Faculty of Mathematics and Informatics

VILNIUS UNIVERSITY
FACULTY OF MATHEMATICS AND INFORMATICS
INSTITUTE OF INFORMATICS
DEPARTMENT OF SOFTWARE ENGINEERING

Autonomous Car Driving using Reinforcement Learning and Genetic Algorithm

Autonominio automobilio vairavimas naudojant skatinamąjį mokymą bei genetinį algoritmą

Master's Thesis

Student: Žygimantas Milvydas
Supervisor: Prof. Dr. Aistis Raudys
Reviewer: Prof. Dr. Olga Kurasova

Vilnius - 2025

Summary

This study explores applications of three learning strategies for autonomous racing in the TORCS simulation environment - NEAT (NeuroEvolution of Augmenting Topologies), DDPG (Deep Deterministic Policy Gradient), and Evolutionary Reinforcement Learning (ERL) – a hybrid algorithm created during this project. which was created during this project. Each algorithm was trained and tested under controlled conditions to assess performance across three core metrics - reward progression, lap times, and driving behavior.

NEAT demonstrated strong early learning and consistent generalization across both seen and unseen tracks but achieved the lowest peak reward and slower lap times. DDPG excelled at fine-tuning policies and produced smoother and faster trajectories but struggled with overfitting and instability. ERL, which integrates NEAT-evolved architectures into DDPG's gradient-based learning, combined the exploratory strength of NEAT with the policy refinement of DDPG. It outperformed both baselines in peak reward, learning speed, and lap time performance, validating the advantage of hybridizing evolutionary and gradient-based methods.

The research confirms the potential of hybrid approaches in autonomous vehicle control and opens pathways for future work focused on multi-phase training strategies.

Keywords: Autonomous driving, TORCS simulator, NEAT, DDPG, Genetic algorithm, Deep reinforcement Learning, Hybrid algorithms, Evolutionary Reinforcement Learning.

Santrauka

Šiame tyrime nagrinėjamos trijų mokymosi strategijų taikymo galimybės autonominiam lenktyniavimui TORCS simuliacijos aplinkoje – NEAT (Neuroninių tinklų evoliucija su topologijų plėtra), DDPG (Deterministinis politikos gradientas) ir evoliucinio skatinamojo mokymosi (ERL) - hibridinio algoritmo, sukurto šio projekto metu. Kiekvienas algoritmas buvo mokomas ir testuojamas kontroliuojamomis sąlygomis, siekiant įvertinti jų veikimą pagal tris pagrindinius kriterijus – atlygio progresiją, rato įveikimo laikus ir vairavimo elgseną.

NEAT pasižymėjo stipriu ankstyvuoju mokymusi ir nuosekliu bendrinimu tiek matytose, tiek nematytose trasose, tačiau pasiekė mažiausią maksimalų atlygį ir turėjo lėtesnius ratų laikus. DDPG puikiai tobulino politiką, sukūrė sklandesnes ir greitesnes trajektorijas, tačiau kentėjo nuo persimokymo ir nestabilumo. ERL, integruojantis NEAT-evoliucionuotą architektūrą į DDPG gradientinį mokymąsi, sujungė NEAT tyrinėjimo stiprybę su DDPG politikos tobulinimu. Jis pranoko abu bazinius metodus pagal maksimalų atlygį, mokymosi greitį ir rato laikų rezultatus, patvirtindamas evoliucinių ir gradientinių metodų hibridizacijos pranašumą.

Tyrimas patvirtina hibridinių metodų potencialą autonominių transporto priemonių valdyme ir atveria galimybes tolimesniems tyrimams, orientuotiems į keletos fazių mokymo strategiją.

Raktažodžiai: Autonominis vairavimas, TORCS simuliatorius, NEAT, DDPG, Genetinis algoritmas, Gilusis skatinamasis mokymasis, Hibridiniai algoritmai, Evoliucinis skatinamasis mokymasis.

Table of Contents

Abbreviation List

ABS	Anti-lock Braking System
AI	Artificial Intelligence
ANN	Artificial Neural Network
DDPG	Deep Deterministic Policy Gradient
DNF	Did Not Finish
DNN	Deep Neural Network
ERL	Evolutionary Reinforcement Learning
GA	Genetic Algorithm
NEAT	NeuroEvolution of Augmenting Topologies
NN	Neural Network
OU	Ornstein-Uhlenbeck process
RL	Reinforcement Learning

1. Introduction

1.1 Background

In the recent years, autonomous driving has emerged as an important area of research within the artificial intelligence field. It has potential to revolutionize transportation by improving safety, reducing traffic congestion, and enhancing mobility for all.

Motorsport is an important part of the automotive industry and greatly contributes to its technological advancement. Many innovative solutions have found their way into consumer vehicles from racing series, such as Formula 1, Le Mans or the World Rally Championship. This led to significant advancements in performance, as well as safety.

Although the goal in racing is clear and simple – driving as fast as possible to achieve the fastest lap times and win the race, controlling a car at its dynamic limits of handling in order to achieve that is difficult and requires skill and knowledge.

Most of previous technological innovation that came from motorsport was focused on improving the race car. However, there is another important piece of the racing puzzle – the driver. The rise of autonomous vehicles kickstarted innovation within the field of autonomous car control and autonomous car racing has become a field where advanced algorithmic car control approaches get tested and eventually implemented into autonomous passenger vehicles, similar to classic motorsport.

At the core of autonomous driving research lies the challenge of developing intelligent control systems that are capable of making real-time decisions in complex environments. As the demand for robust and adaptable driving policies grows, the exploration of advanced learning techniques becomes increasingly relevant.

Reinforcement Learning (RL) and Genetic Algorithms (GA) are two such techniques with promising potential for autonomous driving applications. Reinforcement Learning, particularly deep variants like Deep Deterministic Policy Gradient (DDPG), enables agents to learn optimal actions through interactions with the environment by maximizing cumulative rewards. On the other hand, Genetic Algorithms, such as NeuroEvolution of Augmenting Topologies (NEAT), evolve neural

network structures and weights using principles of natural selection, often requiring fewer assumptions about gradient information and offering stable performance in high-dimensional spaces.

While both approaches have demonstrated success in various domains, their comparative strengths and limitations in the context of autonomous driving remain an open research question. Moreover, there is unexplored potential in hybridizing these algorithms - leveraging the architectural flexibility of NEAT and the fine-tuned policy learning of DDPG to potentially achieve superior performance and generalization.

1.2 Goal and objectives

This study aims to address this gap. To do so, the following goal has been set: **To develop and evaluate a novel hybrid algorithm that combines Genetic Algorithm with Reinforcement Learning for autonomous car control, and to compare its performance against the standalone approaches in a simulated racing environment**.

In order to achieve this goal, the following objectives have been identified:

- Explore the application of Reinforcement Learning and Genetic Algorithms in autonomous driving and racing.
- Conduct a comparative analysis of DDPG and NEAT within a simulated driving environment.
- Develop a novel hybrid approach that integrates the evolutionary capabilities of NEAT with the policy refinement of DDPG, and evaluate its performance against the standalone methods.

It was chosen to carry out the experiments in TORCS (The Open Racing Car Simulator), which offers a realistic and controlled environment for training and evaluating autonomous driving agents.

1.3 Research workflow

Overall workflow of the research process for this study is given below:

1. Identify research gap
    - Lack of comparison between GA and RL methods for autonomous driving
    - Unexplored opportunity to hybridize these methods
2. Define goals
    - Investigate NEAT, DDPG, and a hybrid ERL method for autonomous racing.
    - Evaluate performance via reward, lap time, and driving behavior.
3. Review literature
    - Analyze classical, supervised, and unsupervised driving methods.
    - Focus on NEAT, DDPG, and hybrid applications in TORCS.
4. Select methods
    - Use NEAT for topology evolution.
    - Use DDPG for continuous control refinement.
    - Combine both into ERL for improved performance.
5. Design experiments
    - Run simulations in TORCS with consistent inputs/outputs.
    - Train each agent for the same number of episodes.
    - Use identical reward/fitness functions.
6. Implement and train
    - Implement NEAT via python-neat, DDPG via TensorFlow.
    - Convert NEAT genomes to TensorFlow models for ERL.

- Apply OU noise and episode termination rules.

7. Evaluate performance
    - Measure reward trends, lap times, and safety violations.
    - Test generalization on unseen tracks.
    - Compare against built-in TORCS bot.
8. Analyze and conclude
    - Draw conclusions from gathered results.
    - State whether goal has been successfully achieved.

1.4 Relevance

This project sits at the intersection of two powerful AI patterns and one of the most critical real-world applications. With safety, adaptability, and efficiency being core issues in autonomous driving, such research could have a real impact due to several reasons.

Autonomous driving is one of the most transformative technologies in transportation. Major companies like Tesla, Waymo, and Cruise are investing heavily in making self-driving vehicles viable, safe, and efficient. Any advancement in decision-making algorithms, especially those that improve learning and adaptability, contributes directly to real-world applications in this field.

RL has already shown remarkable performance in domains where continuous decision-making is needed, such as robotics, games (AlphaGo, OpenAI Five), and autonomous driving. This project investigates the latest in RL. Comparing top-performing RL algorithms means it contributes to identifying the best candidates for real-world deployment.

Genetic algorithms, although less prominent in recent years, offer unique advantages like global search capabilities and robustness to local minima, which can help solve complex problems where traditional learning struggles.

In complex, dynamic environments like driving, combining the exploratory robustness of GAs with the adaptive learning of RL could lead to more generalizable and stable autonomous agents. This kind of research is essential for pushing beyond the limits of either method alone.

Using simulated environments (e.g., CARLA, TORCS, or AirSim) allows researchers to test thousands of edge cases and rare events without physical risk, making it ideal for evaluating and refining AI algorithms. This aligns with the modern development lifecycle in autonomous systems.

2. Literature Review

2.1 Background

Previous work in the autonomous car racing field can be grouped into three categories: classical approaches, supervised learning approaches and unsupervised learning approaches.

Classical approaches separate the problem into three submodules – perception, trajectory planning and using a controller to follow planned trajectory.

Instead of pre-planning trajectories and tracking them using a controller, supervised learning approaches, such as Imitation Learning (IL), learn mappings from observing actions. Supervised learning requires labelled data, which is usually provided by human experts.

Unsupervised learning approaches optimise their policies based on sampled experiences and therefore do not suffer from some of the main problems related to the other approaches to autonomous car control, such as requiring non-linear equations to be solved online or being dependent on labelled data. This project will focus on unsupervised learning techniques.

Autonomous car control approaches can be further separated into end-to-end, partial end-to-end and classical control groups.

As shown in Figure 1 [BZL+22], in end-to-end approaches all software modules are replaced by data-driven techniques, such as Deep Neural Networks (DNNs). In these approaches the final actuator output (e.g. steering, throttle, brake) is predicted directly by the data-driven techniques. This project will focus on such end-to-end control approaches.

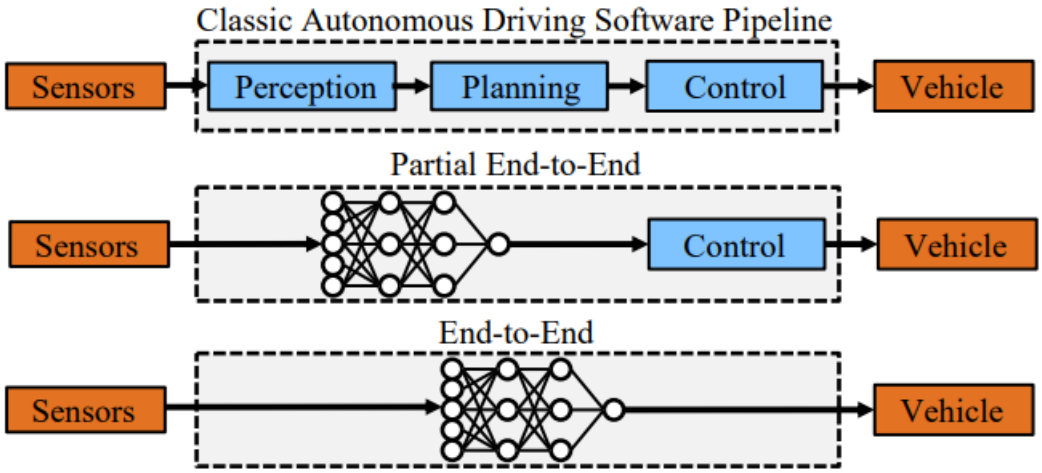

Figure 1: end-to-end, partial end-to-end and classical approaches diagram [BZL+22]

On the other hand, in partial end-to-end approaches only a portion of software modules are replaced by such techniques. A DNN usually provides the trajectory for the ego vehicle, which is then used by classic control techniques, such as a PID controller.

