation into a suitable action. The value network tries to assign some value to the current state that the agent is in, which is useful for calculating the advantage. The details of each of these types of networks will be further described in the following chapter, motivating how transferring weights from one type of layer might help the transfer agent improve.

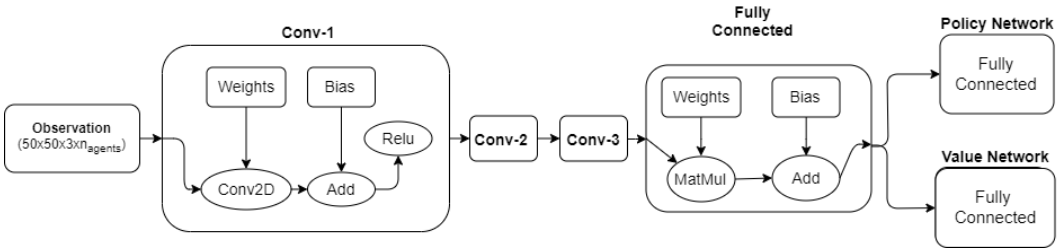


Figure 10.2: Illustration of PPO component structure

11. Artificial Neural Networks

This chapter will go into different artificial neural network topics such as fully connected, convolutional and different optimizers used for training the networks.

11.1 Fully Connected Neural Network

A fully connected neural network is a specific neural network architecture that can be used as a 'Universal Approximator' which can learn any function [17]. It is structured as a series of fully connected layers, where each layer consists of a number of neurons and each neuron is connected to all neurons in the following layer. In Figure 11.1 the left illustration shows such a network, and the right illustration looks at how each node deals with the inputs given to it.

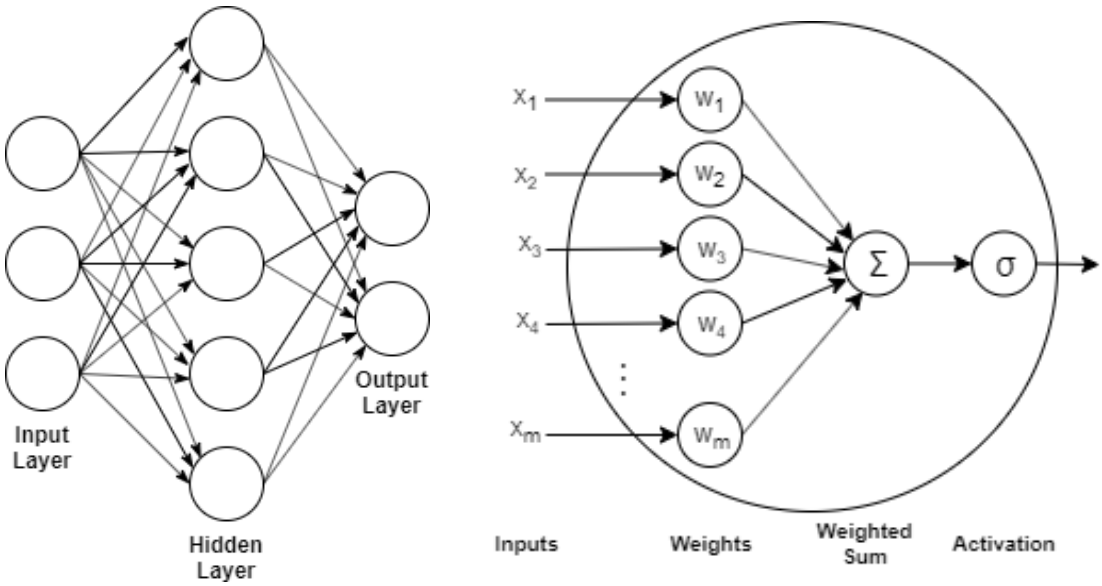


Figure 11.1: Illustration of a fully connected ANN (Left) and a single perception (Right)

A network such as the one illustrated works by receiving inputs, applying

the respective weights, summing these, and lastly activating the result. By doing so it converts a number of inputs into a single number that can be applied to other neurons. The way such a network 'learns' is by changing the weights such that the number resulting from the inputs is closer to the result that was expected.

Mathematically a single feed-forward operation is quite simple as displayed in Equation 11.1, where y is the output of the neuron, w_m the m'th weight, and x_m the m'th input:

$$y = \sigma(w_1 * x_1 + ... + w_m * x_m) \tag{11.1}$$

σ refers to an activation of the weighted sum, which is done to figure out how the result of this neuron should be fired. Since results potentially range from $-\infty$ to ∞ an activation of this is often useful e.g. the Sigmoid activation function which squeezes the value into the range (0,1). The squeezing is done to allow the weights to be finely tuned.

To recognize more complex patterns such as images a convolutional neural network can be used as will be described in the following section.

11.2 Convolutional Neural Networks

When working with images, or other data with spatial relationships, it might be tempting to flatten the data into a single vector and feed that to a fully connected layer. In practice fully connected layers have a poor performance on data such as images [18], and the reason why will be examined in this section.
As explained in the previous section, a fully connected network has a weight between each neuron in the previous layer to the current layer, and as each weight is a parameter, the amount of parameters might quickly become too large for spatial data such as images [18].

An input RGB image with a mere 100x100 pixels, will flattened be a vector with a length of 30.000 values. And if the first hidden layer only has one percent as many neurons (300), then there will be a total of 9.000.000 parameters, just for the first layer alone, and we should expect many more layers.
Scale that up to a high resolution image 4K (3840x2160), and the parameter count quickly become unfeasible to train.

How Convolutional Neural Networks Can Help

Instead of having one individual weight for every pixel in each channel, a convolutional neural network utilizes a combination of weight reuse and parameter minimization to avoid the problem of an exploding parameter count. [18]

11.2.1 Kernels

Convolutional neural networks reuse weights by using kernels instead of a single fully connected ordinary weight matrix. A kernel is a set of weights that functions as a filter which is applied to an image multiple times on individual sub-parts of the image. By applying the same filter multiple times, we reuse those same weights for the whole image. [18]

A kernel is a feature detector that when trained can be used to detect certain features in a sub-part of an image, such as horizontal lines. By building ever more complex features of lesser complex features, the convolutional neural network can learn to detect complex structures [18], such as which way a road is curved.

A kernel with size *n* will create a kernel matrix of size $n*n$ (Only one color channel). Such a kernel is moved pixel by pixel throughout the image and applied to that sub-part as a unit-wise matrix multiplication, and then as a sum over the result matrix [18]. An example of how the kernel is applied to an image can be seen in Figure 11.2 where the purple part of the image matrix, is the sub-part the kernel is currently being applied to. The operation is formalized in Equation 11.2, where:

- O: Output feature map
- i: Input image
- k: Kernel
- x and y: The dimensions of the kernel

$$O[m,n] = (i * k)[m,n] = \sum_x \sum_y k[x,y]i[m+x,n+y] \tag{11.2}$$

The output matrix on Figure 11.2 is called a feature map [20], and describes how prominent a feature is, in a specific sub-part of the image. An example of a kernel applied to a segmentation sensor image can be seen in Figure 11.3.

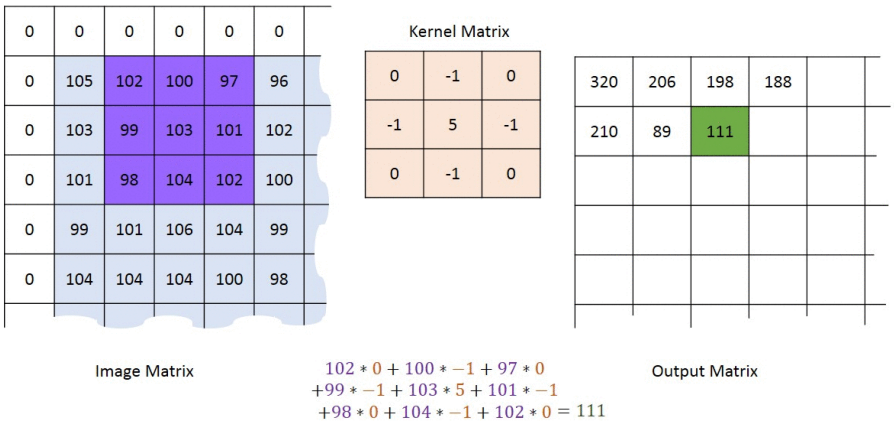


Figure 11.2: Kernel multiplication example [19]

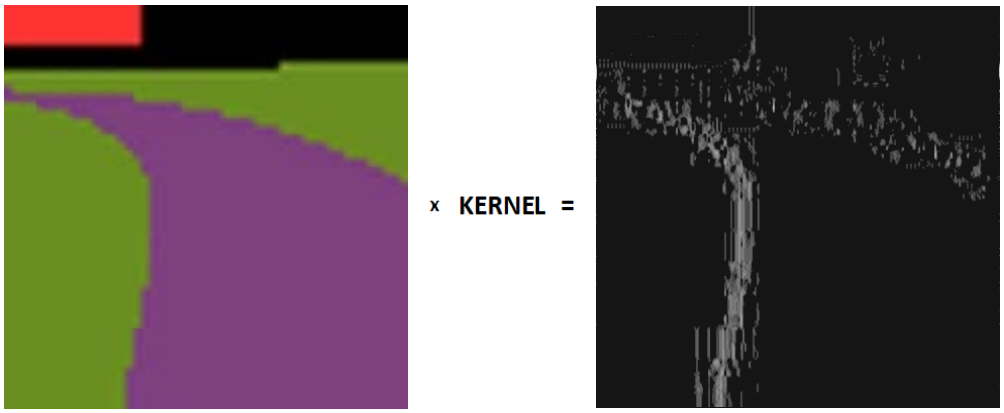


Figure 11.3: Illustration of a convolution input (left), and an output feature map (right)

Pooling

Pooling layers reduce the parameter count by reducing the size of feature maps. A pooling layer is placed after a convolutional layer, and its input is therefore the feature map. The output is a shrunk feature map, that only keeps the most relevant information from the original feature map. By reducing the feature map size, there will be less data for the next convolutional layer to extract features from, thereby minimizing the parameter count [18].

Kernel Stride

The kernel stride defines how often the kernel is applied to an image, as an offset between each sub-part. In Figure 11.2 the purple area with a stride value of one, will move one pixel to the right, and one pixel down

once it wraps around. If the stride value is increased to n, the purple part will move with an offset of n instead of 1. By using a larger stride, there will be less kernel overlap, and the output feature map will be reduced in size [18].

While pooling is often used, it has been found that a convolutional layer with an increased stride can be used instead of pooling layers without loss of performance [21], which is the case in the used implementation of ppo which is also visualized in Figure 10.2 from the component overview.

11.3 Optimizers

An optimizer is the method used to train our model, by minimizing or maximizing an objective function, also known as the error or loss function. The loss of a machine learning model, is simply a mathematical function which is dependent on the model's internal trainable parameters, such as weights and biases. By using an optimizer to minimize the loss, we iteratively improve the trainable parameters of the model. [22].

11.3.1 Gradient Descent

The most commonly used optimizer is the gradient descent method and its variants. With gradient descent the update step is calculated as the negated partial derivative of the loss function, with respect to the individual trainable parameters, multiplied by a constant called the learning rate. This can be seen formalized in Equation 11.3 where θ_t is the model's current parameters, θ_{t+1} the updated model parameters, η the learning rate and $\Delta J(\theta)$ is the gradient of the loss function $J(\theta)$. [22]

$$\theta_{t+1} = \theta_t - \eta * \Delta J(\theta) \tag{11.3}$$

Stochastic Gradient Descent

For stochastic gradient descent (SGD) one update is calculated for each training example. This creates a model with a high update frequency, which causes the updates to have a high variance which causes the loss function to fluctuate. The fluctuation is caused by individual data points suggesting very different directions for the update. While this might sound problematic, it is often the reason why SGD outperforms methods such as the batch gradient descent, since the high variance helps us discover new and possibly better local minima. [22]

But too high a variance can be problematic, since it ultimately complicates the convergence to the final minimum. High variance updates might push the model slightly off course and then back again thereby making the model circle around the minimum. [22]

Batch Gradient Descent

Another gradient descent variant, is batch gradient descent (BGD). In BGD we calculate the loss gradient for the whole dataset before doing a single large parameter update. One of the problems with BGD is that some datasets might be too large to fit in memory, and that a smaller subset of the whole dataset might give the same update direction. This means that redundant gradients are calculated. Finally BGD is only guaranteed to converge to a local minimum for non-convex problems. [22]

Mini-Batch Gradient Descent

The mini batch method is a midway solution between SGD and BGD. It helps alleviate the high variance problems with SGD, without regaining the problems of BGD. In mini batch we decrease the amount of samples for one training update from the whole dataset in BGD to n training samples, resulting in a batch size of n where $1 < n < |Data|$. Mini batch gradient descent reduces the variance, and can utilize optimized matrix operations for increased efficiency over SGD. [22]

11.3.2 Momentum

Because of the high variance in SGD or low batch sizes, a technique called momentum was created. Momentum helps increase training in the relevant directions and reduce the impact of oscillation in irrelevant directions. This is done by adding a momentum factor to the previous Equation 11.3, which can be seen in Equation 11.4, where γ is the momentum term which is usually set to 0.9 or similar. [22]

$$m_t = \gamma m_{t-1} + \eta * \Delta J(\theta) \tag{11.4}$$

The update is now calculated as in Equation 11.5. The property of momentum is inspired by classic physics, as long as a ball keeps rolling downhill it will gain momentum and its velocity keeps increasing. Using momentum helps us take larger updates in the direction most of the data agrees on is correct, while reducing the impact of any oscillation. [22]

$$\theta_{t+1} = \theta_t - m_t \tag{11.5}$$

Momentum can help decrease training time, but it has certain drawbacks. As with classic physics, when a ball is rolling downhill once it reaches the

bottom it will not magically lose all its momentum, but instead begin to roll uphill on the other side before returning, and at some point settles at the bottom, this also happens while training models. [22]

11.3.3 ADAM

One of the latest widely used optimizers is the ADAM optimizer. The ADAM optimizer is an adaptive learning method, which means that it computes individual learning rates for each parameter. ADAM utilizes two methods for calculating the adaptive learning rates for each parameter.