In classical approaches, all software modules are implemented using traditional techniques, described previously.

Autonomous racing is a good testing ground for end-to-end approaches, since it has clear driveable area, less signage than usual driving, only one class of objects and a clear training goal – optimal lap time.

However, large amounts of data are needed for training the DNNs when using end-to-end systems. A large variety of situations needs to be covered in these datasets in order to achieve good performance and therefore generalisation is a common issue, similar to other applications of DNNs. Other known end-to-end issues within the autonomous car racing field include computational requirements, difficulty learning vehicle dynamics (especially non-linear dynamics, such as tyre performance), simulation-to-reality gap and performance during out of distribution events [BZL+22].

2.1.1 Reinforcement learning

Reinforcement learning (RL) is a type of unsupervised learning and consists of training an agent to perform suitable action in order to maximise reward in a particular situation [GFG20]. Figure 2 [Bha20] shows a basic reinforcement learning diagram. For every discrete timestep t, the agent considers its environment and chooses a suitable action A_t according to its policy or Q values. The agent then gets a certain reward R_t based on the action that was taken and the environment transitions into the next state S_{t+1}. Recently this area of machine learning has experienced dramatic growth in interest due to its promising results in many areas like robotics, vehicle control, finance, natural language processing, healthcare and video games.

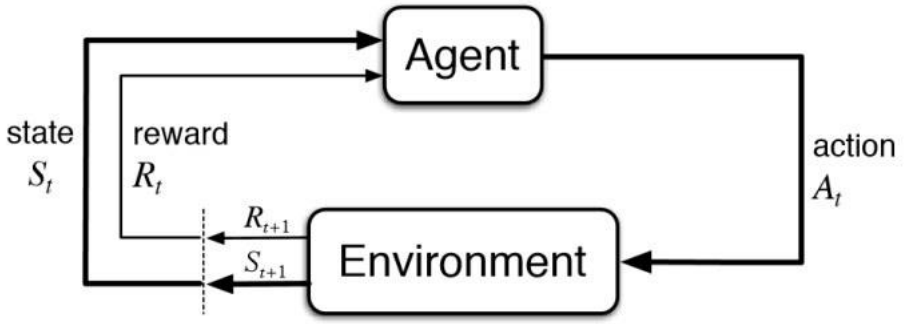

Figure 2: Basic reinforcement learning model diagram [Bha20]

2.1.2 Genetic algorithm

Genetic algorithm (GA) is an optimisation technique inspired by Charles Darwin's theory of natural evolution. It reflects the process of natural selection, where the fittest individuals are chosen for reproduction to create next generation's offspring [Mal17]. Such algorithms start with randomly generated solutions. After observing the performances of these solutions, most successful solutions are chosen for reproducing new solutions, evolving towards the optimal solution as iterations pass. Random mutations can be applied in order to explore new optimal solutions. Genetic algorithms can be used in a wide variety of applications and they search parallel from a population of points, so they can avoid the possibility of getting trapped on a local optimal solution like traditional methods, which search from a single point. Figure 3 [Alz17] shows a basic flow chart for a genetic algorithm.

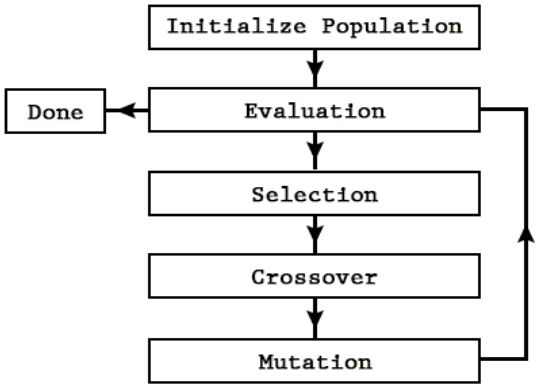

Figure 3: Genetic algorithm flow chart [Alz17]

2.1.3 Neural networks

Neural networks (NNs) are particularly useful for partially observable cases, where the action space or the state space is very large. Autonomous car control is one of such cases. Figure 4 [BGD17] shows a diagram for an example of a neural network architecture. The nodes within a NN have biases and are interconnected with weighted links. Based on these weights and biases, certain nodes are activated and an output is produced. Given enough training, NNs can find links between inputs and outputs, serving as excellent function approximators.

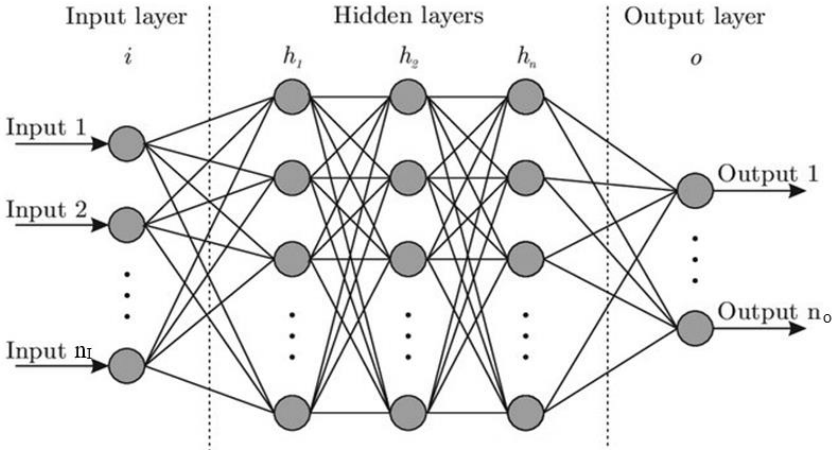

Figure 4: Artificial neural network architecture [BGD17]

2.2 Algorithms

This section will introduce the algorithms that will be implemented, analysed and compared in this project.

2.2.1 NEAT

The NEAT (NeuroEvolution of Augmenting Topologies) algorithm is a genetic algorithm (GA) designed for evolving artificial neural networks (ANNs) with varying topologies, making it well-suited for complex tasks like autonomous racing. NEAT was introduced by Kenneth O. Stanley and Risto Miikkulainen in their 2002 paper titled "Evolving Neural Networks Through Augmenting Topologies" [SM02].

Unlike conventional neural network evolution methods, NEAT begins with minimal, simple networks and progressively enhances them through generations, introducing new nodes and connections if needed. NEAT's encoding scheme tracks the historical development of neural networks, simplifying the evolution of complex topologies. This algorithm has proven particularly effective in domains that require adaptive and dynamic solutions, making it well suited for tasks such as autonomous car racing.

Because of the ability to evolve both the architecture and weights of neural networks simultaneously, NEAT (NeuroEvolution of Augmenting Topologies) is particularly well-suited for autonomous driving tasks. This allows the system to adaptively discover the right level of sophistication required for driving behaviours such as following the track and avoiding obstacles.

NEAT’s use of speciation is also important in complex environments like autonomous driving, where useful behaviours may require intermediate structures that don’t yield immediate fitness gains. By preserving diversity, NEAT avoids premature convergence and enables better exploration of the solution space.

Its track record in control tasks like simulated driving and robotic locomotion further reinforces its applicability. For these reasons, NEAT offers clear advantages over traditional genetic algorithms in autonomous driving scenarios where adaptability and efficiency are critical.

An overview of the main features of the NEAT algorithm in the context of autonomous car racing is given below.

Initialization

A population is started consisting of randomly generated neural networks with minimal structures, usually having just input and output nodes. Each neural network is assigned a unique genome, which encodes the network's structure and connection weights, which at the start are random.

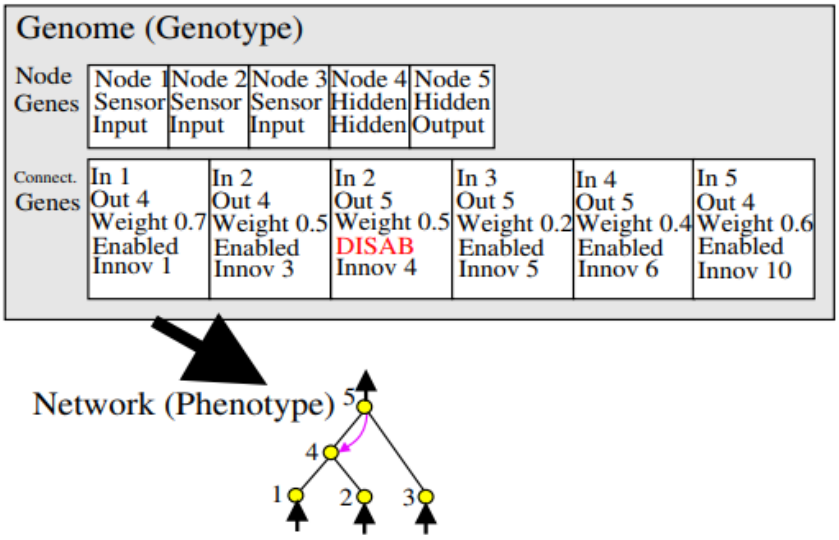

Figure 5: A genome mapping example [SM02]

Each genome includes a list of connection genes, each of which refers to two node genes being connected. Each connection gene specifies the in-node, the out-node, the weight of the connection, whether or not the connection gene is enabled, and an innovation number, which will be explained below. Figure 5 shows an example of a genome and the network that it represents.

Evaluation

Each neural network in the population is evaluated by running it in a simulated racing environment. The car's performance, such as lap time or ability to navigate the track, is used as the fitness score. Fitness is a crucial metric in the algorithm because it determines which networks are more likely to reproduce and pass on their genes to the next generation.

Reproduction

Networks with higher fitness scores have a higher chance of reproducing. NEAT employs the concept of "survival of the fittest" to select parents for reproduction. Crossover and mutation are applied to the genomes in the population. After reproduction, a new generation of networks is created, replacing the previous generation. The process repeats, with networks continually evolving over multiple generations.

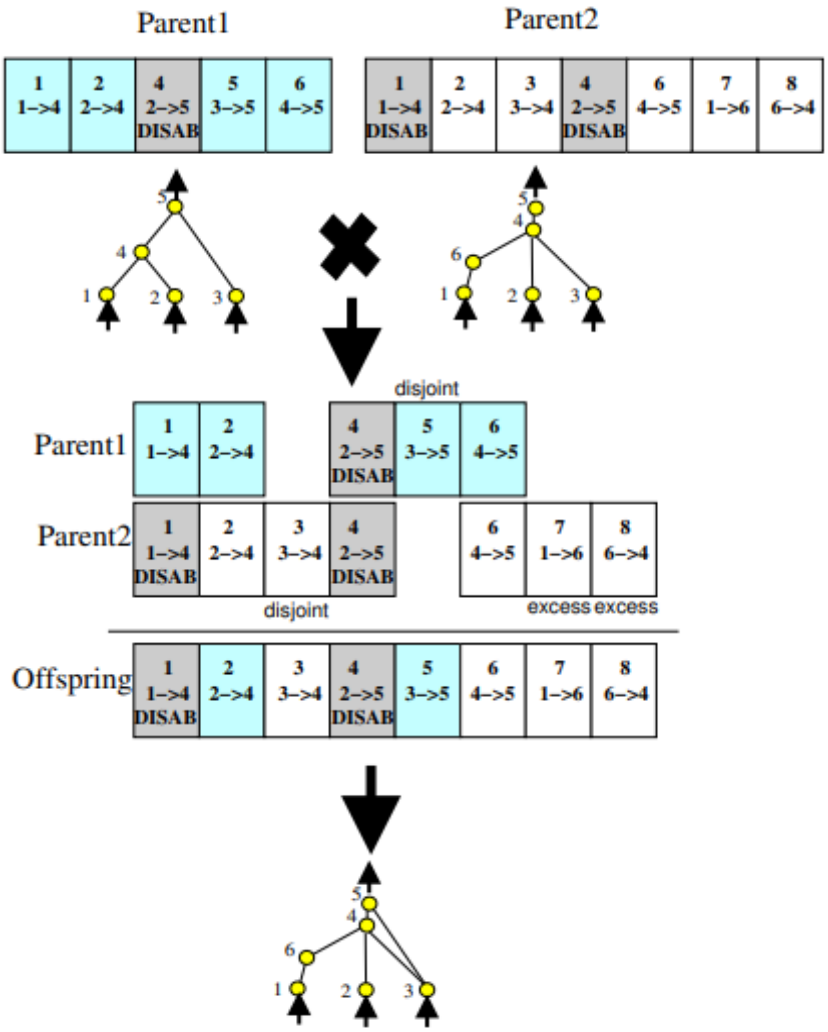

Figure 6: An example of crossover between genomes [SM02]

Crossover: Pairs of parent genomes are combined to create offspring. This includes inheriting connection genes, which represent neural connections, and node genes, which represent neurons. Figure 6 shows an example of how two parents would be matched up. The top number in each genome is the innovation number of that gene, which identifies the original historical ancestor of each gene, making it possible to find matching genes during crossover. New genes are assigned new

increasingly higher numbers. When crossing over, the genes in both genomes with the same innovation numbers are lined up. Genes that do not match are inherited from the more fit parent, or if they are equally fit, from both parents randomly. This way, historical markings allow NEAT to perform crossover without the need for expensive topological analysis.