The full ADAM update step can be seen at Equation 11.6 where $\hat{m}_t$ and $\hat{v}_t$ are estimators for first(mean) and second(variance) moment of the gradients and ϵ a small constant used to avoid division by zero.

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t \tag{11.6}$$

The most relevant part of Equation 11.6 is $\frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t$ which can be separated into two parts for easier understanding.

The first part can be seen at Equation 11.7 whose job conceptually is to scale the learning rate for each parameter, depending on a moving average of previous calculated gradients. The effect of this scaling is that parameters which usually have a large gradient will have a lower learning rate, while parameters which usually have a lower gradient will have a higher learning rate.

$$\frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} : Adaptive\ learning\ rate \tag{11.7}$$

$$\hat{m}_t : Momentum \tag{11.8}$$

The second part, which can be seen in Equation 11.8 is a term similar to momentum described in Section 11.3.2 and calculated similarly, except that $\hat{m}_t$ is calculated as a moving average of the gradients and then multiplied with the adaptive learning rate of Equation 11.7. The conceptual job of the momentum is to maintain step direction, so high variance updates do not affect the training too much.

By multiplying Equation 11.7 and Equation 11.8 we get an adaptive learning rate for each parameter, which automatically adjusts the learning rate depending on the size of previous learning rates and the momentum. By

using ADAM one would often see an improved training time, since it rectifies all of the previously mentioned problems.

For some problems, such as image classification ADAM does usually not converge to an optimal solution, which means that typical learning with SGD and Momentum is still widely used for those kinds of problems.

ADAM was the optimizer used in the project, since it was the one chosen by the Stable Baselines library PPO2 implementation used to train our agents.

12. Knowledge Transfer Approaches

This chapter will cover the different approaches to knowledge transfer between agents that will be tested in this project. Below on Figure 12.1 a simplified component overview of the agents' learning model is depicted. The overview is meant to showcase why it could be beneficial to transfer concepts between agents using the full and selective approach described in the following sections.
An alternate approach 'Imitation in Times of Uncertainty' will also be explained. The attempt instead transfers rules which an uncertain agent can follow and learn from instead of receiving concepts directly.

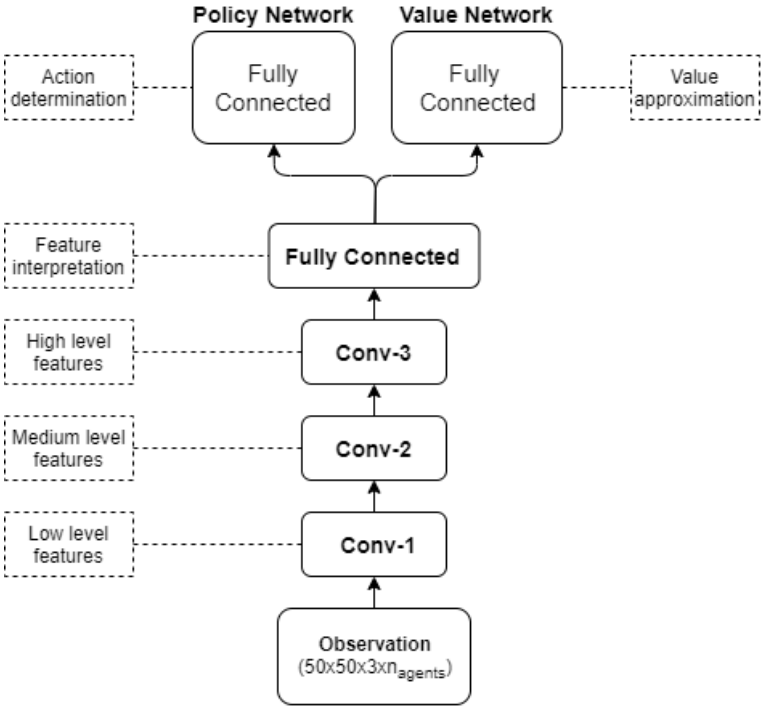


Figure 12.1: Simplified component overview, with assigned conceptual component objectives.

12.1 The Full Approach

The most straightforward way of transferring knowledge from one agent to another is the starting point approach described in Section 9.2.1, where the weights and biases of the old agent are copied over to the new one. Since these make up the entirety of what we call an agent's knowledge, if they are all carried over, we end up with a clone of the old agent as can be seen in Figure 12.2.

The idea with this approach is to assume the new task is not too different from the old, and leaving it up to the new agent's training to fix wherever that assumption does not hold.

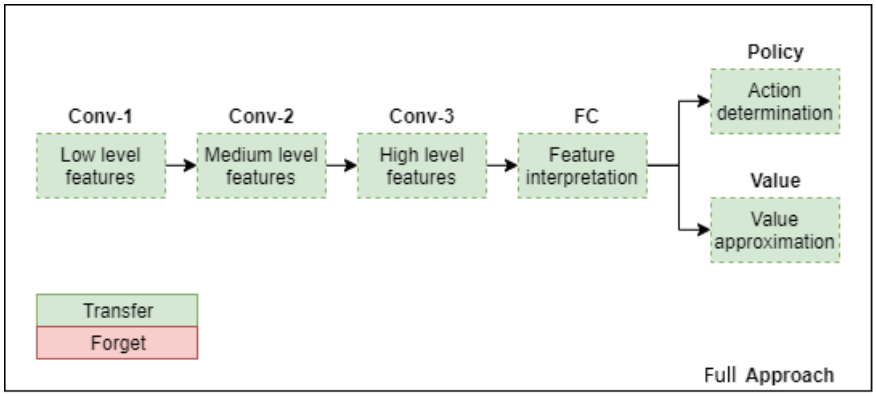


Figure 12.2: Full transferal of the network.

12.2 The Selective Approach

Since not all knowledge from a previous agent will have a positive influence on a new agent, some discretion is warranted as to what knowledge to transfer over to the new agent's starting point. Instead of transferring over the whole agent, specific layers can be picked out to transplant each time, while leaving the rest to train from scratch for each new agent.

Each layer represents some operations, so the question for each layer is whether that operation is generalizable between the source and target task. Figure 12.3 contains an example selection where the convolutional layers for extracting features were deemed generally applicable, but interpretation of those features was deemed too task-specific. So only the former are chosen to be transferred.

Once the chosen layers have been transferred, the new agent needs to train its other layers. Since the transferred layers are those that were already deemed applicable to the task, they may not need any further training. If that is judged to be the case, their values can be frozen during all or most of the training, to allow the training to be focused on getting the task specific layers right.

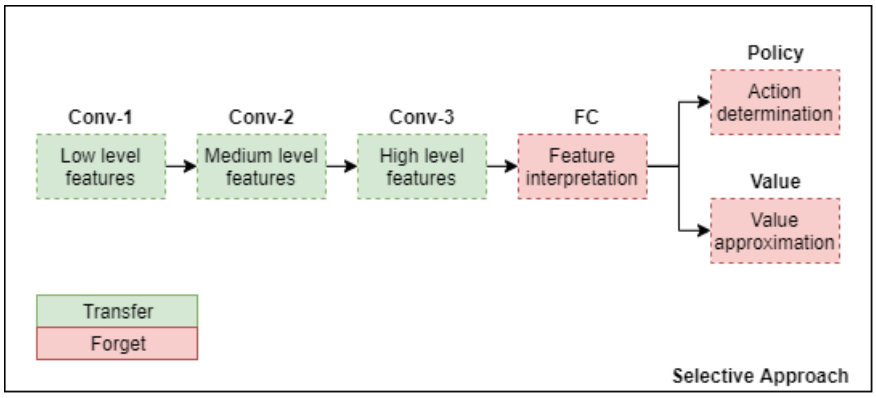


Figure 12.3: An example selection, where just the convolution layers are deemed transferable.

12.3 Imitation In Times Of Uncertainty

Imitation aims to improve the new agent more over time instead of all at the start, as described in Section 9.2.2. Instead of transferring a selected amount of knowledge right from the beginning of training the target task agent (TTA), as in the starting-point approaches, imitation only guides the TTA in states where it is uncertain of what to do. This uncertainty could be calculated by the number of times it has seen a similar state as opposed to the state it has seen the most, which would reflect how certain it is in a state, compared to the one it should be most certain in. At first a high uncertainty would be the case for all states, but should decrease over time as the TTA acquires experience. If the complexity of the task were to increase, the idea is that the source task agent (STA), having not seen the extension of the state space for the more complex task, would still be able to provide a better action than just doing a random action. In theory the most similar state could be used as guidance

One tested variant, for Q-learning, of guiding the TTA is to extract and analyze state-action pairs based on high-level features from the STA. The analysis is used for generating a policy, consisting of a symbolic representation of the best state-action pairs learned by the STA [23]. If the

source and TTAs use a cost function to calculate which action to take in a given state, the source agent would provide the best action according to the cost function, every time the TTA encounters an unseen state.

For the state space described in Section 6.2.1, there are 4^{50*50} possible states, according to an image at 50 * 50 pixels, each with 4possible values, given an environment equal to the one seen in Figure 6.1. To find the difference between two states could be checking whether or not, there is a difference in value for each pixel. As the color code of each pixel does not correlate to the impact a change would make, as a road and a wall could be two shades of grey, the Euclidean distance would not retain this. A simple measurement of difference such as the Hamming distance[24], would capture big or small changes in value, in an equal manner.

Part IV

Experimental Setup

13. Simulation

In this chapter a simulation tool for the project will be chosen. The following sections will analyse a simple simulation tool and a more realistic one.

13.1 CarRacing

As a simple driving simulator environment, a brief overview will be made of the OpenAI gym environment: CarRacing [25]. The environment has a top-down point of view, with sensors specifying speed, steering wheel position, and gyroscope data. An image from the view of the agent is shown in Sub-figure 13.1b.

The most significant simplification of the simulation is the point-of-view which greatly improves the perspective of the agent gaining a large amount of information, and is infeasible if one wanted to scale the model to a real world scenario.

13.2 CARLA

A more realistic simulation tool that fits the project requirements and is also publicly available for free is CARLA. Its main goals are to support development, training, and validation of autonomous driving systems, a screenshot from within the simulation can be seen in Figure 13.1a. The simulation platform supports flexible specification of sensor suites, environmental conditions, full control of all static and dynamic actors and track generation [26].

13.2.1 Sensors

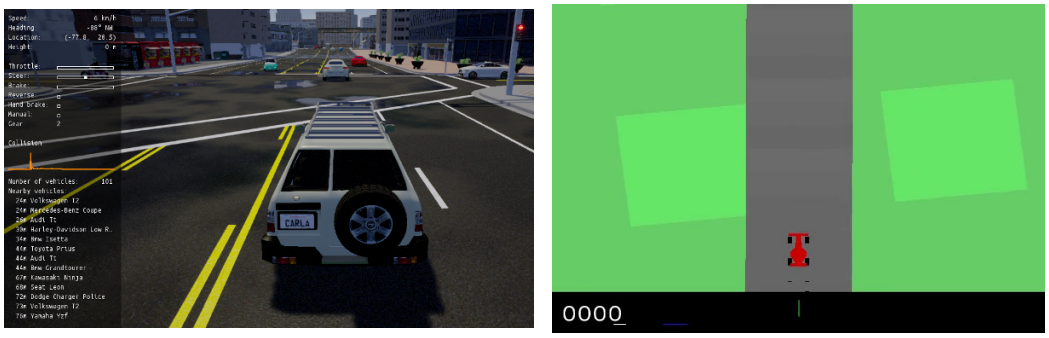
Along the fact that it is designed for machine intelligence solutions, it has the option to add fully customizable specifications of which sensors are being used. With these sensors, it will be possible to show the perspective

of the cars, but also determine distance, collision, lane invasion among other features. The sensors are as follows:

- The depth sensor
- The semantic segmentation sensor
- The global positioning system (GPS)
- A collision sensor

The last two sensors will not provide information to the autonomous car, but rather the system in order to evaluate the degree of correctness in the cars traversal of the map.

The sensors will provide information so that the car has some point of reference to adjust in order to learn how to act.



(a) Image from Carla simulation

(b) Image from CarRacing simulation

Figure 13.1: Comparison of realism and simplicity in simulation tools

13.3 Realistic vs. Simple environment

As stated in Chapter 2, the main focus of this project, is not to solve autonomous driving but instead to explore the potential of transfer learning in relation to reinforcement learning, in the context of autonomous driving. This might beg the question, why choose a realistic simulation instead of a simple simulation, if it is the transfer and reinforcement learning aspects that are important?

While it is true that transfer is the most important aspect, it is also important to have a sense of the applicability to real world tasks. A very simple environment, might increase the likelihood of meaningful transfer, but it does not reflect any real life implications since there is quite a

difference between a 2D racer and a real world car.

While CARLA is not a perfectly realistic simulation, it is close enough to satisfy the proposition that if transfer learning is able to increase overall performance or decrease training time, it might transfer to real world scenarios. For these reasons, we have chosen a realistic simulation environment instead of a simple one such as CarRacing, which is very distant from the real world, and thereby does not tell us anything about the applicability for real world problems.

With simulation covered generally with a broad perspective, and the decision of which simulation tool to use in this project, it is possible to start modelling the problem in the next chapter.

14. Evaluation Plan

Before implementing transfer learning for the problem domain of autonomous driving, we need to define an evaluation plan. The evaluation consists of different agents which allow for comparisons and increasingly difficult tasks to show transfer.

14.1 Agents

To evaluate the transfer of knowledge, two different types of agents are used: A transfer agent and a baseline agent. The transfer agent is an agent which carries experiences from other tracks. It will be trained for a fixed amount of time on each track, generally for shorter than the baseline agent, it might have an advantage on each new track, as it carries previous experiences. The baseline agents are new agents for every track, each trained from scratch on their own respective tracks. The purpose of the baseline agents is to get a baseline reading for how an agent, with the same parameters as the transfer agent, would learn the tracks from scratch, without prior knowledge of driving a car. The use of these agents will be described in the following sections.