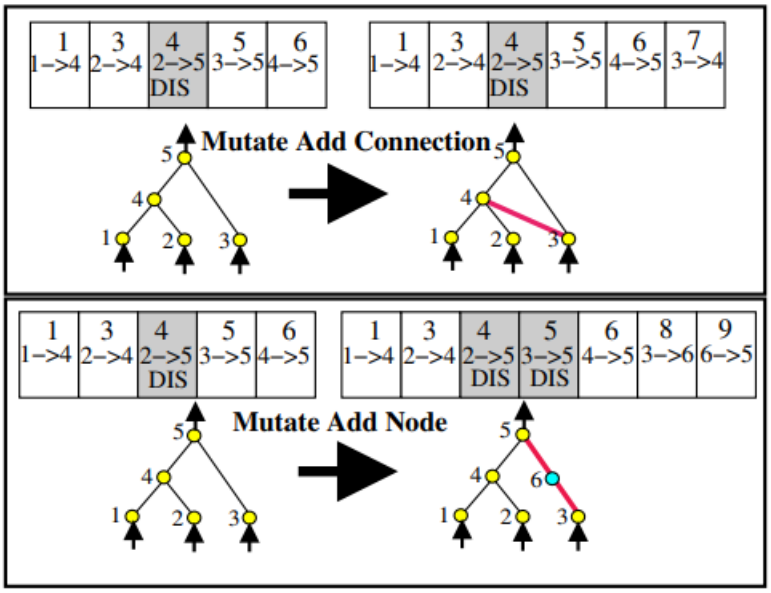

Figure 7: Examples of added connection and added node mutations [SM02]

Mutation: Some offspring undergo random mutations, which include adding or removing nodes or connections and changing connection weights. Figure 7 shows examples of mutations involving new connections and new nodes.

Speciation

NEAT uses a method called speciation to maintain diversity within the population. Networks are grouped into species based on genetic similarity. Fitness scores are adjusted to encourage cooperation within species, which prevents dominant species from erasing others prematurely.

A flow diagram for NEAT is shown in Figure 8.

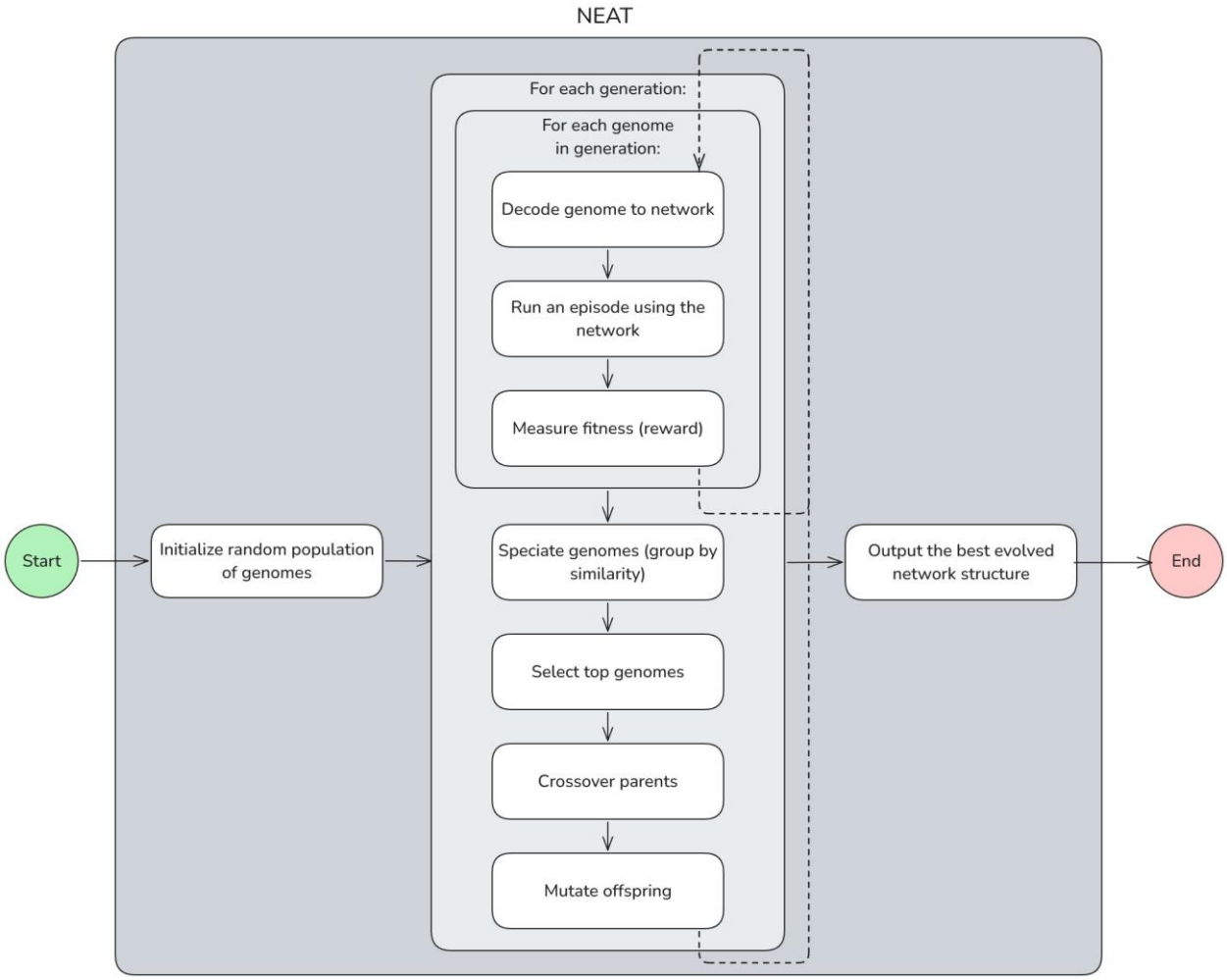

Figure 8: NEAT flow diagram

2.2.2 DDPG

The Deep Deterministic Policy Gradient (DDPG) algorithm [LHP+19] is a relatively recent advancement in the area of reinforcement learning (RL), specifically designed for addressing continuous action spaces. It is an off-policy, model-free strategy that extends the ideas from Deep Q-Networks (DQN) and Deterministic Policy Gradient (DPG) to work with real-valued actions by combining the strengths of actor-critic architectures and deep neural networks. The algorithm uses two separate networks: an actor, which maps states to specific actions, and a critic, which evaluates the quality (Q-value) of those actions given the state.

During training, DDPG relies on a replay buffer to store past transitions and samples mini-batches from this buffer to update its networks. This off-policy learning approach improves data efficiency and helps with correlations between sequential data. Also, DDPG uses previously mentioned target networks. These are slowly updated copies of the actor and critic, which help stabilize learning by reducing the risk of divergence due to quickly changing policies.

The algorithm uses a neural network-based actor to approximate the policy and a critic network to estimate the value function, as shown in Figure 9. Also, DDPG introduces the concept of target networks, which helps stabilize learning by decoupling the target estimation process from the current policy and value networks and a replay buffer mechanism to improve sample efficiency.

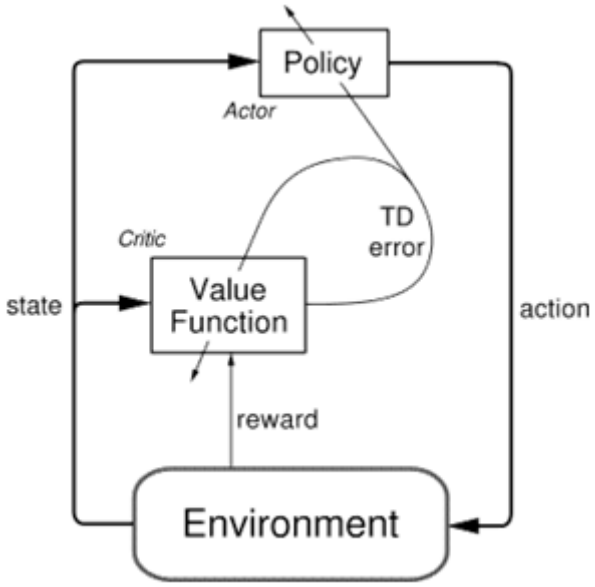

Figure 9: A diagram, showing the relationship between Actor and Critic [Lau16]

Main hyperparameters of DDPG are the discount factor (γ), exploration noise (σ), learning rates (α for actor and β for critic), and batch size.

Predicted Q-values for the next states are computed by the target critic network using the Bellman equation. It estimates what the expected return (Q-value) will be if the agent follows the target policy from the next state:

$$\text{target\_Q} = \text{reward} + \gamma \times \text{critic\_target}(\text{next\_state}, \text{actor\_target}(\text{next\_state}))$$

Where:

- reward: The immediate reward received after taking an action in the current state.
- γ (gamma): The discount factor (between 0 and 1), which determines the importance of future rewards.
- critic_target: The target critic network estimates the value of the next state-action pair.
- actor_target: The target actor network provides the next action to be evaluated by the target critic.

The actor's loss is computed using the deterministic policy gradient. It defines the objective for training the actor:

$$\text{actor\_loss} = - \text{critic}(\text{state}, \text{actor}(\text{state}))$$

Where:

- actor(state): The actor network outputs the action to take in a given state.
- critic(state, actor(state)): The critic evaluates how good that action is in the current state.
- The negative sign means we want to maximize the Q-value output by the critic. (We minimize loss in optimization, so we take the negative.)

A flow diagram for DDPG is shown in Figure 10.

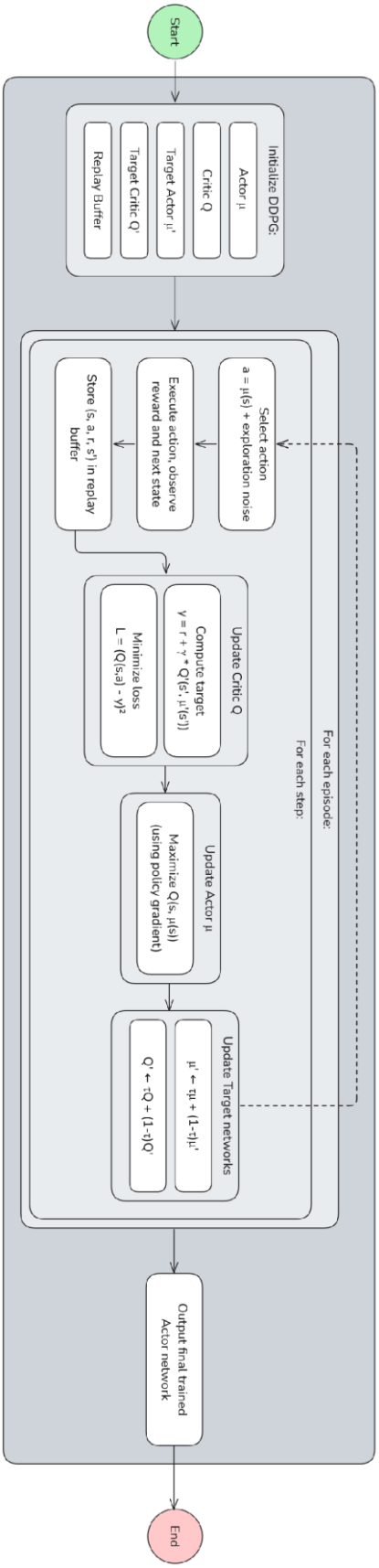

Figure 10: DDPG flow diagram

DDPG is a strong candidate for autonomous driving tasks because it performs well in continuous action spaces, which are essential for real-world control problems like steering, throttle, and braking. Unlike discrete-action algorithms like DQN, DDPG outputs smooth, real-valued actions. This makes it a natural fit for driving where fine-grained control is crucial. Its actor-critic architecture also helps by allowing more sample-efficient learning compared to purely evolutionary or policy-gradient methods.

Another advantage of DDPG is that it's an off-policy algorithm, which means that it learns from past experiences stored in a replay buffer instead of requiring fresh data at every step. This improves data efficiency and makes it possible to train using simulated driving environments without needing real-time interaction.

DDPG also utilizes target networks and soft updates, which help stabilize training and prevent issues like Q-value overestimation - a common problem of earlier actor-critic approaches.