14.2 Training Time

To increase the comparability of results between baseline and transfer agents, we have to define the allowed training times for each agent type. Allowing for example a continuous agent, see , to train for the same amount of time on each individual track as the respective baseline agents would mean that the accumulated training time for the transfer agent would quickly become larger than the baseline agent, creating an unfair advantage by comparing an agent with 1.000 episodes of training with one with 5.000 episodes.

To define a fair comparison, we define a constant training time for the transfer agents, and then for each new track, we increment the baseline agents training time with this number. If we define the constant training

time for transfer agents as 300 episodes, then for track two the baseline agent will be trained for 600 episodes, and the transfer agent for 300. For track 6 this would give the baseline agent 1.800 episodes of training while the transfer agent will only have 300 episodes on track 6, but since it has experienced 1.500 episodes on previous tracks, they both have a comparable training of 1.800 episodes each. See Figure 14.1.

To define a completely fair comparison is hard, and most methods will have drawbacks. For the above method to be completely fair, one would have to assume that the quality of a training episode is identical between tracks, and that the information gained from training 300 episodes on track 1 is as usable on track 6 as 300 episodes on track 6 itself would be. This is most likely not the case, since the tracks are designed to be increasingly difficult. But since the quality of training episodes are presumably at the highest for the current track, we likely do not give the transfer agent an unfair advantage.

14.3 Tasks

The goal of transfer learning is to transfer usable knowledge from one track to another. As was defined in the problem statement in Chapter 4, the goal of the project is to improve generalizable performance by transferring knowledge through increasingly difficult tracks. An illustration of this can be seen in Figure 14.1, which shows how training will be conducted between the transfer agent and the baseline agent.

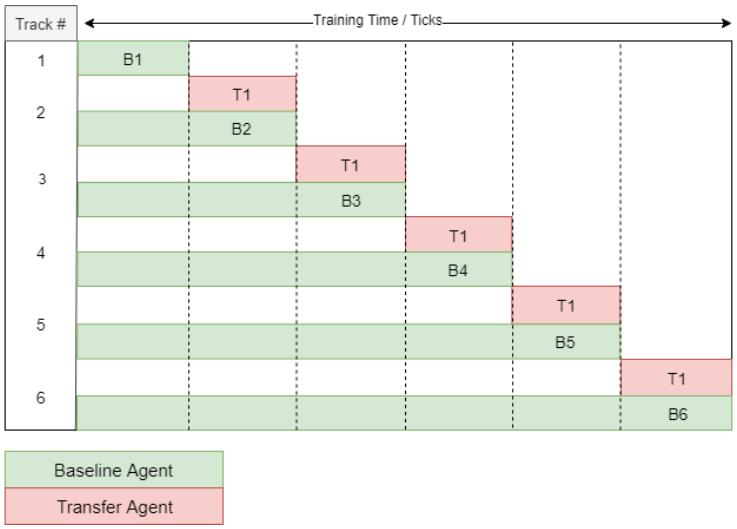


Figure 14.1: Transfer learning vs non-transfer learning agents training times

Part V

Implementation

15. CARLA Gym Environment

Due to the time restrictions and scoping of the project, manually implementing and optimizing a reinforcement learning algorithm is infeasible, so OpenAI's library is chosen to facilitate this. To use OpenAI's library of algorithms, *Baselines* [27], the CARLA simulation must be wrapped in a custom Gym environment that enables interaction between the CARLA environment and an agent training in that environment.

15.1 Gym Environment Interface

Gym is a toolkit developed by OpenAI for developing and comparing reinforcement learning algorithms. It is applicable to arbitrary agents and compatible with a lot of modern numerical computation libraries like Tensorflow. [28] The toolkit comes with many predefined environments including the classic Cartpole and Mountain Car problems that often are used to benchmark reinforcement learning algorithms. The CarRacing simulation mentioned in Section 13.1 is also one of those pre-implemented environments. All environments share a common environment interface that can be used to implement custom environments. The interface requires custom environments to have a set of functions, the most important ones being:

- The *step*-function: It should perform one action in the environment, returning the observation made based on the applied action, the rewards received in this state and lastly whether the current episode should end.
- The *reset*-function: It should reset the environment to the initial state, returning the initial observation.

The implementation of the custom gym environment will be described further in the following sections.

15.2 Interfacing With CARLA

It is possible to interface with a CARLA simulation through their Python API [29]. The CARLA simulation runs as a server in a separate process, listening and transmitting data through 3 designated ports. The Python API communicates with the server through these ports, sending requests to and receiving responses from the server. Through the API one can control which *world* is loaded. The tracks from Section 7.1 are implemented as worlds. In a loaded world one can create *actors* such as vehicles, sensors and pedestrians. Vehicles can be controlled by applying controls to them that set the throttle, steering angle, and brake intensity for the vehicle. Sensors produce data for every simulation tick which can be listened to through callback functions.
By default the simulation runs in real time, but this can be changed by using synchronous mode. In synchronous mode CARLA only simulates the world when it is told to by an API call. The control is convenient for wrapping the simulation in a gym environment, as it expects a fixed time step.

15.2.1 Modifying The Simulation

The source code for the CARLA simulation tool being available means that modified versions of CARLA can be compiled. CARLA is written in C++ and uses an underlying game engine called Unreal Engine 4. The source code has been modified by adding a few new sensors that are useful for calculating the reward function. As mentioned in Section 7.2 we punish and reward based on the amount of wheels touching grass, so a sensor supplying this information was implemented. When sensors are implemented the Python API is also changed to facilitate their access from Python. Some sensors implemented were related to performance optimization and will be talked about briefly in Section 15.5: Optimizing The Simulation

15.3 CARLA Gym Environment

As custom environments are expected to implement the methods, the following implementation of the environment methods *step* and *reset* have been made for the CARLA gym [30].

15.3.1 Reset Method

The simulation is divided up into episodes that represent a single run for the cars in the environment. The reset method is used to clear the environment from the previous episode and setup the environment for

the coming episode. The reset method broadly consists of the following components:

1. Destroy all actors in the environment such as vehicles and sensors
2. Reset episode variables such as accumulated reward
3. Create new actors at the initial starting points
4. Collect observations from the sensors of the cars
5. Return the initial observations from the cars

The components are straightforward as they are just simple calls made to the CARLA API. After this reset the agents are ready to start acting in the environment with the step method.

15.3.2 Step Method

Each episode is divided into steps representing each actionable moment in the episode where the agents are required to perform an action. Furthermore, the CARLA environment performs a tick synchronised with the steps of the agents. In specific a step for an agent consists of the following components:

1. The agent performs an action
2. The environment updates based on that action
3. The reward for being in this state gets calculated
4. An observation from the sensor on the car gets collected
5. Whether the episode should end is evaluated
6. The observation, reward, and whether the episode is done are returned

15.4 Training An Agent

To train an agent with the PPO algorithm described in Section 10, the Stable Baselines implementation is used. Stable Baselines is an implementation of RL algortihms based on OpenAI Baselines [31]. It is meant to be an improved version with greater documentation and unified structure allowing for greater customization.

Stable Baselines has two different implementations of PPO: PPO1 and PPO2. PPO1 being a CPU based implementation and PPO2 being a GPU accelerated version. As the GPU is far more suited for such a task, PPO2 is used for this project.

A model of the algorithm is instantiated with a custom environment, a policy network and various other hyperparameters. The CARLA environment is wrapped in a vectorized environment called *SubProcVec*: A vector of environments each running in sub-processes, allowing for an arbitrary amount of agents training the same model in parallel. The policy network used is Stable Baseline's CNN actor-critic policy network is the one described in Section 10.4.

An instantiated model can be used to train the agent with the *learn* method. The method trains the model for a specified amount of time steps logging training results to Tensorboard which will be further explained in Section 17.1. During training or once training is complete the model can be saved to disk, saving the policy weights for further training, transfer or evaluation.

15.5 Optimizing The Simulation

In order to lower the training time, the amount of agents being trained in parallel was increased. First from a single agent in a single simulation to four agents in four different simulations. The memory overhead of running each of these simulations was too large to run many simulations in parallel. The overhead of the actual training of each agent with PPO was minor in comparison.

To allow for an even larger amount of agents running in parallel a specific sensor and vehicle was created. The sensors are not able to see other cars and the cars do not collide with each other. By doing so the cars train on the same track in the same simulation but because they do not change the environment in any way, they can all share the same environment without knowing that any other agent is there. The change resulted in more than 100 agents being able to run in parallel. This is illustrated in Figure 15.1 where the left image shows the simulation from above and the right image shows what one of the cars in that cluster observes, not being able to see any of the other agents.

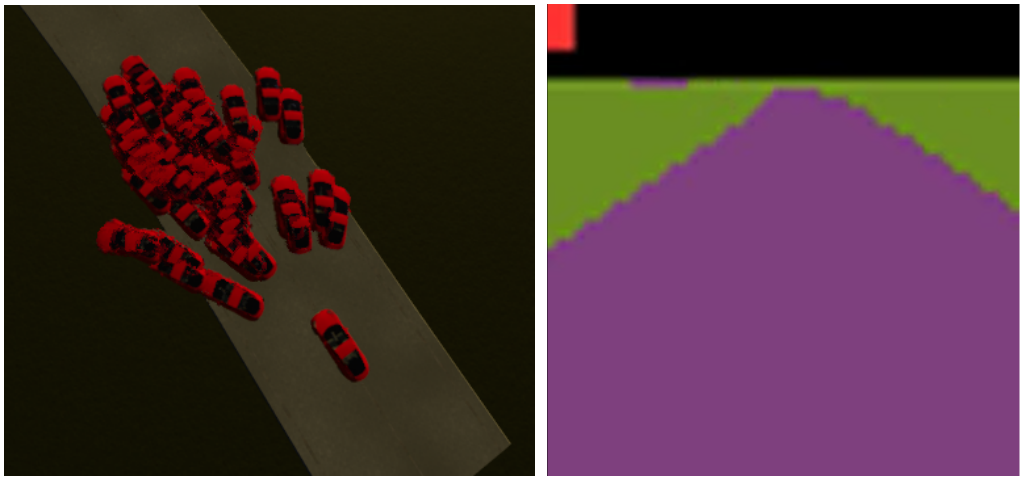


Figure 15.1: Illustration of custom sensor and vehicles. Left: Simulation from above, right: Observation from one of the agents

15.6 Hyperparameter Tuning

The PPO2 implementation has hyperparameters that can be tuned to improve training time and accuracy. During training with default hyperparameters several problems were observed which were mitigated by tuning the parameters. The tuning was done by trial and error with educated guesses based on how the hyperparameters were expected to change the training. The reasoning behind the expected changes, will be explained at each hyperparameter.

Learning Rate

The learning rate is a measure given to the optimizer, ADAM[32] in the case of PPO2, for how aggressively it should change the weights of the model to optimize the objective at each training step. Generally the learning rate is decreased over time so that the optimizer does not overshoot an optimum by updating too aggressively. Reducing the learning rate with time polynomially resulted in less sporadic performance jumps. The learning rate used at each episode is defined by the following formula, based on the default learning rate scheduler in Keras [33]:

$$lr_{episode} = lr_0 * \left(\frac{1}{1 + 0.015 * episode}\right)$$

where lr_0 is the initial learning rate, 0.015 is the decay factor, and *episode* is the current episode number.

Entropy Coefficient

The entropy coefficient is a measure for how much the entropy of a policy should affect the objective function. Remembering that PPO maximizes the objective, this means that it rewards policies with higher entropy. Reducing the entropy coefficient would incentivize policies to perform their 'best' action, but could result in premature convergence. This is because it can be locked into only performing that action and never trying new ones. The opposite problem of having a too high coefficient would result in the agent acting completely randomly, even in cases where some actions are more preferable than others.
It was at first thought that the entropy coefficient was too high because a high proportion of agents performed poorly even after hundreds of episodes. It instead turned out to be caused by the horizon parameter, so the coefficient was kept at its default of 0.1.

Horizon

The horizon, called N_{steps} in Stable Baselines, is the number of steps to simulate the agents before each policy update. To get more accurate updates the number of steps should be high because it ensures that the agent is more farsighted. In the extreme case where N_{steps} is 1, the policy only updates based on immediate rewards and does not consider any future rewards or penalties. However, if N_{steps} is too high the training time is negatively affected as the policy can only be updated once every N_{steps}, and the size of the update is restricted.

Due to the reward function design the agents receive rewards and penalties continuously at each step, so being completely farsighted is not necessary. It does need some level of farsightedness though. In cases where the agent slows down to take a turn successfully one would like the policy to associate the punishment of slowing down with the reward of taking the turn. The association is difficult to make with low N_{steps}, so the number of steps should be high enough to take that into account.
With low N_{steps} like 64 the agents performed rather poorly as expected, but once it was increased to the 256-1024 range it performed better.

Number Of Agents

The number of agents training at once is related to N_{steps} in that the batch size, the number of experiences used to update the policy, is their product. Whereas N_{steps} ensures farsightedness, the number of agents ensures more consistent and general updates. More agents can experience more variance in the environment lowering the chance of making poor updates

based on lucky or unlucky experiences.
Having many agents does come at a higher simulation time cost though. As mentioned previously the simulation can support more than 100 agents at once, but it seems like keeping it around 20 is better for the overall training time.

Clipping Range

The clipping range ϵ is, as mentioned in the PPO objective Section 10.3, a measure that ensures reasonable policy updates by limiting how much they can deviate from the current policy. Even though it may seem similar to the learning rate it has a different purpose: The parameter makes it so that it is beneficial for the objective to make a new policy that is slightly better in most cases, instead of a policy that is much better in a few cases. With a high ϵ a single action's probability can overshadow all other actions. With a very low ϵ a policy may never change to make one action more probable at the cost of another, even if it would be highly beneficial. Generally low ϵ results in smaller changes and high ϵ results in bigger changes to the policy, but not necessarily.
The clipping range was kept at its default value of 0.2.

Episode Length

Episode length is the final hyperparameter that is tuned. It is the amount of simulation steps that each episode should consist of. An infinite episode length is infeasible as agents act randomly and can easily get into situations where they are far from the track and learn nothing new.

The initial approach to solve this was to scale the episode length linearly with the episode number, so that early episodes are short and late episodes are long. The problem with this approach is that it is suddenly difficult to compare graphs of episode rewards: The potential reward is increasing so episode rewards can go up while exploitation goes down. To remedy the problem each episode is a fixed length and agents start at random points along the track, so they get to experience the entire track but only in segments of what corresponds to 35 real time seconds.

With the hyperparemeters tuned, baseline agents with more stable performance improvement can be trained on all tracks. In the following section the implementation of knowledge transfer between these agents will be explained.

16. Knowledge Transfer

This chapter will look at how each of the different approaches to transfer described in Chapter 12 are implemented.

16.1 Weight Transfer

As was described in Sections 12.1 and 12.2, one approach to transfer that we explored entailed moving layers of weights from one agent to another, to potentially improve its training-session in some way. When saving an agent the Stable Baselines library exports it to a zip file. It contains three overall elements used to describe the agent:

1. A data-file which contains all the hyperparameters and settings used for training
2. A parameter_list file which contains a list of all components used in the model
3. A parameter file of the type npz which is a format used to export multiple arrays, where each folder is a component in the model containing its weights and biases as an array. E.g c1 would be the first CNN layer and pi would be the policy layer.

The overall structure of the agent save is illustrated in the tree in Figure 16.1

To allow for quick testing of full and selective weight transfer between agents and tracks, a program was written to easily facilitate it. It requires the path for the agent from which and to which the parameters should be transferred and a list of the parameters that should be transferred.

The following section will look at the implementation of a more complex approach to transfer.

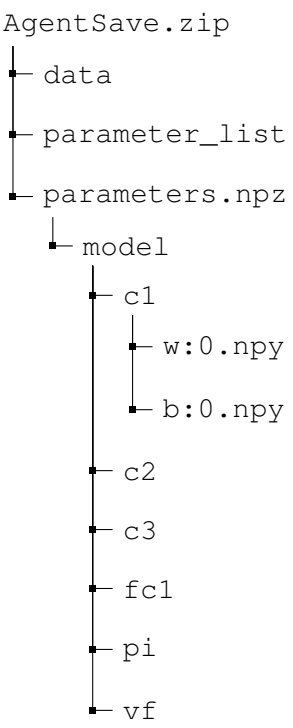


Figure 16.1: Agent save file structure

16.2 Imitation In Times Of Uncertainty

As described in Section 12.3, another transfer approach we want to explore is imitation in times of uncertainty. In this transfer approach, instead of directly transferring knowledge in the form of weights, we instead inject knowledge at specific times of uncertainty. While otherwise letting the TTA learn by itself - Closely related to a teacher-student relationship.

16.2.1 Uncertainty - State Counting

In Section 12.3, the term *uncertainty* was defined as being related to how many times a state has been seen before.

How the uncertainty is calculated can be seen in Equation 16.1. Where s is the current state, s_{count} is how many times the agent previously have seen state s while also imitating the STA, which acts as a director for the TTA, and MI is the max imitation count allowed before no more imitation will be allowed for a specific state. The uncertainty is limited to the interval $[0, 1]$.

$$uncertainty(s) = max\Big(1 - \frac{s_{count}}{MI},\ 0\Big) \tag{16.1}$$

The uncertainty tells us how uncertain the TTA is of a state, depending on how many times it has seen it before.

Counting States

To get the value of s_{count}, we need a way of counting which states the agent has seen and how many times. Since a state for the agent is defined as a 50x50 RGB image, it means that for each pixel there are 256^3 different values, and since there are 2500 pixels it means there is a total of $16.777.216^{2500}$ unique states, which is a totally unfeasible number to work with.

To work around the large number of states, we need to cluster similar states and count how many times the agent sees a cluster instead of the individual states. Since the agents input is from a segmentation sensor, it means that only 4 different pixel colors are in use given the environment modelled. This reduces the amount of unique states to 4^{2500}. While reduced by orders of magnitude, this is still a very large number.

To group the 4^{2500} unique states into a manageable amount, we slowly build a list of cluster centroid images as the agent trains. At the start of training, there are no centroids in the list. Whenever the agent sees a new state, we check all centroid images and if the new state is similar enough to one of the centroids, we increment the counter for that centroid and otherwise add the new image to the list as an entirely new cluster centroid.

The similarity is calculated as the hamming distance[24] between the images, by comparing the amount of identical pixels. This threshold of when two images are similar enough to be clustered together is a hyperparameter value, and was chosen to be 2250 pixels out of the 2500 possible pixels. An illustration of how this clustering and creation of new centroids works can be seen in Figure 16.2. If two clusters intersect, then the one which was created first will be dominant, and claim all images in their intersection, since the list is only checked until any similar cluster is found.

By using this method with a reasonably adjusted similarity threshold, we group the 4^{2500} unique states into 100-600 clusters of similar images. This means that for every step we only have to compare the state with up to 100-600 different centroids, instead of 4^{2500} unique states. The specific amount of clusters depends on the similarity threshold and the chosen track. Examples of centroid images can be seen at Figure 16.3. These

Figure 16.2: Illustration of how new states are clustered, in relation to the cluster centroid images.

depict common situations such as a turning road. More radically different centroids can show up from rare situations like a car rolling over.

16.2.2 Polling Rate

We expect the TTA to become more and more confident and correct in its decisions, which should be expressed in how often the TTA should imitate the director, so we introduce a new hyperparameter, the polling rate.

The polling rate is a value in the interval $[0, 1]$, which defines how likely it is that the TTA should poll the director for an imitation action. A value of 1 means that it is very likely that the TTA will imitate the director, and a value of 0 means to never imitate. The polling rate should preferably decrease with time as the TTA improves.

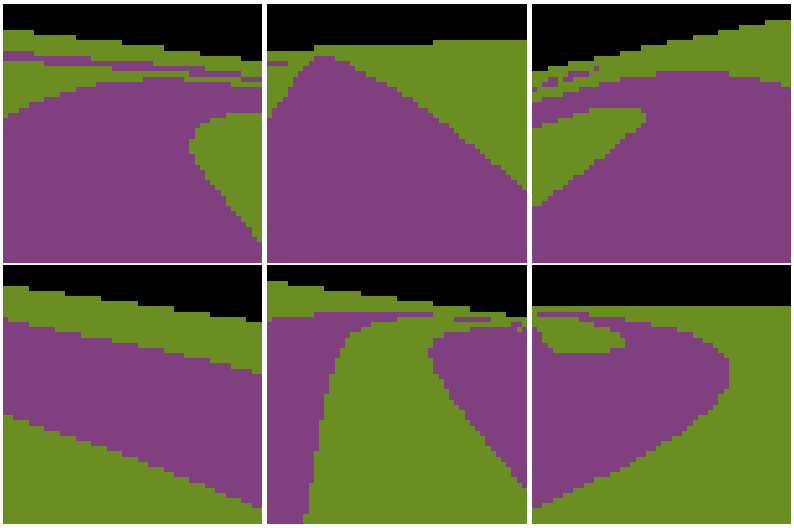


Figure 16.3: Example of cluster centroid images for track 6

16.2.3 Imitation Chance

To calculate the final imitation chance, given a state s, we multiply the polling rate with the state uncertainty from Equation 16.1. The imitation chance is formalized in Equation 16.2.

$$imitationChance(s) = pollingRate * uncertainty(s) \tag{16.2}$$

Dependent on the imitation chance, the TTA might choose to imitate the director. When imitating, the TTA polls the director for which action it would take given state s, and then performs that exact action and observes the changes in reward and environment.

16.2.4 Imitation Advantage

PPO2 uses a value function to evaluate how good a state is expected to be. The value function is estimated based on rewards achieved from doing actions. If an action receives a higher reward than was expected from the state, it is said to have a positive advantage. With the training we seek to increase the action probability for actions with a positive advantage, to aim for these higher rewards.

While imitating, the TTA's value function is updated based on the rewards it achieves, and will at some point closely approximate the correct value function. The problem is that early in training the value function is wrong, and the value function is used to calculate the advantage. This means that the TTA does not know that the directors actions are advan-

tageous, and will therefore not update its policy to make the directors actions more likely.

In the Q-learning variant of the method [23] this is handled by setting the value of best actions to that of the director and all other actions to a low bias value. With our method variant one cannot simply set the value of actions, as there is no action-value function. Instead, to further inform the TTA that the actions supplied by the director are good, we manually tell the TTA that the advantage for following the director is always positive. This lets the TTA know, that the actions given by the director are the actions it will gain the most from, and should update its policy towards.

This also means that advantages often are wrong, because the director is not perfect, especially not in the target task. This results in the TTA's policy being updated quickly to follow how the director acts, whether it is good or bad in the new domain. While performing the director's actions the TTA learns the correct value function, which it can then use to make better updates to its policy whenever it is not imitating.

17. Manual Training Evaluation

17.1 Tensorboard Logging

To visualize the reward obtained in different sessions of training agents, the data is logged to a tensorboard as seen in Figure 17.1

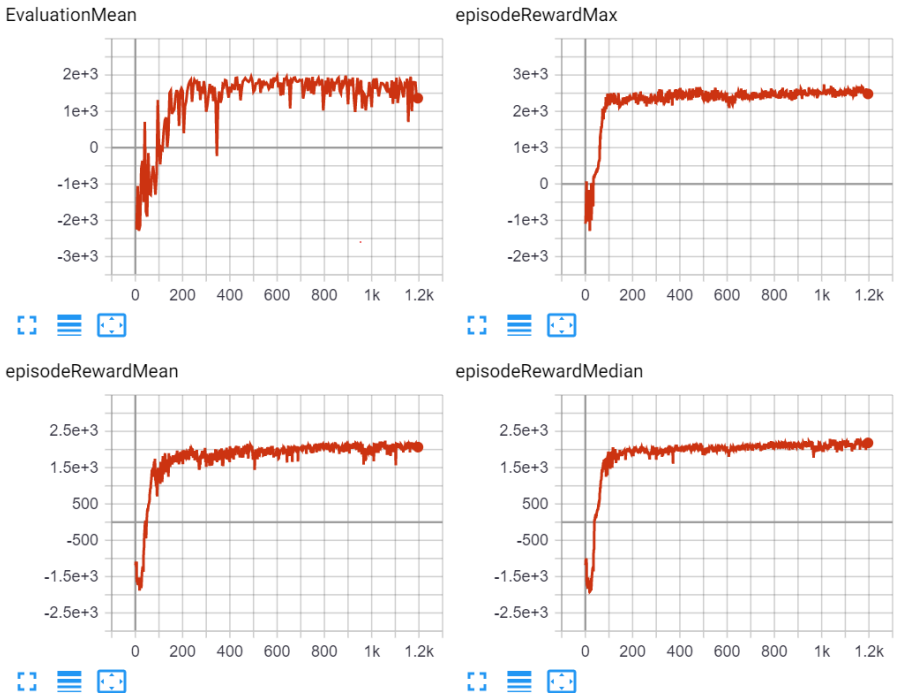


Figure 17.1: Tensorboard displaying agent performance in terms of four different metrics

The x-axis displays number of episodes in the simulation, while the y-axis displays the reward in each of these episodes. Depending on the graph the *episodeMaxReward* plots the single best agent each episode out of the 21 agents training. The *EvaluationMean* describes the mean reward of the 21 agents every fifth episode where they run a deterministic evaluation episode.

The graphs have primarily been used to manually evaluate training sessions, as the quality of the training is determined by how much reward it achieves, so comparisons between baseline and transfer agents are visually possible.

17.2 Visualization

After a set amount of episodes, image frames are stored in primary memory, which then are converted into a mp4 file and stored in the database. The visible segmentation in those videos have previously been mentioned in 6.2.1. This section will describe how those videos are used to evaluate the overall performance of the agent.

As driving a car is an action performed by humans, humans are good candidates at hand to evaluate the agent's performance and whether it is driving optimally. Having a manual evaluation technique, beside the evaluation provided by the reward functions, adds another level of evaluation, which have had an influence on the progressive design of the reward functions. One instance of such an evaluation was the elimination of obtaining a high reward for driving in circles within the track, which would not have been directly visible by the results in Figure 17.1.

Figure 17.2: Post-observation numbers applied to semantic segmentation videos

Because the videos are stored with the intention of doing manual evaluation, information about the simulation run is presented on the video post-observation, so that the person evaluating has access to information, which is not available to the agent. This can be seen in Figure 17.2.

Part VI

Findings

18. Results

This chapter will contain the results of following the evaluation plan from Chapter 14 to do knowledge transfer with the methods outlined in Chapter 12. Each method will be evaluated with three evaluation metrics which were introduced in Section 9.1:

- Jump Start: The average reward improvement compared to the baseline agent in the first 5 episodes.
- Quicker performance improvement: A combination of the episode at which the accumulated reward reaches its minimum and the episode that it reaches zero.
- Asymptotic Performance: The average of the 10 best concurrent episodes of reward.

Asymptotic performance will generally be shown as a percentage increase/decrease of the baseline agent's metrics on the same track.

18.1 Baseline Agents

The baseline agents are agents trained for each track without any transfer and using the hyperparameters specified in Section 15.6. The baseline models function as a control group that the transfer agents can be compared to. As specified in the evaluation plan set up in Chapter 14, the baseline agents learn for 300, multiplied by the track number, episodes.

The results the baseline training is shown as the blue line in Figure 18.1. The plotted results reflect the reward over a number of episodes by using results from the 21 agents being trained. The solid line follows the mean of the agents, whereas the shaded area around the line denotes the confidence interval(CI). A confidence interval with 95% confidence level shows that there is a 95% probability that the calculated confidence interval from some future training session encompasses the true value of the reward parameter [34]. All baseline agents achieve acceptable performance, concluded from both the reward and generated mp4 files, but still

have minor flaws such as not taking turns optimally and driving slower or faster than they should.