Compared to other RL algorithms like PPO or A3C, which work well in high-dimensional discrete environments, DDPG is better suited for low-latency, continuous control tasks that require real-time responsiveness. This makes it a good choice for autonomous driving. Its combination of control precision, sample efficiency and training stability makes it a preferred choice for applications involving complex vehicle dynamics.

2.2.3 Ornstein-Uhlenbeck process

The Ornstein-Uhlenbeck (OU) process is a stochastic process often used in reinforcement learning to model and add noise to actions taken by an agent. The process helps strike a balance between exploration and exploitation, making the learning process more stable and efficient.

In the context of an autonomous racing car application, the Ornstein-Uhlenbeck process can be used to add controlled, smooth, and temporally correlated noise to the car's control inputs (e.g., throttle, brake and steering) to encourage exploration and improve learning stability.

The OU process can be described by the following differential equation:

$$dx(t) = \theta(\mu - x(t))dt + \sigma dW(t)$$

Where:

- x(t) is the value of the process at time t,
- θ is the mean-reversion rate, controlling how quickly the process reverts to the mean μ,
- μ is the target mean value,
- σ is the volatility, controlling the amplitude of the noise,
- dW(t) represents a Wiener process or Brownian motion, which is a random increment at each time step.

2.2.4 ERL

A hybrid algorithm consisting of both, NEAT and DDPG, was also implemented. It is a relatively unexplored idea that has only a few documented applications, none of which are for autonomous driving. It will be referred to as ERL (Evolutionary Reinforcement Learning) for the remainder of this report. The algorithm first uses NEAT to evolve a well performing neural network topology. The evolved neural network architecture is then plugged into the DDPG algorithm as the Actor network and trained further.

NEAT adjusts the network structure dynamically, while DDPG optimizes the strategy through deep reinforcement learning, allowing the algorithm to adapt to complex decision-making environments well. The advantage of combining DDPG with NEAT is that NEAT optimizes the topology of the Actor network through an evolutionary algorithm, which makes the Actor network better adapt to the complex environment. DDPG then helps to better explore the state–action space, which might lead to higher policy quality and improve the overall performance.

Pseudo-code for ERL is given below:

1. Input: NEAT parameters and DDPG parameters
2. Initialize NEAT population with population_size individuals

3. Initialize experience replay buffer D with capacity N
4. Evolve NEAT population for num_generations iterations with crossover_rate and mutation_rate
5. Get best NEAT genome
6. Create actor network based on NEAT topology
7. Initialize DDPG networks (actor, critic, target actor and target critic)
8. for episode in 1 : num_episodes do
    a. Initialize episode
    b. for t in 1 : T do
        i. Select action at according to current policy $\pi(s_t)$ + OU noise
        ii. Execute action a_t, observe reward r_t and next state s_{t+1}
        iii. Store transition (s_t, a_t, r_t, s_{t+1}) in D
        iv. Sample mini-batch of transitions from D
        v. Update critic by minimizing loss
        vi. Update actor policy using the sampled policy gradient
        vii. Update target networks
    c. end for
9. end for

A flow diagram for ERL is shown in Figure 11.

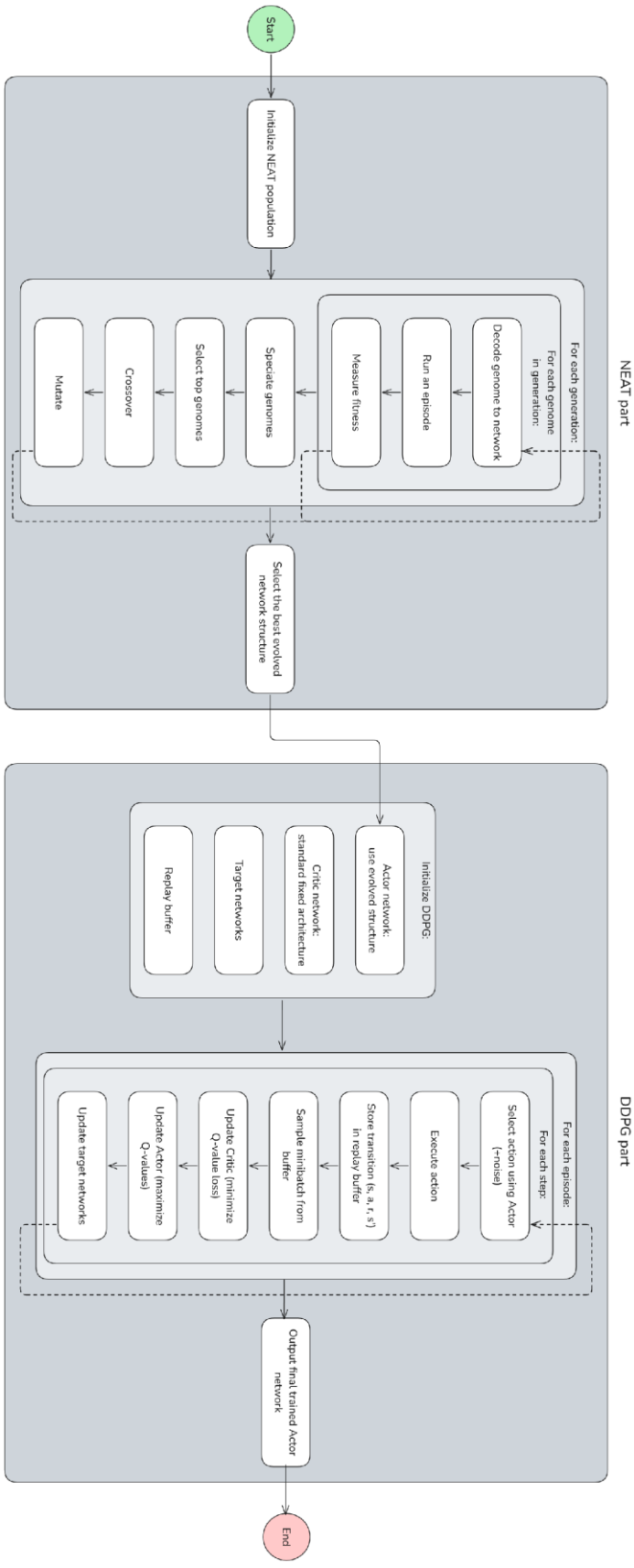

Figure 11: ERL flow diagram

2.3 Relevant Experiments

This section surveys the experiments documented in available literature that are relevant to the project.

A study on autonomous car racing published in 2022 [BZL+22] gives a good overview of the current state of the art in the field. Table 1 [BZL+22] below lists some research that has been carried out in the field of end-to-end control approaches.

Table 1: Research overview in the field of end-to-end approaches for autonomous racing [BZL+22]

Name	Year	End-to-End Category	Topic	Method	Tested on Hardware	Racing Series
Perez et al.	2008	Optimization	Optimal Control Policy	Evolutionary Algorithm	No	-
Salem et al.	2017, 2018	Optimization	Optimal Control Policy	Fuzzy Logic	No	-
Korkmaz et al.	2018	Optimization	Optimal Control Policy	Fuzzy Logic	No	-
Oliveira et al.	2018	Optimization	Vision based Planning	Bayesian Optimization	No	-
Lee et al.	2019	Deep Learning	Vision based Planning	MPC + CNN	Yes	AutoRally
Weiss et al.	2020	Deep Learning	Vision based Planning	CNN, RNN	No	-
Tatulea et al.	2020	Deep Learning	trajectory planning	NMPC & DNN	No	F1TENTH
Weiss et al.	2021	Deep Learning	Trajectory Prediction	RNN	No	-
Drews et al.	2019	Deep Learning	Localization	CNN, LSTM, + MPC	Yes	AutoRally
Mahmoud et al.	2020	Deep Learning	Vision based Planning	CNN, LSTM	Yes	Donkey Car
Wadeka et al.	2021	Deep Learning	Vision based Planning	CNN	No	IAC
Perot et al.	2017	Reinforcement Learning	Vision based Planning	Advantage actor-critic	No	-
Jaritz et al.	2018	Reinforcement Learning	Vision based Planning	Advantage actor-critic	No	-
De Bruin et al.	2018	Reinforcement Learning	Vision based Planning	Q-Learning+ State representation Learning	No	-
Remonda et al.	2019	Reinforcement Learning	Vision based Planning	DDPG	No	-
Niu et al.	2020	Reinforcement Learning	Vision based Planning	DDPG	No	-
Gückiran et al.	2019	Reinforcement Learning	Vision based Planning	SAC, Rainbow DQN	No	-
Fuchs et al.	2021	Reinforcement Learning	Vision based Planning	SAC	No	-
Chisari et al.	2021	Reinforcement Learning	Vision based Planning	SAC + policy output regularization	Yes	1:43 car
Lee et al.	2021	Reinforcement Learning	Vision based Planning	Bayesian Deciscion Making	Yes	AutoRally
Pan et al.	2021	Reinforcement Learning	Vision based Planning	Imitation Learning	Yes	AutoRally
Cai et al.	2021	Reinforcement Learning	Vision based Planning	Imitation Learning	Yes	1:20 car
Schwarting et al.	2021	Reinforcement Learning	Vision based Planning	Model based RL	No	-
Brunnbauer et al.	2021	Reinforcement Learning	Vision based Planning	Model based RL	Yes	F1TENTH
Song et al.	2021	Reinforcement Learning	Vision based Planning + Overtaking	SAC + 3-stage curriculum learning	No	-
Gundu et al.	2019	Reinforcement Learning	Model-free optimal control	Q Learning + Soft-Actor Critic	No	-
Ivanov et al.	2020	Reinforcement Learning	Verification	Variation of Algorithms	Yes	F1TENTH

2.3.1 Actor-critic

Actor-critic algorithms are a family of reinforcement learning algorithms that combine aspects of both, policy-based methods (Actor) and value-based methods (Critic). Examples of Actor-critic algorithms include DDPG, Advantage Actor-Critic and Soft Actor-Critic.

The authors of [JCT+17] and [JCT+18] have both successfully applied the Advantage Actor-critic algorithm to a rally simulator to achieve end-to-end driving. The obtained results were satisfactory

and the vehicle was able to manoeuvre rapidly on differing road conditions. However, it failed to generalise well in other situations.

The Soft Actor-critic (SAC) algorithm was successfully implemented for an autonomous racing application in papers [CLR+21], [GB19], [SKD+21] and [SKL+21]. In [CLR+21], the SAC method was applied to a 1:43 scale vehicle. The performance of SAC was compared to a Model Predictive Control (MPC) planner, MPC outperformed SAC in that case.

2.3.2 DDPG

The first convincing showcase of DDPG for the autonomous driving use case came from Wang, Jia and Weng in 2019 [WJW19]. Working in TORCS, they fed the policy nothing but 29 low-level sensor channels and let it train for roughly two hundred episodes. By the hundredth episode the controller was already lapping the Aalborg circuit without crashes. By the end it learnt to brake before S-curves to avoid drift and could overtake all nine scripted opponents on corner exit. Training logs revealed that average per-step reward stabilised after the first hundred laps, however lane-centering kept improving for another fifty. It shows evidence that DDPG's actor-critic architecture can keep improving performance even after the exploration noise decays.

DDPG was also implemented using the TORCS simulation package in papers [RSV+19] and [NHJ+20]. The authors in both cases have enhanced DDPG to make it better suited for the racing environment. Experiments showed good learning results and performance.

Taken together, these papers show that DDPG is a well-established algorithm within the autonomous driving field and is capable of high performance.

2.3.3 Genetic algorithm

In 1996, a case study published by Pyeatt et al. [PHA+96] experimented with autonomous driving using the RARS simulation software (later used as the base for TORCS). The work applied neural networks to implement an evolutionary system in which individuals are composed of a set of rules,

linking sensor data to control actions like throttle or steering input. Even back then, results showed that evolutionary systems have potential as competitive approaches to autonomous car racing.

The authors of [PSR+08] have implemented a similar GA based system using a later version of the TORCS simulator. In two out of three tested circuits the agent obtained acceptable results. However, on the third track the car would lose control due to an unexpected zigzagging effect combined with banked turns present in that track.

Papers published in 2005 and 2006, [TJS+05] and [TJS+06], proposed another automated evolutionary design for driving agents. GAs were used to design an agent, which operates a scaled down remote control car. In this study, the environment was perceived using the input from an overhead mounted camera. Inputs of the system included position, orientation, velocity, approach angle, distance to the apex and outside/inside slow down zone. These perceptions were used to control the throttle, brake and steering of the car. Comparative analysis showed that during long runs the agent was 5% slower, than a human counterpart.