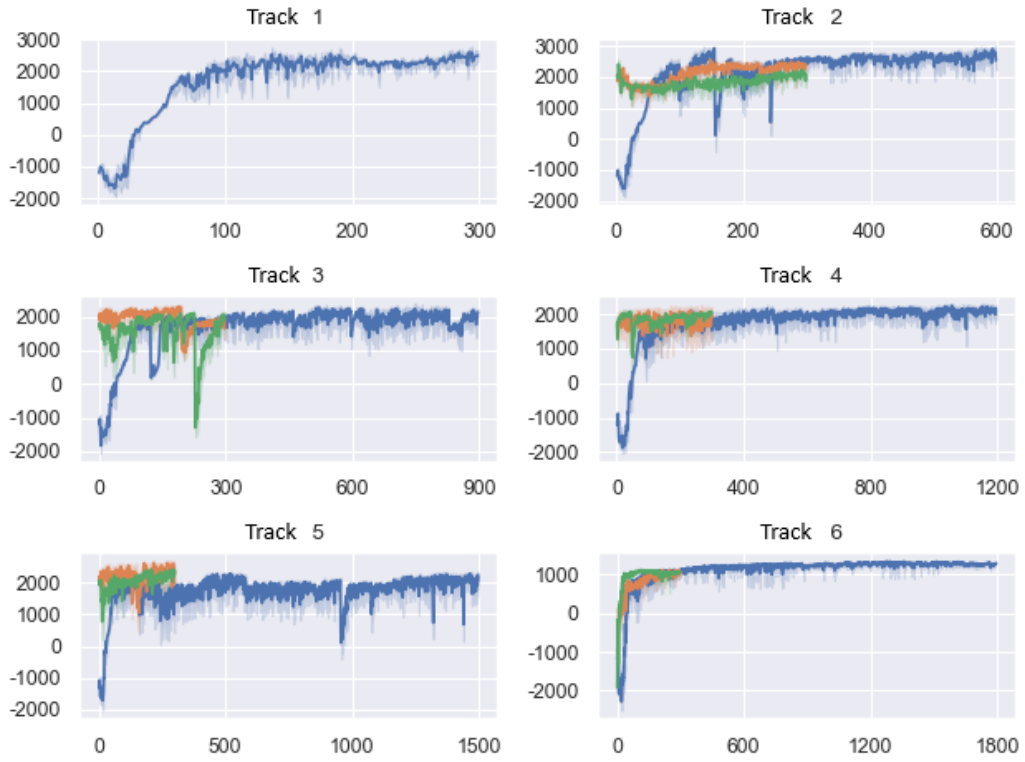


Figure 18.1: Mean rewards as a function of the number of episodes. Baseline agent (blue), Full single transfer(orange) and Full continuous transfer(green).

Another way of evaluating an agent's training session could be with cumulative rewards instead of current rewards [35]. An illustration of the cumulative rewards from the training sessions shown in Figure 18.1 can be found in Figure 18.2. There are three primary metrics that are better visible in this kind of plot in comparison to the raw training reward over time:

1. The minimum of the curve shows the amount of reward that must be sacrificed before the agent starts being reward positive.
2. The zero crossing shows how long it takes the agent to regain the reward it lost to learn the given task.
3. The asymptotic slope shows how good the policy has become after the agent's performance stabilizes.

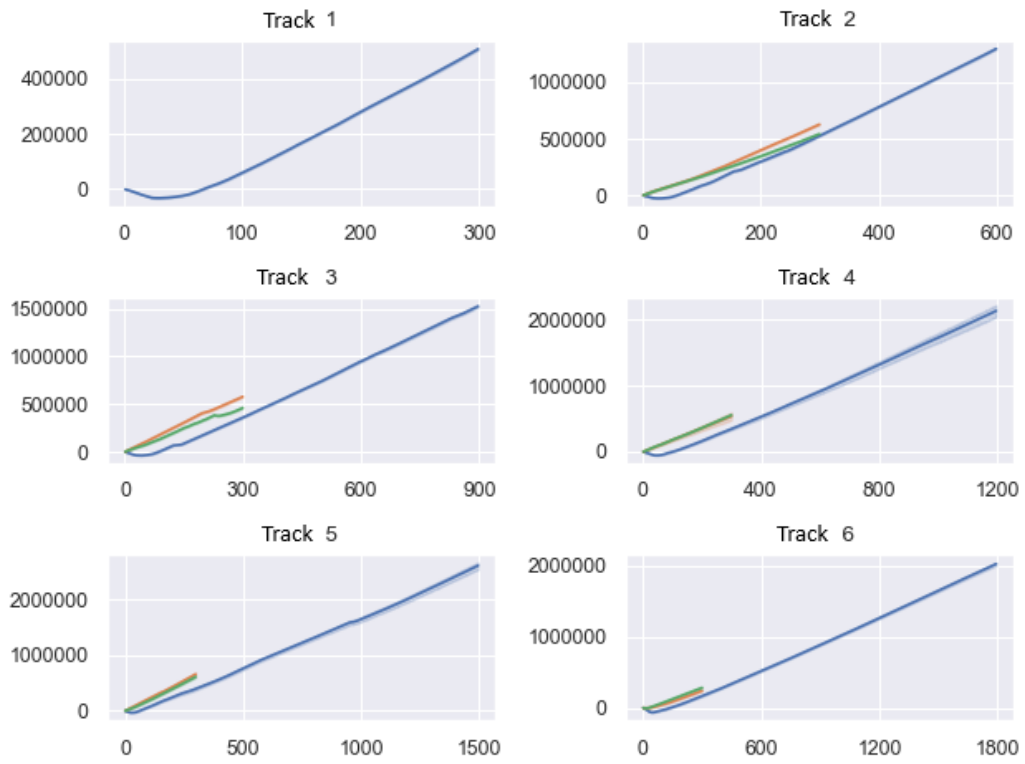


Figure 18.2: Cumulative rewards as a function of the number of episodes based on training data from Figure 18.1. Baseline agent (blue), Full single transfer (orange) and Full continuous transfer (green).

		Track 1	Track 2	Track 3	Track 4	Track 5	Track 6
Baseline	*Slope (rew/ep)*	2188	2449	1814	1882	1810	**1150**
	Min (ep)	29	27	41	47	27	45
	Zero Cross (ep)	68	58	87	101	61	140
	Jump Start (rew)	−1159	−1190	−1326	−1094	−1210	−1092
Full Single	*Slope (%)*		−12.2%	+2.8%	−11.2%	+11.4%	−18.4%
	Min (ep)		1	1	1	1	15
	Zero Cross (ep)		1	1	1	1	33
	Jump Start (Δ)		+3230	+3340	**+2802**	**+3350**	**+454**
Full Continuous	*Slope (%)*			−5.4%	−6.0%	+7.8%	−16.2%
	Min (ep)			1	1	1	12
	Zero Cross (ep)			1	1	1	31
	Jump Start (Δ)			+3054	+2649	+3193	+53
Selective Single	*Slope (%)*		−18.7%	−36.5%	−47.5%	−5.6%	−9.4%
	Min (ep)		13	16	43	17	31
	Zero Cross (ep)		29	50	129	34	139
	Jump Start (Δ)		+142	+110	−271	+105	−272
Selective Continuous	*Slope (%)*			−12.0%	−13.8%	+12.3%	−13.3%
	Min (ep)			14	9	12	27
	Zero Cross (ep)			34	28	31	88
	Jump Start (Δ)			+47	+113	+224	+48
Imitation Single	*Slope (%)*		**+8.9%**	−18.3%	−12.5%	+2.4%	−15.1%
	Min (ep)		1	20	1	1	56
	Zero Cross (ep)		1	54	1	1	152
	Jump Start (Δ)		**+3255**	+51	+1422	+2090	+3
Imitation Continuous	*Slope (%)*			**+26.8%**	**+1.0%**	**+15.9%**	−5.5%
	Min (ep)			1	1	1	96
	Zero Cross (ep)			1	1	1	198
	Jump Start (Δ)			**+3449**	+2714	+3017	+200

Table 18.1: Results for the different transfer methods. Results are in **bold** when they are the best performance achieved for the metric on the respective track, second best are underlined. E.g. Imitation Continuous has the best performance in the metrics of *slope* and *jump start* on track 3 compared to other transfer methods.

18.2 Full Transfer

18.2.1 Single Track Transfer

With single track full transfer the training was performed by transferring all the baseline model's weights from track 1 to track 2, then baseline 2 to 3 etc. and letting the model learn from there on. This approach resulted in varying levels of success with regard to the validation metrics which can be seen in Table 18.1 under the *Full Single* agent. Generally the jump start and accumulated rewards were high but often at the cost of asymptotic performance, which is highlighted as the slope of the accumulated reward as seen in Figure 18.4. This might be caused by the fact that the tracks are different and when the agent from one track is moved to another, it already knows the fundamentals of driving which results in the head start. Some negative transfer may occur because the agent is set in its ways from the source task. Another factor might be that it only trained for 300 episodes but the baseline agent trained for $300 * TrackNr$ episodes but this is unlikely to have a big effect as the performance seems to have converged.

18.2.2 Continuous Track Transfer

In continuous track transfer the same agent first trains on track 1 then 2 then 3 and so on iteratively changing weights while transferring, and moving the same agent through all the tracks. Table 18.1 shows that this transfer method sometimes provides better metrics than single track transfers like in the case of transfers 2-3, but also sometimes significantly worse in the case of transfers 4-5.

18.2.3 Comparison

An overall comparison between the single and continuous reveals that for full weight transfers, the single transfer from the baseline agents, are marginally better than iteratively building on to the same agent.

18.3 Selective Transfer

Selective transfers were performed as both single and continuous transfers as it was done with the full transfers. The transfers were attempted with different layer selections and no freezing. The results shown in this chapter are from the selective transfer of the three CNN layers and the fully connected layer, but all selective transfers had similar results. Neither of the single or continuous transfers performed well, getting outperformed by the baseline agent in asymptotic performance in almost all

cases.

In one case the selective continuous transfer performed well: From track 4 to 5. It performed 12.3% better than the baseline which is close to the best transfer performance on that track, see Table 18.1. The performance could probably be attributed to luck as all other transfers were very poor. The graphs for the transfer training can be seen in Appendix A where it is clear that the training was rather unstable on most tracks. As opposed to full transfer, which sometimes also lost asymptotic performance, the selective transfer does not even have a significant jump start, and sometimes even a negative one, giving it seemingly no upsides.

The generally poor performance with selective transfer may be because of the limit in training time. The selective transfer agents only had 300 episodes of training, whereas the baseline agents, which they are compared to, had many more episodes on the later tracks. The selective transfer agents may be more similar to baseline agents than they are to full transfer agents, because they have a lot more to learn and may therefore need more episodes.

18.4 Imitation Transfer

Imitation transfer was, as with the other methods, performed both as single track and continuous transfer. This is mostly done because the imitation method from Section 12.3, that was adapted to our problem, found better results with continuous transfer, and it would be interesting to see if we achieve the same. The method for calculating the uncertainty is state counting as described in Section 16.2.1.

The results from single and continuous imitation transfer are shown in Figure 18.3 and as accumulated reward in Figure 18.4. Generally the imitation single track transfer results are comparable in performance to that of the full transfers. While the continuous imitation outperforms single transfer. This is also clear in Table 18.1 where continuous imitation outperforms all other transfer methods in asymptotic performance (slope) and is the second or best in most jump starts.

Notably the single imitation transfer from track 1 to 2 performed very well, which may be the reason behind continuous imitation doing so well on most other tracks. The initial imitation transfer performance could be because the two tracks are very similar in some way, but full transfer did not see the same performance. Another explanation could be that track 1 agents generally drive very fast, forcing the TTA to attempt to drive faster

on track 2 as well. The incentive may have been transferred along with the continuous imitation resulting in greater overall performance.

Imitation transfer resulted in the least amount of negative transfer when transferring to track 6, but was still outperformed by the baseline agent. This is possibly because the incentives learned throughout the different tracks were not applicable to track 6. Imitation transfer may result in less negative transfer than full transfer because the TTA learns the correct value function from scratch as opposed to using the source task values. Imitation agents, intuitively, would have an easier time getting rid of the negative parts of their policies because of their correct value function estimation.

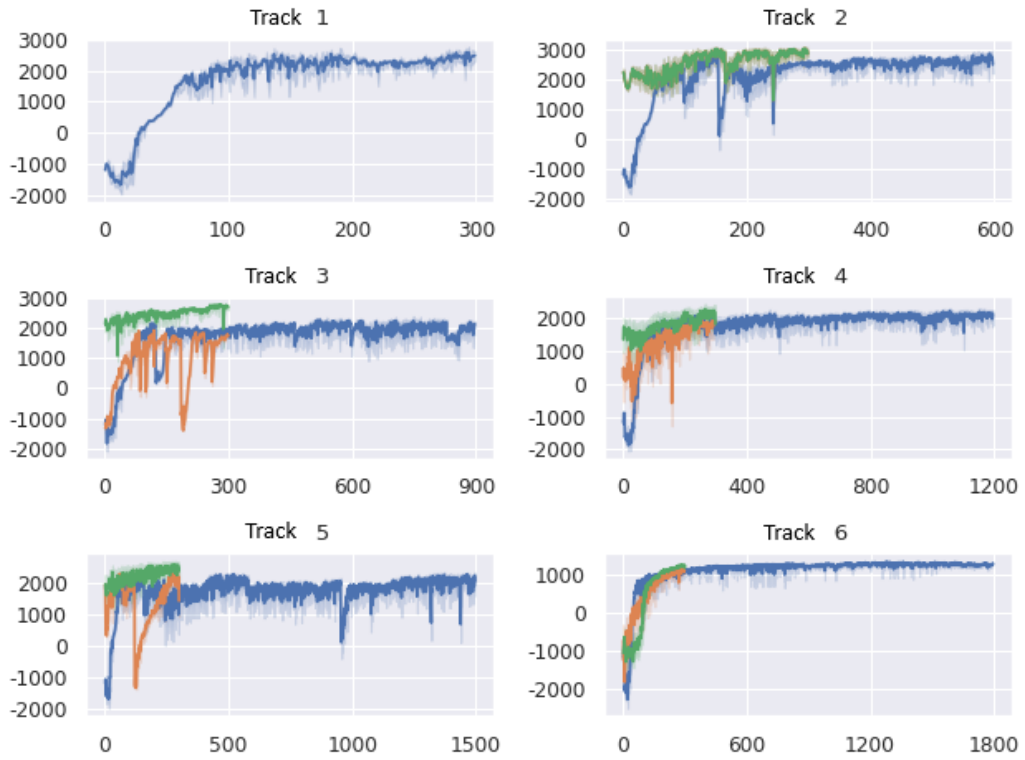


Figure 18.3: Mean rewards as a function of the number of episodes. Baseline agent (blue), single imitation transfer (orange) and continuous imitation transfer (green).

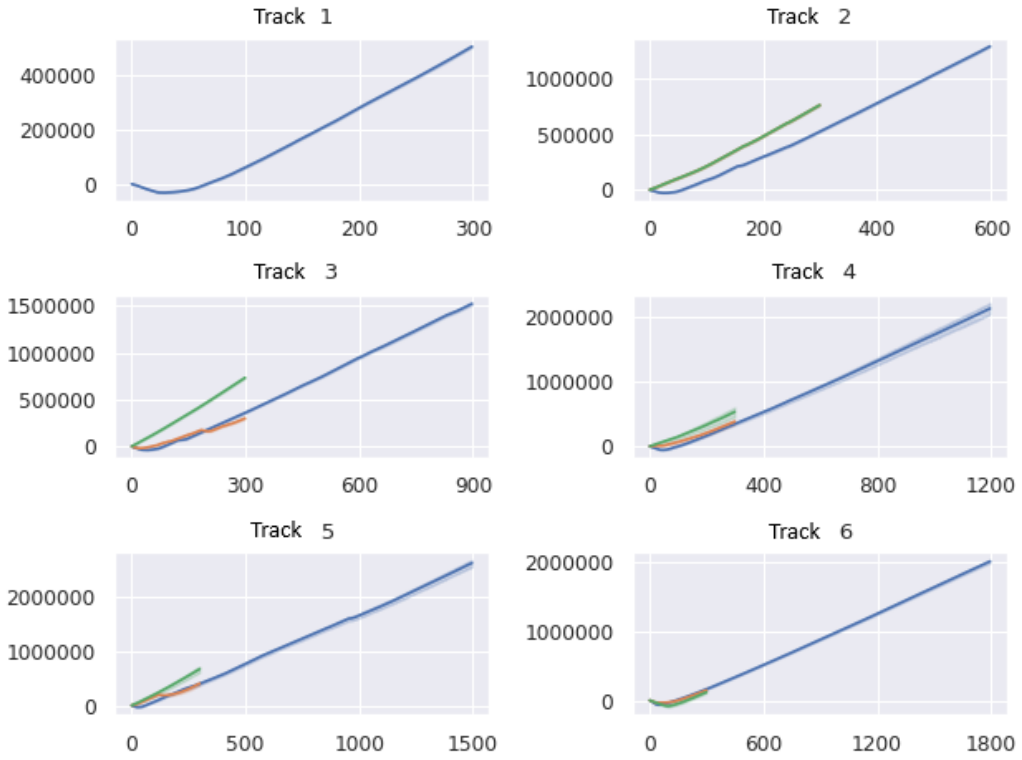


Figure 18.4: Cumulative rewards as a function of the number of episodes based on training data from Figure 18.3. Baseline agent (blue), single imitation transfer(orange) and continuous transfer(green).

19. Evaluation

This chapter will evaluate upon the performance that was achieved from training the agents. It will highlight areas where the agents are at near optimal performance and specific points where they could be better. The performance is evaluated by looking at which paths around the track that the agents take at different points in their training. The final performances will be evaluated as to whether they are near optimal and whether generalizable performance was achieved.

19.1 Training Path Plots

To gain some more insight into how the agent learns to drive, we have plotted the paths it drove for each episode on each track. We tracked the best, worst and median cars for each episode and logged their paths on an image of the track they drove on. An example of such a picture can be seen at Figure 19.1, where the blue lines are the first 1/3 of the episodes, green the next 1/3 and red the final 1/3 of the episodes.

By looking at Figure 19.1 we can see how the early blue and green cars, have a larger tendency to end up in the grass, while the final red cars are more centered on the road and almost never end in the grass, as would be expected by training the agent for longer.

19.1.1 Preferable Starting Points

An insight gained by the path plots, is that some tracks contain very preferable starting points and undesirable starting points. This observation can be seen very clearly on track 4, as the minimum and maximum car paths are plotted. These path plots can be seen at Figure 19.2.

On the max car plots, it is clear that the max cars almost never touch a large segment of the track. While the min cars are a lot more prominent in the areas the max cars are not.

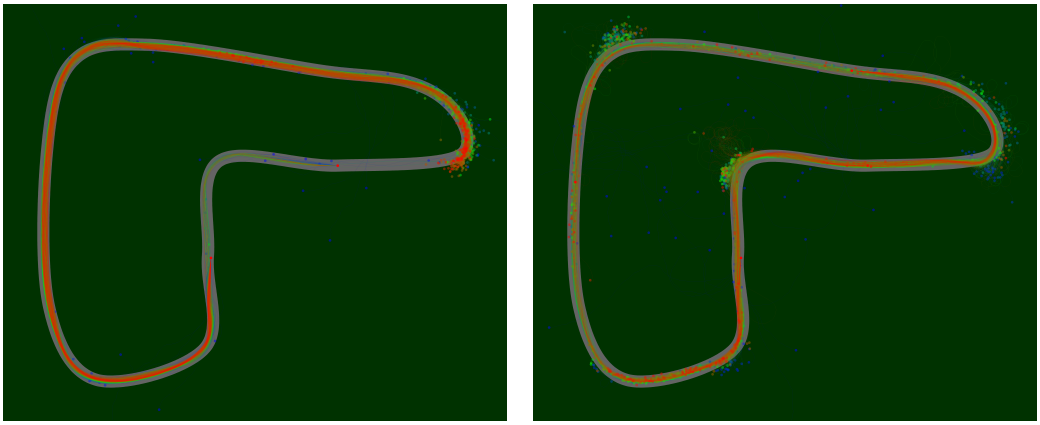
Figure 19.1: Median car paths for track 6.

It is furthermore very visible that the most preferable starting point have been right after the turn in the middle, where most of the max cars started. While the most undesirable starting point, seems to be the one right before the turn at the right side of the image.

Actions Taken To Reduce Impact

To avoid these preferable and undesirable starting points skewing our results too much, we make sure that an even amount of cars start at each starting point. This avoids the problem where some episodes randomly have a larger amount of cars in either the good or the bad starting points, and thereby skewing our results for the better or worse.

This also means that simply using the max car as a metric for how good the agent is, might be unfair, since the rewards gained by those agents would be for cars that never touched half of the track.



(a) Max cars **(b)** Min cars

Figure 19.2: Max and Min car paths on track 4 for each training episode - Full scale pictures can be seen in the Appendix.

19.1.2 Racing Lines

As seen on Figure 19.1 the red agents tends to follow a path close to the racing line [36]. The racing line is defined as the fastest possible path around a track, minimizing the angle of corners by taking corners from the outside to outside, through a center-point called the apex as seen in figure 19.3. The agents are rewarded for driving far and fast and penalized for turning, which would result in the highest accumulated rewards being obtained by agents following the racing line, as it minimizes the angle of turns and maximizes distance traveled along the spline.

Figure 19.3: Example of racing line

19.2 Generalizable Performance

The final agents performed well on the tracks that they were trained on, but that does not mean that they perform generally well.
A measure for general performance could be how well the agents perform on new tracks that they have never seen before, without training. That performance can then be compared to that of an agent which has trained on it exclusively.
This was tested by taking agents trained with the different transfer methods onto other tracks, and comparing their performance to agents who exclusively trained on those tracks.
The performances were at best 15% worse than those trained exclusively on the tracks. This suggests that generalizable performance was not reached. It is however worth noting that some imitation agents performed better than the baseline agents who were trained exclusively on the track, indicating that they may be closer to general performance.

Similarly the performance on validation tracks was always lower than that of the training track. This can easily be observed in Figure 19.4 where the accumulated reward for training and validation tracks are compared. Some validation tracks may have slightly less reward potential, but not enough to make up for the performance decline. Track 1 and 3 did not have validation tracks, so are not shown.
This result coincides with the research done by OpenAI on generalization in Reinforcement Learning, where they found that *"Even with 16,000 training levels, overfitting is still noticeable"*[9]. They also suggested that *"Agents perform best when trained on an unbounded set of levels, when a new level is encountered in every episode."* which might mean that a track auto-generator is the best approach to achieving general driving performance.

While generalizable performance was not achieved, it has been shown that acceptable performance often is reached quickly. The results from continuous imitation transfer demonstrate that it is possible to, with little training, achieve great performance on new tracks as long as tracks are similar.

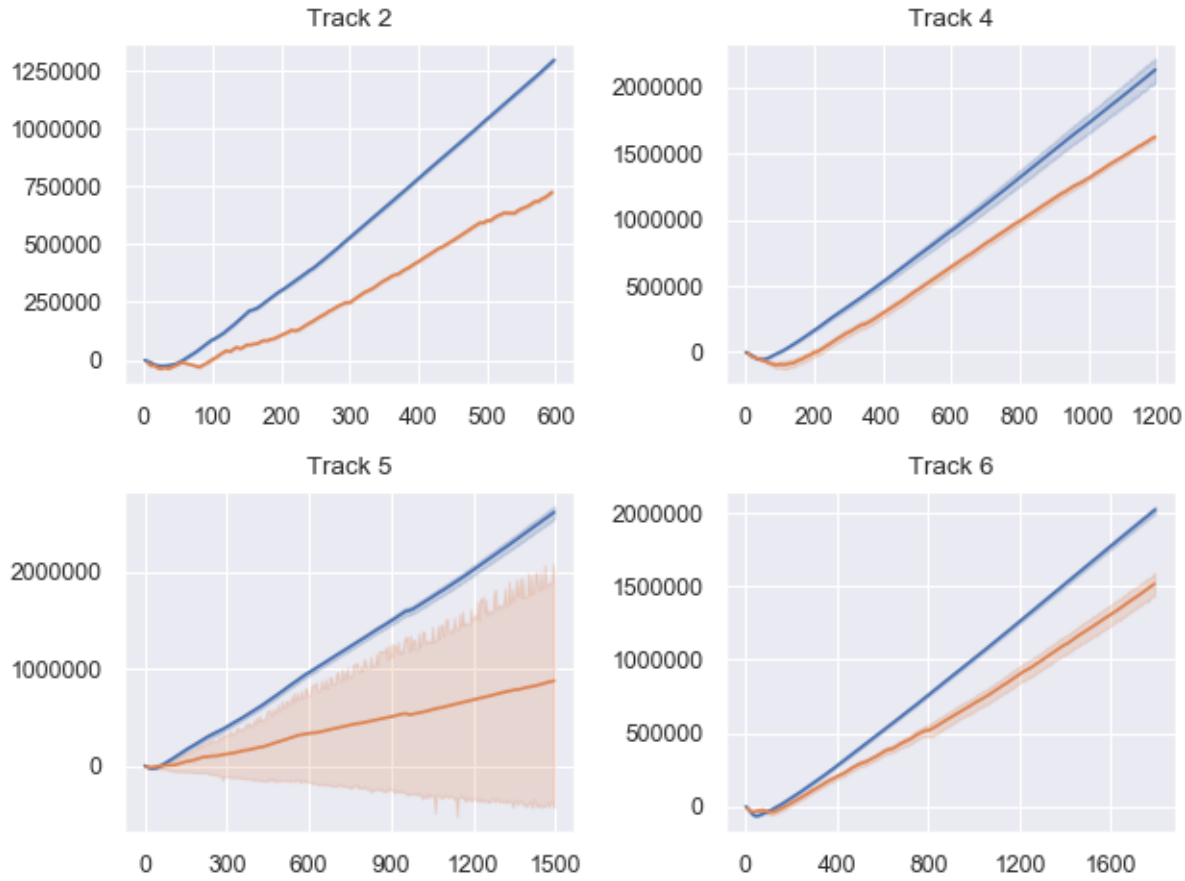


Figure 19.4: The training track accumulated reward (blue) compared to the validation track accumulated reward (orange). Data is from the baseline agents, but all transfers had a similar trend.

19.3 Track Evaluation

In Section 7.1, six tracks with different challenges were chosen and ordered based on their perceived difficulty. The fact that no transfer agent beat the baseline agent performance on track 6 suggests that it was either too hard or too different from the other tracks. It might have been beneficial to introduce one or more intermediate tracks between track 5 and 6, to better prepare the agent for track 6. Transfers from track 3 to 4 were also relatively poor, so a new track could be introduced there as well.
It seems hard to gauge how difficult tracks are compared to each other, especially if there were many different tracks. Furthermore, there is a lot

of human work required to design and implement them.
A different approach to increasingly difficult tasks was showcased in a recent article by OpenAI [37]. Levels were generated from random parameters and the ranges of these parameters were increased as the agent got better. The increasing range produced harder and harder environments, which could potentially be applied to track generation as well with random parameters for turn angles, turn frequency, road width, etc.

19.4 What, How and When to Transfer?

With results in place for the various transfer methods, it is possible to attempt to answer the three questions posed in Section 9.1 for this problem domain. They ask *what*, *how*, and *when* to transfer to effectively apply transfer learning.
The comparisons will be made from the transfer results in Table 18.1.

The information that was transferred for the full- and imitation transfer approaches was the whole previous agent, with the difference between them being in how they use the information. This was in contrast to the selective transfer, where only the CNN layers were transferred. It was found that the methods making use of all information in the previous agent produced better results in all metrics, suggesting that the whole model is *what* to transfer in this problem domain.

The starting point methods were implemented as a transplantation of all or some of the weights and biases from the previous agent, while the imitation method used the previous agent as a source of suggestions for what the new agent should try, instead of as building materials for making the new agent. Imitation is how the best results were produced, with continuous imitation in particular reaching the best slope out of the transfers on all tacks. This might indicate that *how* to transfer knowledge in this domain is through imitation methods.

For determining when transferring is worthwhile, we take another look at the table, comparing between each set of rows. Especially the slope of each set shows how they compare to the baseline that didn't use transfers. On this metric, track 6 stands out as the one track where none of the transfer methods managed to beat the baseline results. An explanation could be that the track is very different from the other tracks, being smaller and having closer together tight turns. This suggests that transfer should only be done *when* target tasks are similar to their source tasks. It is, however, possible that other transfer methods, which were not attempted, are able to be effective even when tasks are less similar.