The authors of [SVS+20] have implanted a GA for controlling a car within a simulated environment made using Unreal Engine 4 and Nvidia PhysX. Over several generations the population was found to evolve enough to avoid crashing into obstacles. During experiments it was also observed that given a relatively simple fitness function, the agent took few generations to start navigating the course acceptably. However, when trying to alter the function in order to make the ego car behave in a certain way (e.g. maintain a certain speed), the authors found it took significantly more generations. Trained agents also showed some advanced actions, such as counter-steering during an over-steer situation (rear end of the car losing grip) in order to regain traction.

2.3.4 NEAT

Early work by Cardamone, Loiacono and Lanzi [CLL+09] showed that classic NEAT could already produce race-worthy behaviour in a realistic simulator at that time. Using TORCS, the authors evolved two separate networks - one for time-trials and one for overtaking. Then these genomes were merged together. The composite controller consistently beat every hand-coded bot that had won the Simulated Car Racing Championship up to that point, confirming that NEAT's topology

search can discover trajectories and manoeuvres that human developers missed. The networks were bred on a suite of 24 tracks and therefore generalised well without any retuning required.

The same group expanded on the idea in “Learning to Drive in the Open Racing Car Simulator Using Online Neuroevolution” a year later [CLL+10]. Instead of evolving in large offline batches, they evaluated small slices of a lap in real time and replaced genomes on-the-fly. This online NEAT variant could overtake a strong offline-evolved baseline after only a few hundred evaluations. Fine-grained fitness updates shortened learning time by roughly a third compared with generation-level updates while preserving the final lap time.

All in all, literature shows that in autonomous racing sandboxes NEAT is still capable of delivering competitive, interpretable controllers with less data than deep RL. The method’s main challenges are high-dimensional perception and real-vehicle validation, both active topics that hybrid NEAT variants are beginning to tackle.

2.3.5 Hybrid algorithms

The idea of letting NEAT supply the network architecture while DDPG fine-tunes the weights is still young, however a couple of papers provide an early glimpse to what the combination can achieve.

The most relevant study so far comes from Wang et al. [WZM+24], who attached NEAT onto the actor of a DDPG agent and let evolution reshape layer counts and skip-connections every few training epochs. Although their application was compressed air energy storage scheduling rather than driving, the control problem is continuous and highly non-linear - much like operating a car on a racetrack. The hybrid “DDPG-NEAT” agent converged to a dispatch accuracy of about 92% - a 31% leap over vanilla DDPG and twice the sample-efficiency of SAC baseline. Their experiments make the benefit clear - once NEAT has removed redundant neurons, the gradients stabilise. Actor updates stop oscillating and the learning curve flattens out sooner and at a higher reward.

Back in GECCO 2017, Peng et al. created an earlier hybrid that paired NEAT with a simpler regular gradient actor-critic loop [PCH+17]. On classic benchmarks the method evolved feature-extractor topologies while the critic updated weights at each episode. On Cart-Pole it balanced twice as long as pure NEAT after 100 generations. Although their critic was linear rather than deep, the paper is often cited as a proof that evolutionary topology search and gradient policy improvement can

coexist without destabilising one another - a design pattern, which later authors swap into DDPG with minimal changes.

These early steps suggest that NEAT's structural search can give DDPG a head-start in hard, continuous-control settings. The approach shows potential to reduce time spent on manual architecture tuning and results in smoother the actor-critic learning dynamics.

3. Methodology

3.1 TORCS

The Open Racing Car Simulator (TORCS) is a driving simulation program. It has relatively realistic physics simulation, considering aspects like weight, traction, and aerodynamics for lifelike driving dynamics. Its customization options allow users to choose vehicles and tracks while tailoring racing parameters. TORCS also serves as a versatile platform for research and development due to its open-source nature and plugin support, making it a popular choice in areas like autonomous driving and computational intelligence.

Experiments were run inside an Ubuntu virtual machine, with assigned 4 virtual CPUs and 8GB of RAM.

3.2 SCR plug-in

Out of the box, TORCS has some drawbacks for the autonomous control use case: races lack real-time due to blocking bot execution, differing access to track data leads to varied driving strategies, and language choices are constrained to C/C++. The SCR plugin competition software improves TORCS by configuring it as a client-server system, achieving real-time performance with UDP-connected external bot processes. An abstraction layer separates driver code and the server, accommodating various programming languages for bots while limiting data access. The software architecture introduces a new "scr server" for UDP connections, serving as an intermediary between the game and client bots, as shown in Figure 12. During races, server-bots supply sensory data to

clients and await actions, with the server updating the race state and enabling race restarts through specialised actions. The server updates the race state every game tick (20ms of simulated time). SCR clients written in Python will be used in this study, seeing as the algorithms were also be implemented in Python.

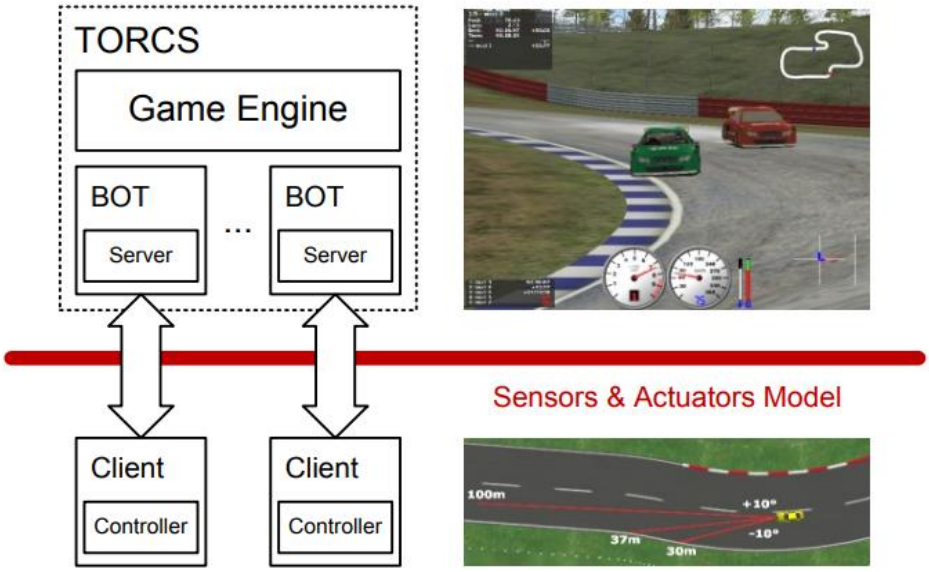

Figure 12: The architecture of SCR plug-in software [LCL13]

3.3 NEAT

The NEAT algorithm, described in Section 2.2.1, has been picked as the GA of choice for the experiments. It is expected to have a high efficiency when compared to other genetic algorithms due to incrementally growing from a minimal structure and employing a principled method of crossover for different topologies. A Python library called neat-python has been used to implement the algorithm for the TORCS use case [Cod19]. Due to performance limitations of the computer that the experiments were run on, the algorithm was configured to run only one car (genome) at a time. Given a more powerful machine, all genomes in a population could be run simultaneously, which would drastically reduce the training time for this algorithm.

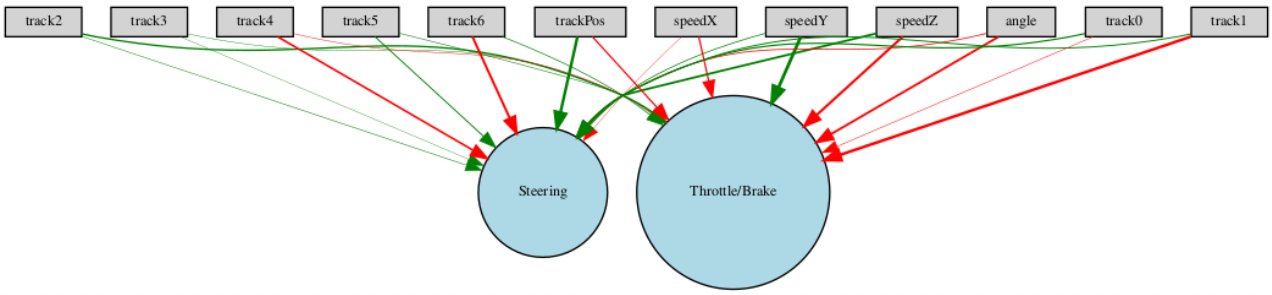

Figure 13: A starting genome's network structure

Figure 13 shows an example of a genome from the initial population for NEAT. It has 12 input nodes, represented by grey boxes, and 2 output nodes, represented by blue circles. These output nodes are then mapped to car control actions in TORCS. Each input node starts having connections with random weights to each of the output nodes.

After some initial testing, it was decided to set the population size to 30 genomes. It was decided to compare the algorithms after running each one for 2500 episodes, therefore the number of generations was set to 83.

The full NEAT configuration can be found in Appendix A.

3.4 DDPG

The DDPG algorithm, described in Section 1.2, has been implemented using the TensorFlow framework for python [Ten25].

DDPG was configured to make its results more comparable to those of the NEAT algorithm. The same sensor inputs were used, as well as the same output nodes and their activation functions. Also, the reward function for DDPG was configured to match the fitness function of NEAT.

For both – the Actor network and the Critic network, 2 hidden layers were used between input and output layers. First hidden layer was set to have 150 nodes, while the second – 300. Both hidden layers used the rectified linear unit (relu) activation functions. A batch size of 64 was used. Discount factor γ was set to 0.99, actor learning rate α was set to 0.0001 and critic learning rate β was set to 0.001.

The Ornstein-Uhlenbeck process was used for the exploration. It was set to run for 200,000 steps with gradually reducing noise addition. Its parameters were set as following – μ of 0.0, θ of 0.6 and σ of 0.3. The same parameters were used for both – steering and throttle/braking noise.

DDPG was trained for a total of 2500 episodes. The full DDPG configuration can be found in Appendix B.

3.5 ERL

3.5.1 NEAT part

For the NEAT part of ERL, slightly different settings were used than during the other NEAT experiments. This was done to encourage the algorithm to favour creation and deletion of new nodes or connections in order to explore a wide range of network structures.

Since the quality of the evolved architectures will heavily influence DDPG's downstream performance, the priority was:

- Maximizing architectural diversity early on
- Encouraging exploration of larger, more expressive structures
- Avoiding premature convergence or stagnation

The full NEAT configuration that was used for ERL can be found in Appendix C.

3.5.2 Genome to TensorFlow model conversion

The neural network evolved by NEAT comes in the form of a genome – an object that is part of the python-neat library [Cod19]. To use this network in DDPG, the genome had to be converted to a TensorFlow model.

This comes with some challenges, as the type of network that is evolved by NEAT is not directly compatible with TensorFlow. Specifically, NEAT does not stick to the usual fully-connected feed-

forward architecture. It allows connections between nodes from all layers, not just neighbouring ones.

Therefore, a rule was introduced when converting the genome to a TensorFlow model – for any nodes, which have input connections with nodes that are 2 or more layers away, proxy nodes are created in the skipped layers. These proxy nodes only have connections with the original input and output nodes. This way, the genome can be converted to a TensorFlow model while keeping as much of the original genome structure as possible.

3.5.3 DDPG part

After evolving a NEAT genome and converting it into a TensorFlow model, this model is then plugged in into the DDPG algorithm as the actor network. The rest of the training happens as described in section 2.4.

For ERL, NEAT was trained for 30 generations with 30 genomes in each, adding up to 900 episodes. DDPG was trained for a further 1600 episodes. This way, the total number of episodes for ERL adds up to 2500 – the same, as in prior NEAT and DDPG experiments. Therefore, a side-to-side comparison can be made.

3.6 Neural network inputs

Table 2 below shows the sensors that were used as inputs for the neural networks in the experiments.

Table 2: Sensor inputs used for the experiments [LCL13]

Name	**Range (units)**	**Description**
angle	[-π, +π] (rad)	Angle between the car direction and the direction of the track axis.
track	[0,200] (m)	Vector of 7 range finder sensors: each sensor returns the distance between the track edge and the car within a range of

		200 meters. The sensors sample the space in front of the car every 30 degrees, spanning clockwise from -90 degrees up to +90 degrees with respect to the car axis.
trackPos	(−∞,+∞)	Distance between the car and the track axis. The value is normalized w.r.t to the track width: it is 0 when car is on the axis, -1 when the car is on the right edge of the track and +1 when it is on the left edge of the car. Values greater than 1 or smaller than -1 mean that the car is outside of the track.
speedX	(−∞,+∞) (km/h)	Speed of the car along the longitudinal axis of the car.
speedY	(−∞,+∞) (km/h)	Speed of the car along the transverse axis of the car.
speedZ	(−∞,+∞) (km/h)	Speed of the car along the Z axis of the car.

3.7 Neural network outputs

The neural networks were set up to have 2 outputs – one for steering and one for accelerating or braking, depending on whether the output value is positive or negative.