Part VII

Reflection

20. Conclusion

In this report, we have examined different transfer learning methods in realistic environments with the objective of bringing transfer learning in reinforcement learning applications closer to being useful outside of a research settings. To try and achieve this we created six racing tracks with incremental difficulty. On each of the tracks a reinforcement learning baseline agent was trained with the state-of-the-art reinforcement learning algorithm PPO. Two general approaches to knowledge transfer were examined; direct and indirect transfer.

It was found that the indirect approach of imitation in times of uncertainty gave the best transfer results, beating the baseline agent and other transfer methods on all but one track. The direct approach resulted in high jump starts, but often at the loss of asymptotic performance.

When evaluating the general performance of trained agents it was found that little general knowledge was achieved, with agents performing significantly worse on even similar tracks, suggesting that a different approach should be taken if the goal is general performance. It was, however, shown that the imitation method, with little training, achieved great performance on similar tracks with increasing difficulty.

The domain of racing is very specific but with a more generic and well-defined approach to defining incremental tasks, the use of imitation transfer might help push incremental reinforcement learning closer to being useful in an industrial setting.

21. Future Work

Given the time frame of the project it was not possible to test and implement all ideas. The implemented designs were the ones critical for the project, but if more time was available at a further point, the following describes some possible ideas to test.

21.1 Increased Skill Ceiling

On all tracks described in Section 7.1 the trained agents managed to drive rather good within a relative short training period, which could suggest that the skill ceilings of the tracks were too low for the results to highlight the learning capabilities of the different implemented methods. To increase the skill ceiling the agents could be trained in one of the city environments already available in CARLA, which are both larger and more complex than the implemented tracks.

21.2 Cross Architecture Learning

The neural network implemented for the project showed a fast learning rate, but may have limitations in utilizing the connection in time series data. A network for utilizing this connection between time series data could be a lstm network[38], which was tested in early phases of the project, but was discarded due to a slow training time. The idea is that the increased training time could be countered by the imitation learning method, as this method showed a high jump start and does not rely on similarity between the architecture of the agents to be trained.

21.3 Hierarchical Learning

Another method for applying transfer learning to reinforcement learning, as described in Section 9.2, is hierarchical learning. The idea is to break the task of driving in a city environment into subtasks such as taking a left turn, driving straight, stopping at a crossing etc. By learning each

subtask individually and combining them, a possible reduction in training time could be achieved compared to learning the whole task at once.

21.4 Multitask Learning

Instead of transferring knowledge from one track to another one by one, the agent could be tasked with training on many different tracks at once, combining the knowledge gained to generate its policy. By not training solely on a single track at a time, a more generally applicable policy could be generated. Training on multiple tracks at once, could also counter any potential over fitting.

21.5 Uncertainty Determined by Action Probability

Given the result seen in Table 18.1 and Figure 18.3, it could be interesting to see if the imitation method could be revised to perform even better, as the currently method for performing imitation shows better results than the baselines agents do. The method only uses the states as described in Section 16.2.1. A different way of calculating uncertainty, could be to include the action probability distribution of the agent.

Bibliography

[1] Lisa Torrey and Jude Shavlik. "Transfer learning". In: *Handbook of research on machine learning applications and trends: algorithms, methods, and techniquesF*. IGI Global, 2010, pp. 242–264.

[2] Karl Weiss, Taghi M. Khoshgoftaar, and DingDing Wang. "A survey of transfer learning". In: *Journal of Big Data* (May 2016). (Accessed on 09/12/2019). URL: https://doi.org/10.1186/s40537-016-0043-6.

[3] L. P. Kaelbling, M. L. Littman, and A. W. Moore. "Reinforcement Learning: A Survey". English. In: 4 (1996), pp. 237–285. ISSN: 1076-9757. DOI: 10.1613/jair.301. URL: http://tiny.cc/7txadz.

[4] Jason Brownlee. *A Gentle Introduction to Transfer Learning for Deep Learning*. https://machinelearningmastery.com/transfer-learning-for-deep-learning/. (Accessed on 06/12/2019). 2017.

[5] Matthew E Taylor and Peter Stone. "Transfer learning for reinforcement learning domains: A survey". In: *Journal of Machine Learning Research* 10.Jul (2009), pp. 1633–1685.

[6] Marco Wiering. *Reinforcement Learning, State-of-the-Art*. URL:https://link.springer.com/content/pdf/10.1007%2F978-3-642-27645-3.pdfChapter5. Springer, 2012.

[7] Matt DosSantos DiSorbo. *Chapter 10 Markov Chains*. (Accessed on 06/12/2019). bookdown.org, 2019.

[8] Mohammad Ashraf. "Reinforcement Learning Demystified: Markov Decision Processes (Part 1)". In: (). (Accessed on 06/12/2019).

[9] Karl Cobbe et al. "Quantifying Generalization in Reinforcement Learning". In: *CoRR* abs/1812.02341 (2018). arXiv: 1812.02341. URL: http://arxiv.org/abs/1812.02341.

[10] Bharat Adibhatla. *Understanding The Role Of Reward Functions In Reinforcement Learning*. https://analyticsindiamag.com/understanding-the-role-of-reward-functions-in-reinforcement-learning/. (Accessed on 06/12/2019).

[11] Bonsai. *Deep Reinforcement Learning Models: Tips & Tricks for Writing Reward Functions*. URL:https://medium.com/@BonsaiAI/deep-reinforcement-learning-models-tips-tricks-for-writing-reward-functions-a84fe525e8e0. (Accessed on 06/12/2019).

[12] S. J. Pan and Q. Yang. "A Survey on Transfer Learning". In: *IEEE Transactions on Knowledge and Data Engineering* 22.10 (Oct. 2010), pp. 1345–1359. DOI: 10.1109/TKDE.2009.191.

[13] John Schulman et al. "Proximal policy optimization algorithms". In: *arXiv preprint arXiv:1707.06347* (2017).

[14] Chris Yoon. *Understanding Actor Critic Methods and A2C*. https://towardsdatascience.com/understanding-actor-critic-methods-931b97b6df3f. (Accessed on 06/12/2019).

[15] OpenAI. *Proximal Policy Optimization Documentation*. https://spinningup.openai.com/en/latest/algorithms/ppo.html. (Accessed on 06/12/2019).

[16] Wikipedia. *Convolutional neural network*. https://en.wikipedia.org/wiki/Convolutional_neural_network. (Accessed on 06/12/2019).

[17] Balázs Csanád Csáji. "Approximation with artificial neural networks". In: *Faculty of Sciences, Etvs Lornd University, Hungary* 24 (2001), p. 48.

[18] Sumit Saha. *A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way*. https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53. (Accessed on 06/12/2019). December, 2018.

[19] Machine Learning Guru. *Undrestanding Convolutional Layers in Convolutional Neural Networks (CNNs)*. http://machinelearninguru.com/computer_vision/basics/convolution/convolution_layer.html. (Accessed on 06/12/2019).

[20] Piotr Skalski. *Gentle Dive into Math Behind Convolutional Neural Networks*. https://towardsdatascience.com/gentle-dive-into-math-behind-convolutional-neural-networks-79a07dd44cf9. (Accessed on 06/12/2019). April, 2019.

[21] Thomas Brox Martin Riedmiller Jost Tobias Springenberg Alexey Dosovitskiy. *Striving for Simplicity: The All Convolutional Net*. 2014. arXiv: 1412.6806 [cs.LG].

[22] Anish Singh Walia. *Types of Optimization Algorithms used in Neural Networks and Ways to Optimize Gradient Descent*. https://towardsdatascience.com/types-of-optimization-algorithms-used-in-neural-networks-and-ways-to-optimize-gradient-95ae5d39529f. (Accessed on 06/12/2019). June, 2017.

[23] MICHAEL G. MADDEN & TOM HOWLEY. "Transfer of Experience between Reinforcement Learning Environments with Progressive Difficulty". In: *Faculty of Sciences, Etvs Lornd University, Hungary* (2004), p. 24.

[24] Wikipedia. *Hamming distance*. https://en.wikipedia.org/wiki/Hamming_distance. (Accessed on 06/12/2019).

[25] OpenAI. *CarRacing-vo*. https://gym.openai.com/envs/CarRacing-v0/. (Accessed on 06/12/2019).

[26] Alexey Dosovitskiy et al. *CARLA: An Open Urban Driving Simulator*. http://carla.org/. (Accessed on 06/12/2019).

[27] *OpenAI Baselines: high-quality implementations of reinforcement learning algorithms*. https://github.com/openai/baselines. (Accessed on 08/10/2019).

[28] *OpenAI Gym Toolkit Documentation*. http://gym.openai.com/docs/. (Accessed on 10/10/2019).

[29] *Python API reference - CARLA Simulator*. https://carla.readthedocs.io/en/latest/python_api/. (Accessed on 08/10/2019).

[30] stable baselines. *Using Custom Environments*. https://stable-baselines.readthedocs.io/en/master/guide/custom_env.html. (Accessed on 08/10/2019).

[31] *Stable Baselines - RL Baselines Made Easy — 2.9.0a0 documentation*. https://stable-baselines.readthedocs.io/en/master/. (Accessed on 14/10/2019).

[32] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. arXiv: 1412.6980 [cs.LG].

[33] Adrian Rosebrock. *Keras learning rate schedules and decay*. https://www.pyimagesearch.com/2019/07/22/keras-learning-rate-schedules-and-decay/. (Accessed on 06/12/2019).

[34] Wikipedia. *Confidence Interval*. https://en.wikipedia.org/wiki/Confidence_interval. (Accessed on 06/12/2019).

[35] David L Poole and Alan K Mackworth. *Artificial Intelligence: Foundations of computational agents 2E*. Cambridge University Press, 2017.

[36] Wikipedia. *Racing line*. https://en.wikipedia.org/wiki/Racing_line. (Accessed on 06/12/2019).

[37] OpenAI. *Solving Rubik's Cube with a Robot Hand*. https://openai.com/blog/solving-rubiks-cube/. (Accessed on 06/12/2019).

[38] Wikipedia. *Long short-term memory*. https://en.wikipedia.org/wiki/Long_short-term_memory. (Accessed on 06/12/2019).

Part VIII

Appendices

A. Training Results

A.1 Full Transfer

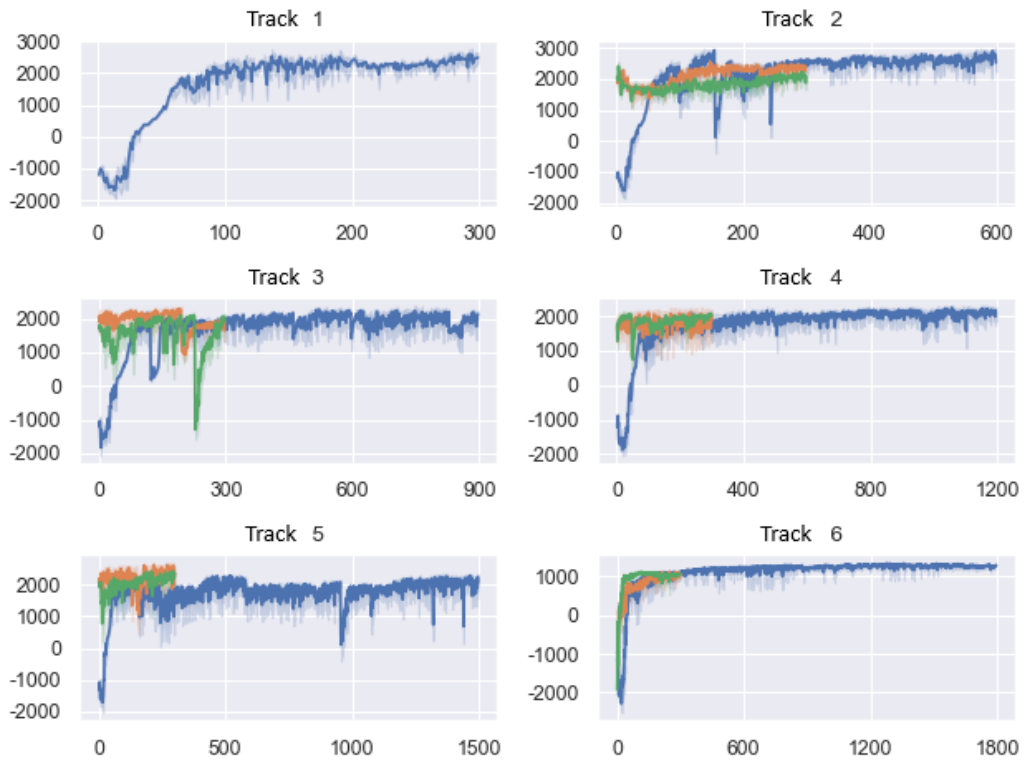


Figure A.1: Mean rewards as a function of the number of episodes. Baseline agent (blue), Full single transfer(orange) and Full continuous transfer(green).