Activation function for both of the output nodes in both of the algorithms was chosen to be the tangent hyperbolic function, also known as tanh – its graph is shown in Figure 17. It was chosen due to the fact that both of the outputs can represent 2 actions each – in the case of the steering node its either steering left or steering right, while in the case of the throttle / braking node its either accelerating or applying brakes. In both cases applying both actions at the same time (steering left and right or accelerating and braking) is undesirable. The tanh activation function suits this use case well, since its result can be mapped to one of the 2 possible actions depending on whether the result is positive or negative.

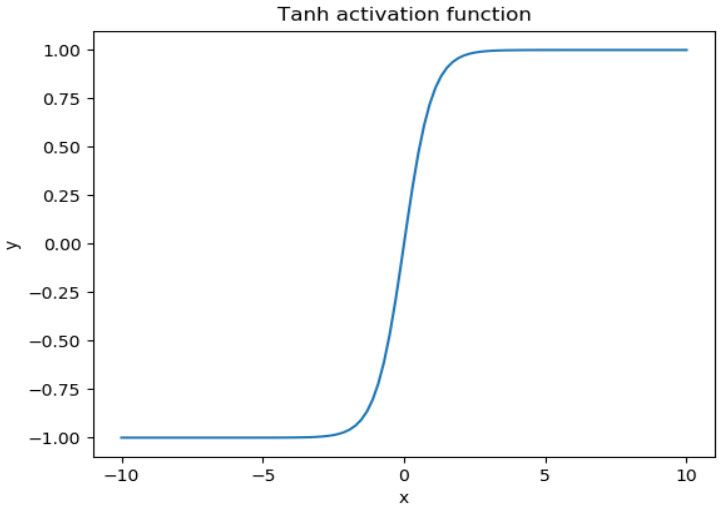

Figure 14: Graph of the tanh activation function [Ant23]

Table 3 below shows how the outputs from the neural networks were related to agent actions in TORCS. Both, NEAT and DDPG, can be used for continuous action spaces. For the use case of controlling a car that is more desirable, than discrete action spaces, therefore continuous controls have been implemented.

Table 3: TORCS actions used in the experiments

Name	**Range**	**Description**
Steering	[-1, +1]	A continuous control, with -1 representing steering at full lock to the right and +1 representing steering at full lock to the left.
Throttle / Brake	[-1, +1]	A continuous control, with -1 representing fully applied brakes and +1 representing full throttle.

3.8 Reward and fitness functions

The following equation has been used to calculate both, the fitness in NEAT and the reward in DDPG:

$$progress = (1 - penalty) \times v\_long$$

Where:

- v_long is the longitudinal velocity of the car: v_long = speed × cos(angle)
- penalty is the amount of fitness/ reward deducted due to unwanted behaviour

The following code has been used to calculate the penalty:

```
penalty = 0
# Out-of-track detection
if abs_trackPos > 1:
    if abs_trackPos > 2:
        abs_trackPos = 2
    penalty += out_of_track_coefficient * (abs_trackPos - 1)
# Collision detection
if collision_detected > 0:
    penalty += collision_coefficient
```

Where:

- abs_trackPos is the absolute value of trackPos sensor reading, described in Table 2.
- The out_of_track_coefficient was set to 5 and the collision_coefficient was set to 5, as well.

This way of constructing the penalty, paired with these coefficient values, was found to discourage the agent from driving off the track and coming in contact with the walls enough, while avoiding huge variations in the fitness/ reward results, which would negatively impact training.

3.9 Repository

Implementation details of all of the algorithms, as well as the settings used during training, can be found in the public *torcs-autonomous-racing* repository [Mil25a].

4. Experiments and Evaluation

4.1 Experimental setup

The study was organised as a set of three independent training runs for each learning method. Experiments were carried out in TORCS with an identical sensor-action interface and the same stochastic starting positions. Every run lasted 2500 episodes in total. During training, instantaneous reward returned by TORCS was logged. This was used to compute the maximum reward values over the last thirty episodes, as well as rolling average. These metrics were plotted in the learning-curve figure for analysis and comparison.

After training, to gauge real-world driving competence, agents were released for a single, no-reset lap on four tracks of ascending difficulty. Elapsed time and collision count were recorded. These metrics produced the lap-time table, which was also used to compare the performances of algorithms.

4.2 Episode termination conditions

In order to speed up the training process, the environment has been configured to terminate and reset if the agent either slowed down to a speed less than 5km/h or started driving backwards along the track. These conditions have been implemented in order to avoid spending too much time on unproductive episodes.

Also, episodes were configured to terminate if the agent has been successfully driving for 1000 steps (made 1000 actions during the episode). This way, once agents learn to drive the drack well, they do not run infinitely and superior policies can be picked more easily, since they are able to accumulate more fitness/ reward within those 1000 episodes.

4.3 Training and testing environments

The algorithms were first trained on a training track and then tested on different, previously unseen tracks. This was to test their generalisation capabilities and investigate whether agents were capable

of driving in a wide variety of scenarios. A good performance in the training track but disappointing performance in the testing tracks would indicate overfitting – that the agents simply "memorised" the track that they were trained on. The TRB1 race car, shown in Figure 15, was used in the experiments. It has rear wheel drive and the gearbox was set to automatic. No anti-lock braking system (ABS) or traction system was implemented.

Figure 15: Car used in the experiments

One of the tightest tracks in the simulator was chosen as the training track. This decision was made hoping that other tracks would be easier to complete after training the algorithms on the most demanding one. Table 4 shows details and layouts for the tracks used in the experiments.

Table 4: Race tracks used in the experiments

Track	**Type**	**Length**	**Width**	**Layout**
E-Track 2 *(Training)*	Road	3148m	12m	
Forza	Road	5748m	11m	
Alpine 1	Road	6356m	12m	

E-Track 5	Oval	1622m	20m	

5. Results and Analysis

5.1 NEAT

Figure 16 shows how the genome fitnesses changed as the NEAT algorithm evolved its populations. As mentioned in Section 3.3, there were 30 genomes in one generation. Therefore 30 episodes of training were being run each generation.

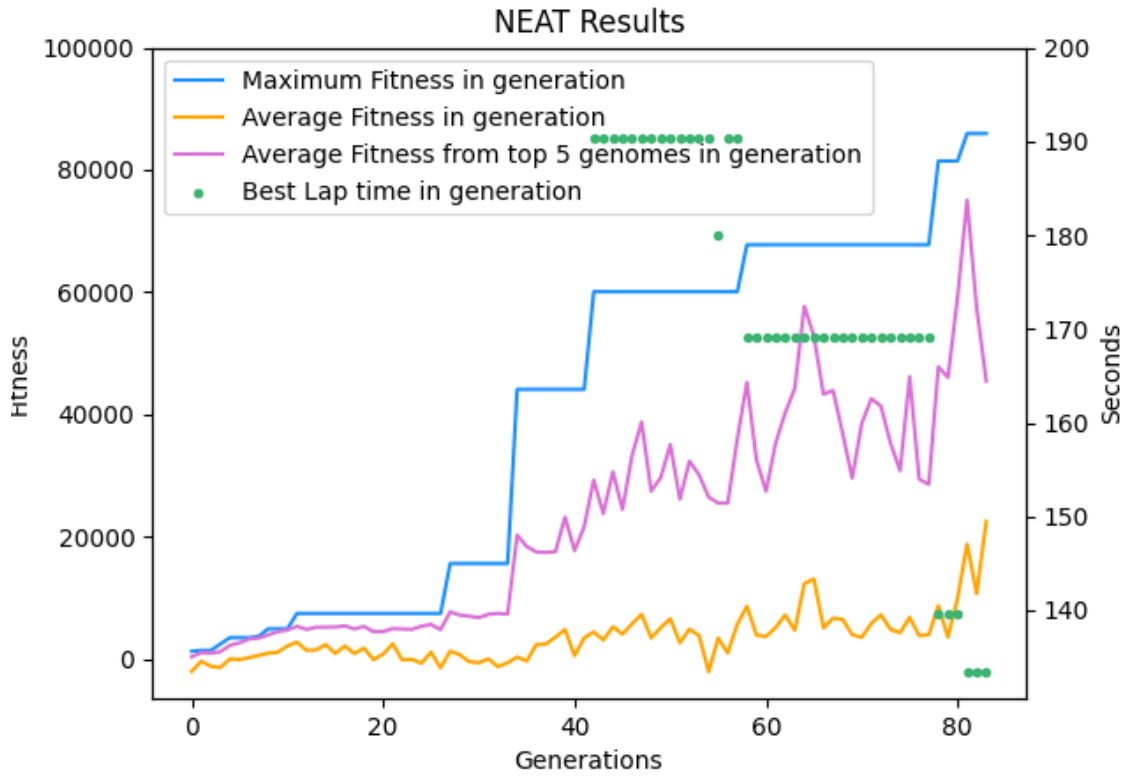

Figure 16: Result fitnesses and lap times from NEAT training (30 episodes per generation)

It is evident that increasingly better car control strategies were found as generations passed. At around 25th generation, the best genome of the population was able to successfully complete the first left-hand turn of the training track. This happened surprisingly soon, since the first turn is tight and comes after a straight, requiring the car to be slowed down before the turn. About 10 generations

later, the next turn, a right-hand one, was also successfully completed. Finally, around the 40th generation mark, first full laps started to get completed, as indicated by the green dots, which represent lap times. From this point onwards, genomes were able to improve the best gained fitnesses and cut down their lap times a few more times. At the end of training, the most suited genomes were achieving lap times of around 133 seconds.

Figure 17 shows the structure of the best genome after 100 generations of evolution using the NEAT algorithm. It is evident that the genome evolved the correct weights of connections between inputs and outputs needed for efficient driving.

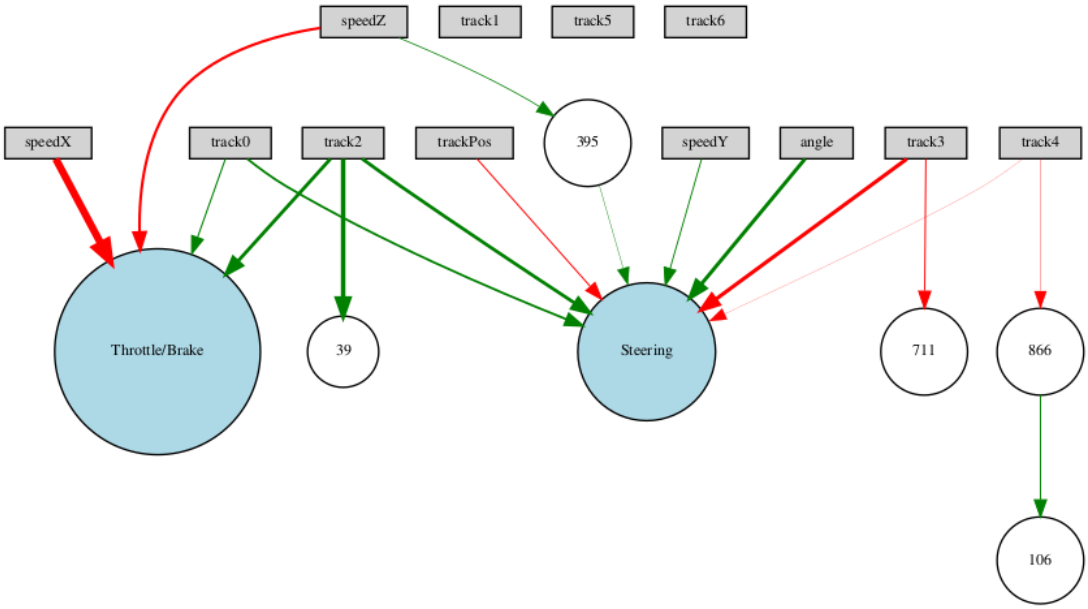

Figure 17: Best genome's network structure

The speedX node, for example, has a strongly inverse relationship with the throttle output, as indicated by the wide red arrow between these nodes. This makes sense, seeing as the more speed a car picks up, the more likely it is to brake in order to stay on the track.

The angle input node has a positive relationship with the steering output, as indicated by the green arrow. That is because a high angle value would mean the car is facing towards the right side of the track and would therefore need to turn left, which would mean applying a higher steering value, as explained in Table 3.

The trackPos node, which sends the car's traverse position along the track, has a negative relationship with the steering node. Again, this is expected, since a high trackPos reading would

mean that the agent is on the left side of the track and would therefore need to turn right, resulting in a negative steering value.

An interesting thing to note is that the final genome's structure appears to have evolved very few extra nodes and some of them don't even have connections to output nodes. This could happen due to the algorithm deciding that the task of driving the car in this simulator does not require a more complex network structure to achieve better performance. The large numbers assigned to nodes, such as 866, indicate that the algorithm tried adding many nodes to the structure during the training, however they provided little value to the agent's performance and were therefore removed. It is likely that the hidden nodes currently present in the final genome would also be removed given some further training.

5.2 DDPG

Figure 18 shows the results from DDPG experiments. The agents did not learn much until about the 1500^{th} episode, at which point one complete lap was successfully completed, as indicated by the green dot at the top of the graph. A couple hundreds of further episodes then passed seemingly without much progress. Then, at around 1800 episodes, weights were updated to sufficient values for consistently completing the training track for the rest of the training time. At around this time the OU process would have reached the set number of steps for applying noise, meaning that further training was done without exploration provided by OU. It was expected that the agent would be able to further fine-tune its policy and refine its behaviour but there is little evidence of this happening – the gained rewards and lap times did not experience much improvement after the mark of around 2000 episodes. Most completed lap times, excluding a few anomalous results, were similar at around 110 seconds.

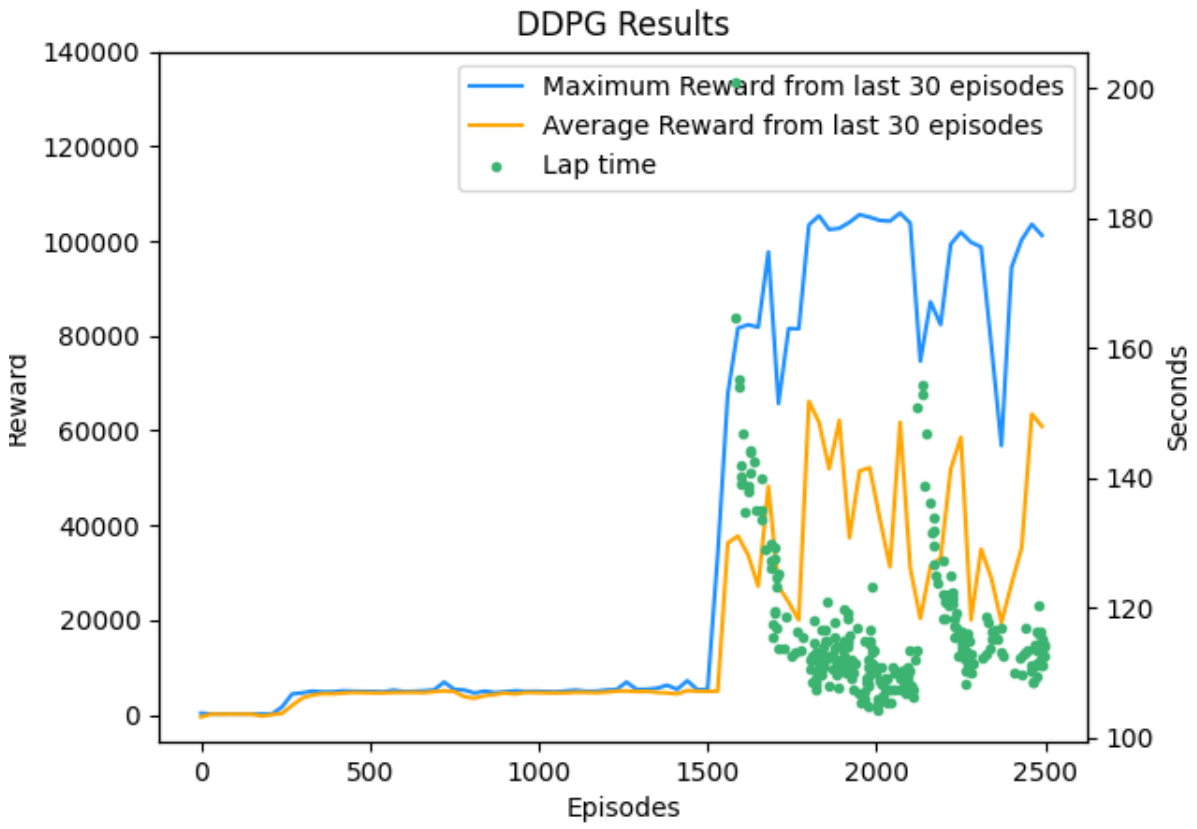

Figure 18: Result rewards and lap times from DDPG training

5.3 ERL

5.3.1 NEAT phase

Results from the NEAT part of ERL training is shown in Figure 19 below. The best genome went through 5 improvements and by the end of training was able to bake and stay on track during the first couple of corners. This provided confidence that the neural network structure has evolved enough to be further trained with DDPG.

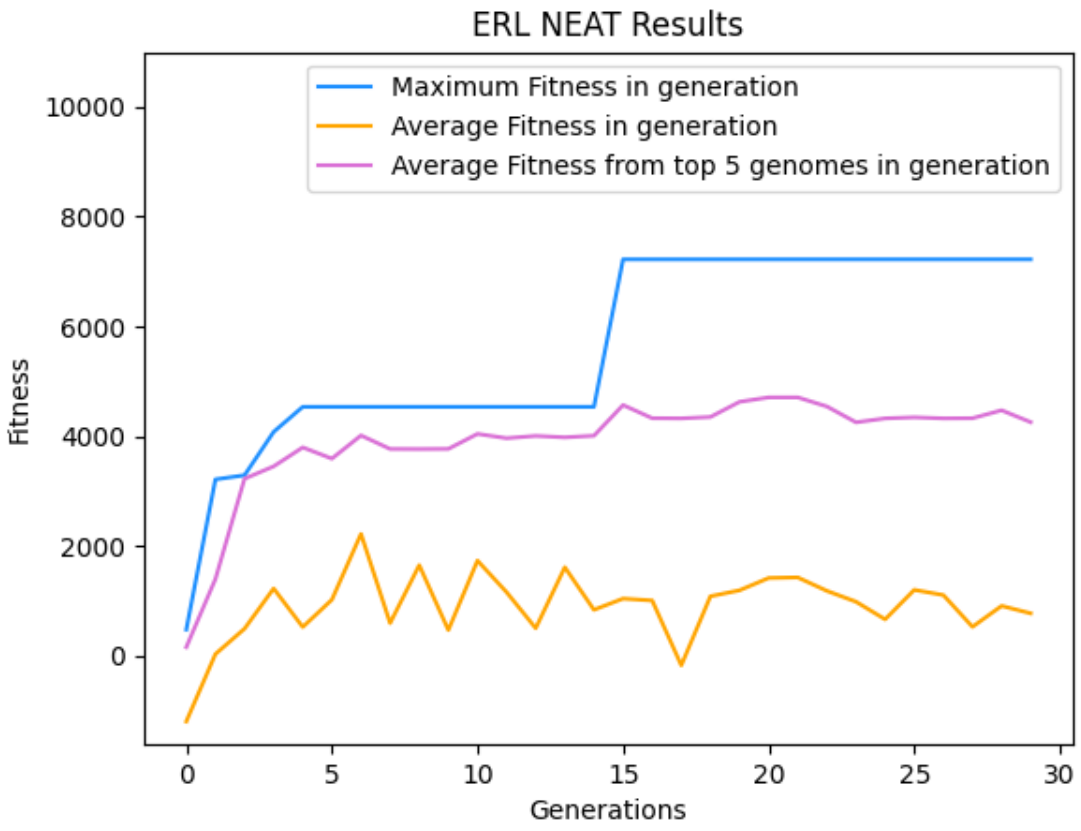

Figure 19: ERL NEAT Results

The structure of the best-performing genome at the end of ERL NEAT training is given in Figure 20:

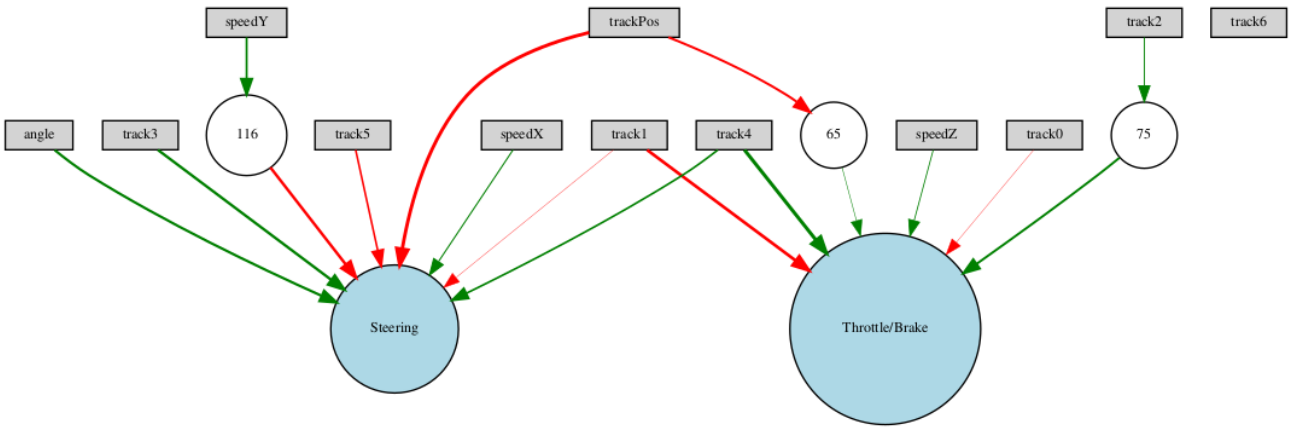

Figure 20: Genome structure at the end of ERL NEAT training

As we can see, all input nodes have direct connections to output nodes except for 3 – node 116 sits between speed Y and Steering, node 65 between trackPos and Throttle/Brake and 75 between track2 and Throttle/Brake. The converted genome after applying the rule described in Section 3.5.2 is shown in Figure 21. It has 1 hidden layer with 5 nodes. 2 of those 5 are proxy nodes – numbered as 1000 and 1001 in the diagram. It is much simpler than the baseline DDPG actor structure (2 hidden layers – 150 and 300 nodes). Simpler structure could allow faster training due to less trainable weights, however may limit performance.

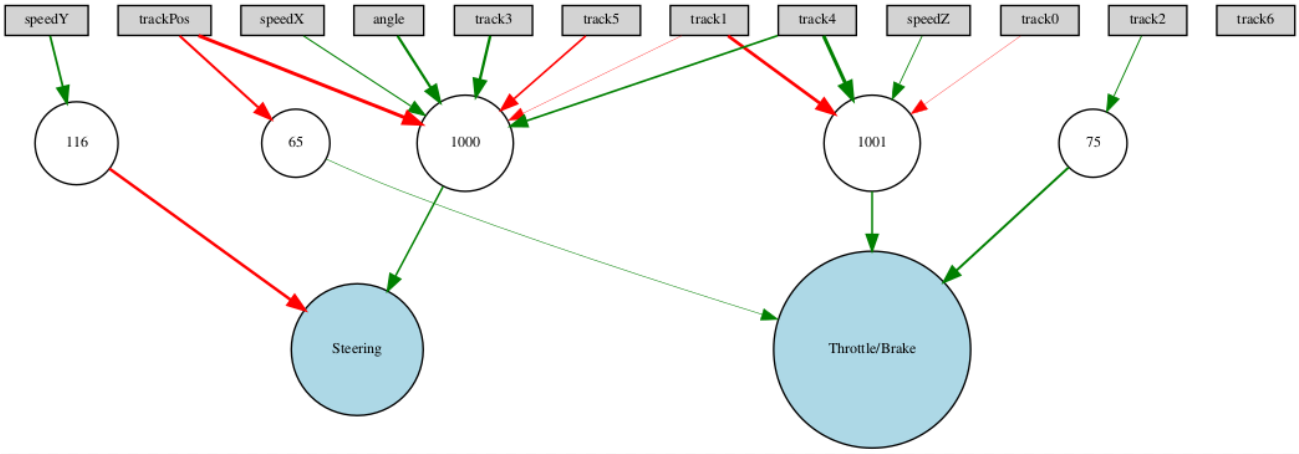

Figure 21: Genome structure after the conversion

5.3.2 DDPG phase

Results from the DDPG part of ERL training are shown in Figure 22. The agents started succesfully completing full laps of the training track early – at around episode 470. Maximum reward of about 110k was reached towards the end of training – at around episode 1350.