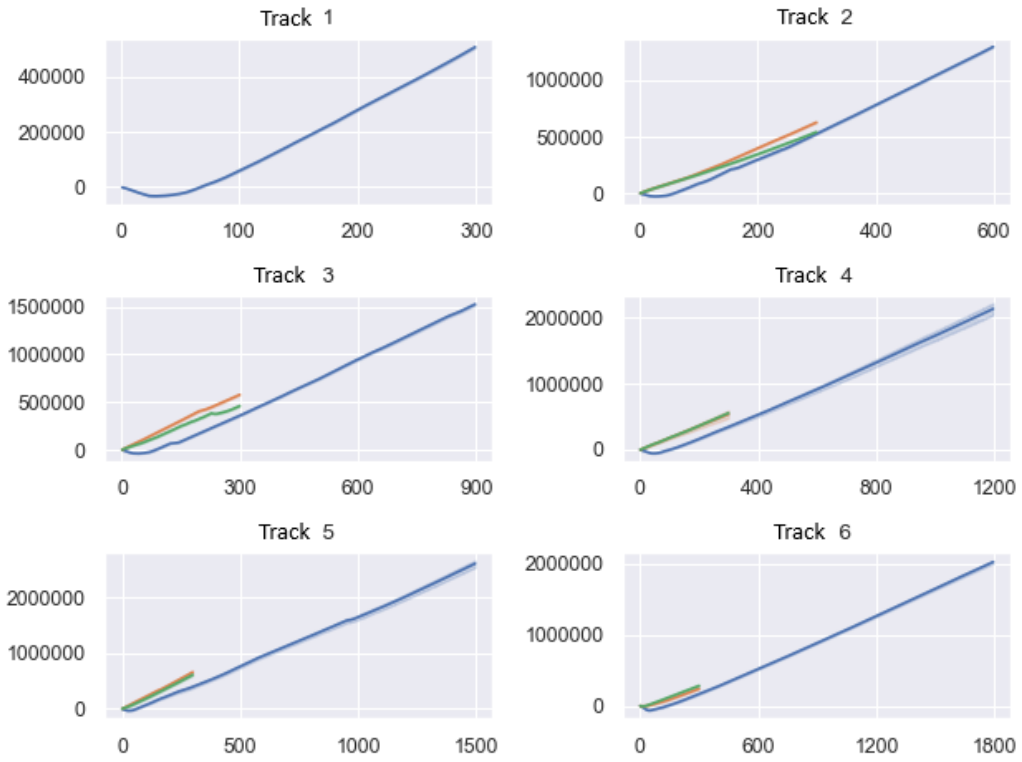


Figure A.2: Cumulative rewards as a function of the number of episodes based on training data from the previous figure. Baseline agent (blue), Full single transfer(orange) and Full continuous transfer(green).

A.2 Selective Transfer

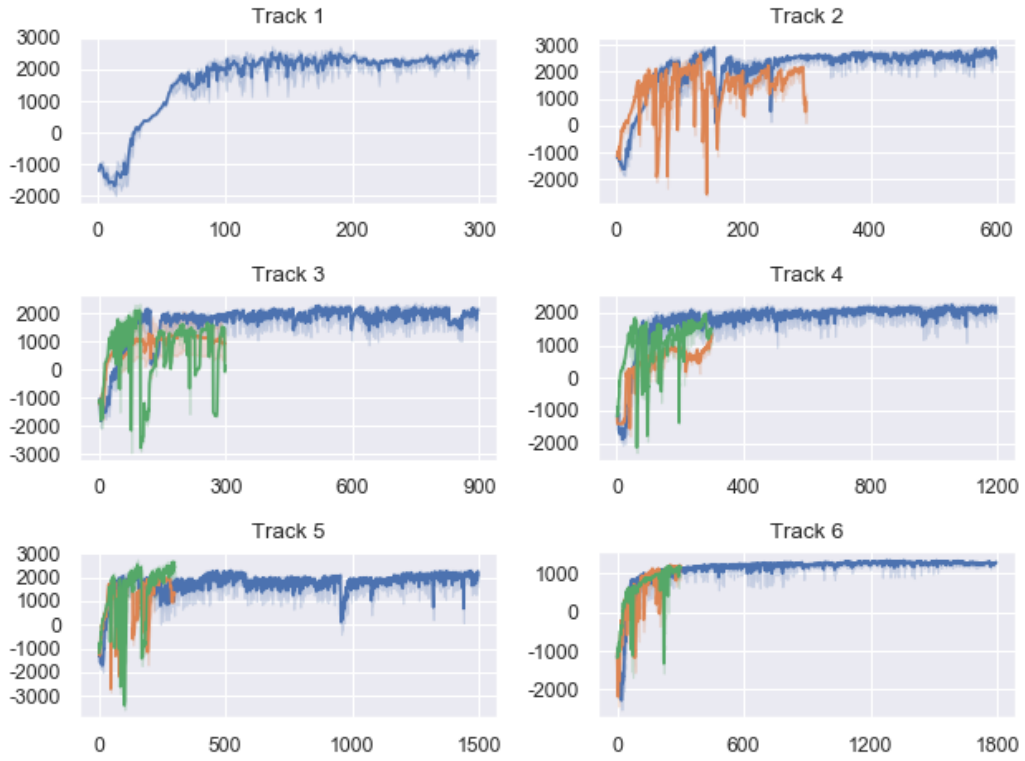


Figure A.3: Mean rewards as a function of the number of episodes. Baseline agent (blue), Selective single transfer(orange) and Selective continuous transfer(green).

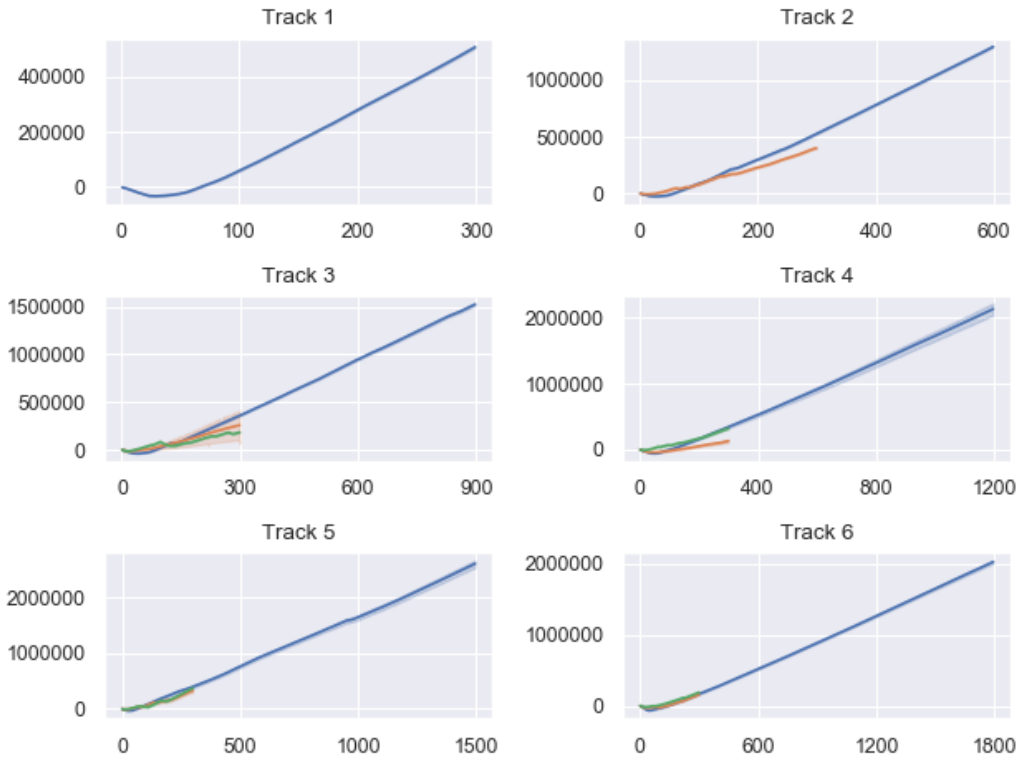


Figure A.4: Cumulative rewards as a function of the number of episodes based on training data from the previous figure. Baseline agent (blue), Selective single transfer(orange) and Selective continuous transfer(green).

A.3 Imitation Transfer

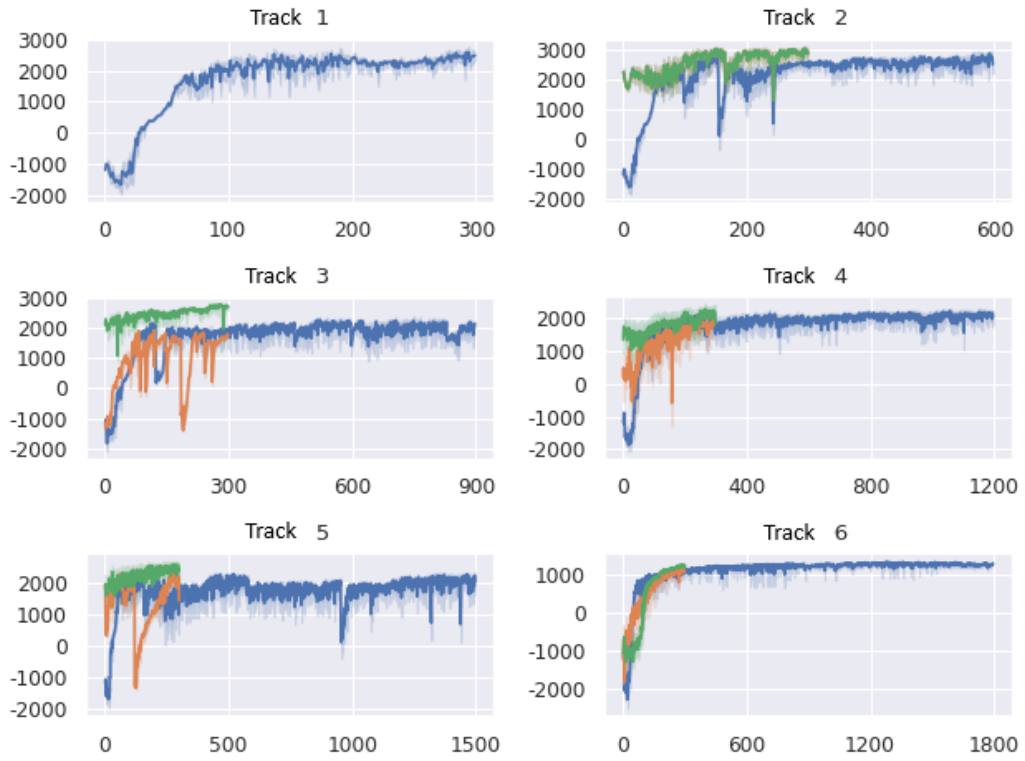


Figure A.5: Mean rewards as a function of the number of episodes. Baseline agent (blue), Imitation single transfer(orange) and Imitation continuous transfer(green).

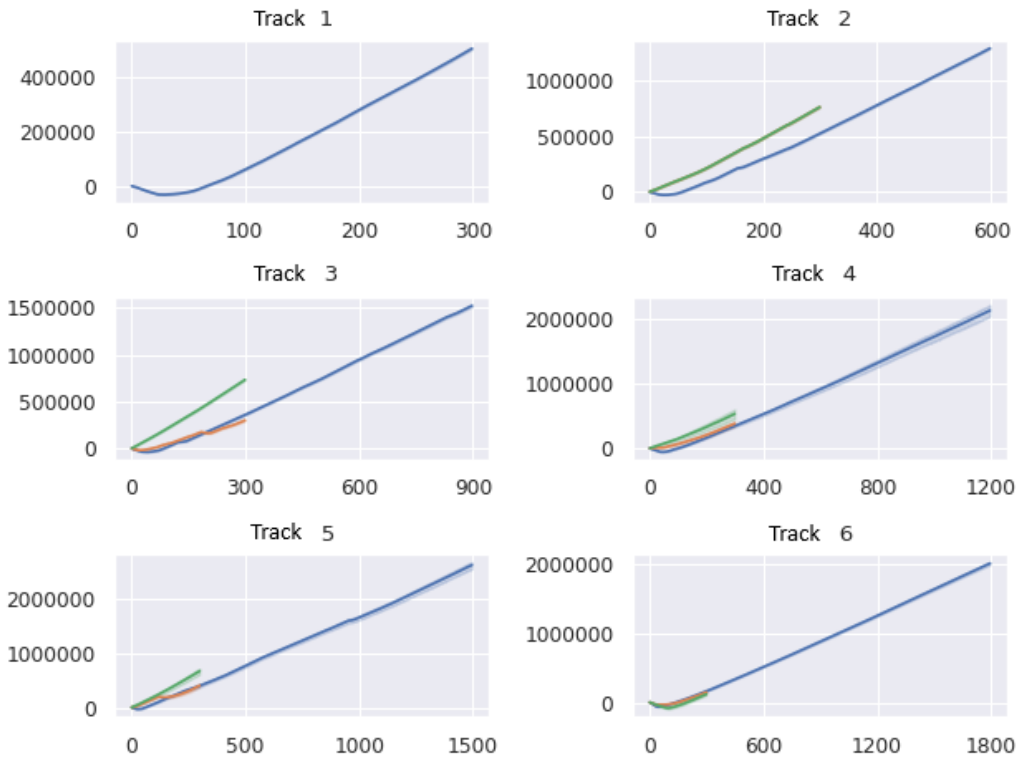


Figure A.6: Cumulative rewards as a function of the number of episodes based on training data from the previous figure. Baseline agent (blue), Imitation single transfer(orange) and Imitation continuous transfer(green).

B. Car Modelling

The car that will be used in simulation is a modeled Tesla, which will be attached with a semantic segmentation sensor. The placement of the sensor will be 2 meters in front and 3 meters above the center of the car, as illustrated on Figure B.1. The view of a more realistic and practically placed sensor would provide an almost identical source of information, as seen in Figure B.2, but would allow the segmentation camera to see the car, thus creating a segment with no informal value, since the placement of the car would always be identical.

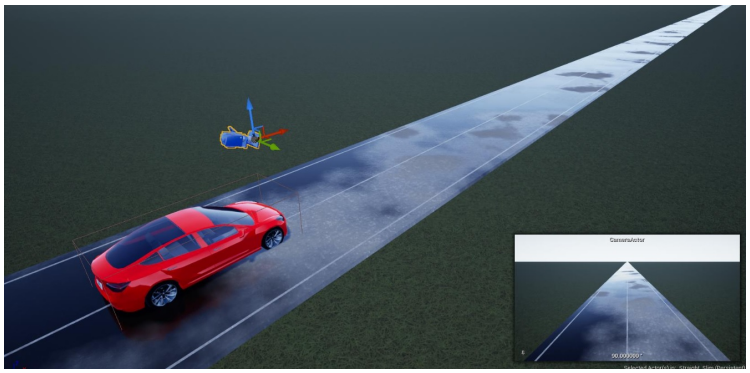


Figure B.1: Segmentation sensor placement

Figure B.2: Realistic segmentation sensor placement

C. GPS Images

The images can be found in the repository folder at the following link: `https://github.com/Reniets/RLTL_Carla/tree/master/GPS_Images`