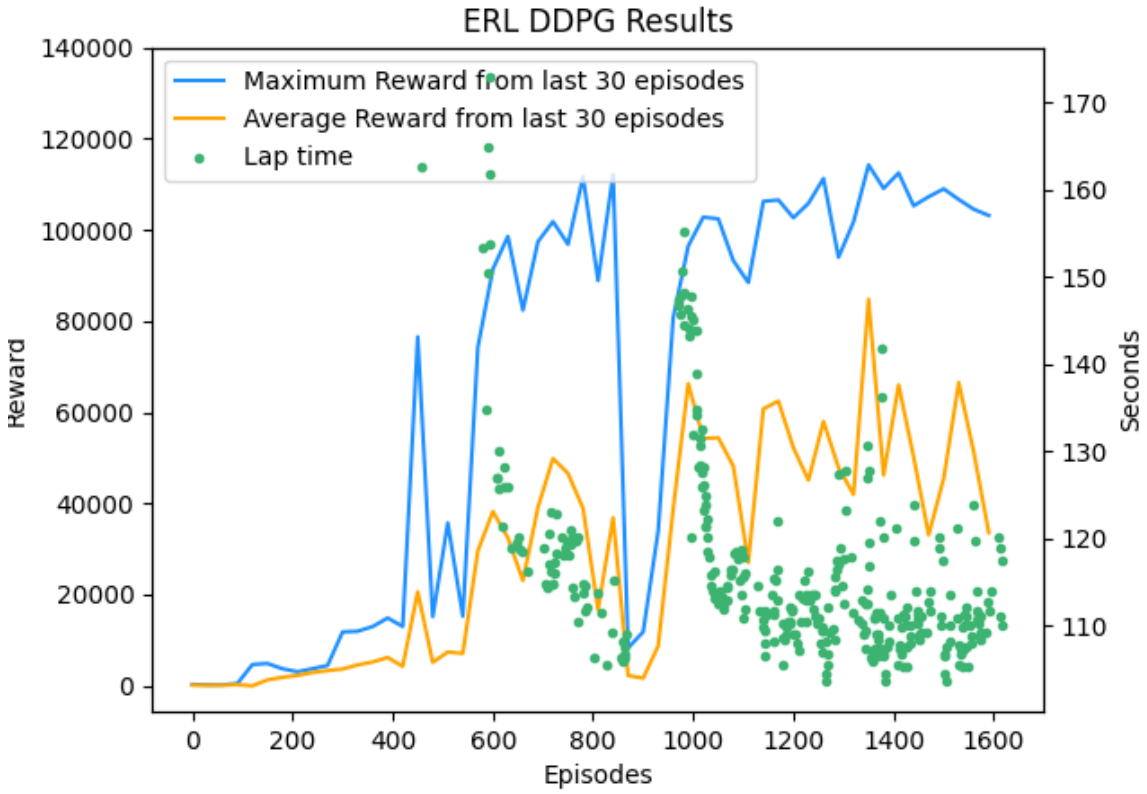

Figure 22: ERL DDPG Results

5.4 Comparison

5.4.1 Rewards

Figure 23 below shows the rewards comparison between the three different algorithms tested. For, NEAT, the standard term used to describe performance of genomes is “fitness”, however it means the same thing as reward. Therefore term “reward” will be used for all algorithms in order to avoid confusion during comparison.

The learning curves show three distinctly different behaviours.

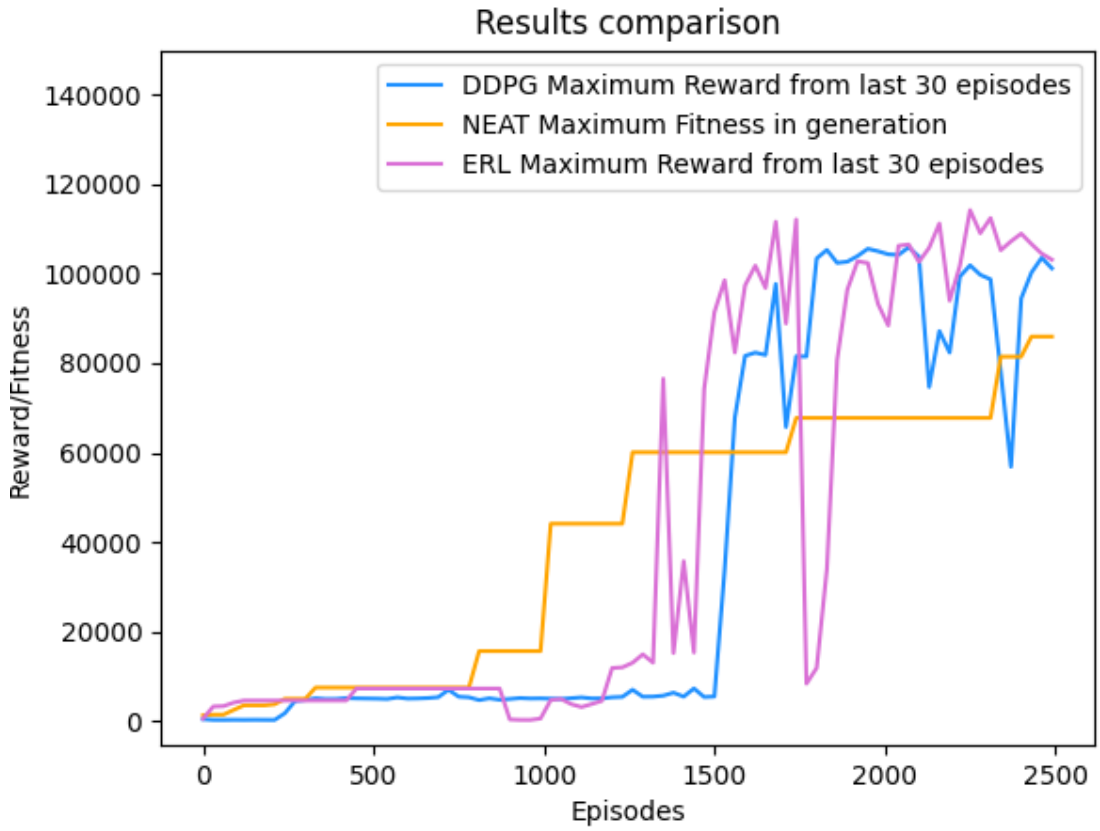

Figure 23: Reward results comparison

NEAT, driven purely by evolutionary search, improves in abrupt steps. Every few hundred episodes, a new network architecture emerges that is better than the last. That pattern continues throughout training and by the end of 2500 episodes NEAT reaches rewards of over 80 000 per episode. The growth of rewards is the steadiest of all algorithms, however highest achieved reward is the lowest.

DDPG follows an almost opposite trajectory. Because it begins with a randomly initialised policy and needs to first accumulate experience for its replay buffer, the algorithm spends roughly the first 1400 episodes achieving nearly zero reward. Once the replay buffer is populated enough, reward levels rise quickly to around 100 000. The rapid ascent demonstrates the power of gradient-based refinement. However, the blue curve also has occasional drops, showing DDPG's susceptibility to critic over-estimation and the instabilities that come with continuous-control optimisation.

The hybrid Evolutionary Reinforcement Learning (ERL) approach inherits the best of both worlds. During its initial NEAT phase, it shows the same exploratory jumps visible in the orange curve. When the evolved actor is plugged into DDPG at episode 900, it takes around 400 more episodes for the agent to start completing full laps. ERL reaches six-figure rewards several hundred episodes

before pure DDPG. It peaks at a reward of around 110 000 – the highest result out of all algorithms tested. It does, however, also suffer from instability issues, just like pure DDPG. At around episode 1750, there was a large decrease in rewards of around 90%, which lasted for nearly 100 episodes.

These dynamics clarify the complementary strengths of evolution and policy gradients. NEAT supplies broad, population-level exploration and automatically discovers a network topology well suited to the driving task, sparing the designer a lengthy hyper-parameter search. DDPG then exploits that head start, using fine-grained weight updates to squeeze out performance gains that random mutation alone could not reach.

5.4.2 Lap times

Table 5 shows a summary of the results from agents representing NEAT, DDPG and ERL.

Table 5: Lap time results

Track	**NEAT**	**DDPG**	**ERL**	**Bot**
E-Track 2 *(Training)*	02:12	1:47	1:43	1:57*
E-Track 5	1:00	00:51	0:44	0:39
Alpine 1	4:15	03:30*	3:17*	2:49*
Forza	3:59	DNF	DNF	DNF
* - car came in contact with a wall				

Lap time results on individual circuits make the contrast between the three learning strategies even clearer. On the training circuit (E-Track 2), NEAT manages 2min12s - slowest out of the 3 algorithms. DDPG and ERL had similar lap times – 1min47s and 1min43s respectively. The built-in bot took 1min57s to complete a lap. The ranking mirrors the reward curves.

Results on E-Track 5, the shortest layout tested, are similar. Ranking between the trained algorithms is the same, however the built-in bot was faster than all of them on this track. This is likely due to the track layout being much less complex than the training track. The built-in bot was manually programmed to achieve fast times here, while the trained agents learned to drive safer to avoid issues on more challenging layouts.

Results on the Alpine 1 circuit - a longer, more technical track - show that windy mountain roads can be challenging for some of the agents. Here NEAT completes a lap in 4min15s without incident. The DDPG agent clips a barrier on its 3min30s run. ERL driver, although faster at 3min17s, also scrapes a wall. The built-in bot was the quickest at 2min49s but also had some contact.

The challenges of navigating a track seamlessly are most notable on the highly demanding Forza circuit. Only the NEAT agent finishes, logging a conservative 3min59s. DDPG and the hybrid algorithm both failed to complete a lap. The default bot also retired. NEAT's population-level exploration evidently discovers a set of behaviours robust enough to keep the car on the road where DDPG-based learners cannot. This solitary victory for NEAT does not change the broader pattern but it does emphasise the complementary strengths that motivate hybridisation in the first place.

Taken together, the lap time results support findings from the reward curves. NEAT excels at rapidly producing a workable solution and shows stability on unforgiving tracks. DDPG, given enough experience, can refine control to a much higher ceiling but risks instability. ERL inherits NEAT's architecture and broad exploration, then harnesses gradient updates to achieve the fastest and most consistent performance on the majority of circuits. The hybrid therefore offers a tangible, measurable benefit - often slicing seconds from lap times and hundreds of episodes from training. Results demonstrate that combining evolutionary search with policy-gradient refinement is more than an academic exercise for autonomous racing.

5.4.3 Driving performance

It is advised to watch the video comparison [Mil25b] before reading this section. It shows how the different algorithms performed on the track that they were trained on, as well as a previously unseen track (Alpine 1).

Observing the driving performance of the algorithms, it is clear that agents trained using ERL are capable of the fastest lap times. They have learned to follow a faster path, more resembling of the optimal "racing line" – often getting close to the apexes of turns, as shown in Figure 24. Also, they tend to carry more speed through the corners. The agent seems to favour sticking to the left side of the track, however that would likely be solved with a bit of further training.

Figure 24: Agent driving close to the apex of a corner

DDPG agent also drives quickly, however it lacks consistency and learned a chaotic policy – constantly turning right and left as it moves along the track.

On the other hand, NEAT agents evolved to have a more conservative driving style but they are more consistent. They tend to mostly stick to the middle of the track. This sacrifices maximum performance but lets the cars to successfully complete more laps – DDPG and ERL agents tend to run off the track onto a surface with less friction and lose control much more frequently.

Also, NEAT agents learned to brake in a more controlled fashion – they would slow down before a corner, coast through the corner and then apply throttle again, much like human racing drivers. DDPG agents would often try applying the brake in the middle of the corner, which would unsettle the car's weight transfer and often cause a spin. Agents of all algorithms were able to learn counter-steering during oversteer situations, which sometimes saved them from spinning, allowing to continue the lap. An example of such a situation is shown in Figure 25.

Figure 25: Agent counter-steering into the corner

Interestingly, straight sections seemed to cause more trouble for all of the algorithms, than corners. Often, agents would start steering left and right in quick succession when the track was straight. This would sometimes result in spinning out and ending the episode. DDPG was most affected by this and was swerving wildly, especially on unseen tracks. ERL was able to maintain a straight line best.

5.5 Future opportunities

In order to advance the idea of developing a hybrid algorithm further, the following possible adjustments have been identified:

- Implement both, NEAT and DDPG, in a single platform such as PyTorch. Having to convert the network from a genome generated by neat-python library to a TensorFlow model caused some performance loss during conversion.

- Iterative approach – evolve a neural network with NEAT and plug it into DDPG as the actor network several times during training. Periodically injecting fresh actors would help the system escape local optima that could trap a single-policy learner.

- Train on multiple tracks – this would expose the agents to a wider variety of driving scenarios, therefore generalisation should be improved.

6. Conclusion

This study presented and evaluated three algorithms - NEAT, DDPG, and ERL - for autonomous car control within a simulated racing environment. Agents of all three algorithms, NEAT, DDPG and ERL, learned relatively good strategies and were able to complete laps around the training track. The comparative results revealed that:

- NEAT offers robust exploration and structural adaptability, enabling consistent lap completion and better generalization to unseen tracks, though with slower lap times. It showed the quickest early learning and the lowest rate of crashes, confirming the exploratory power and robustness of evolutionary learning.

- DDPG achieved high-performance driving on familiar tracks but was hindered by instability and poor generalization to new environments. Once its replay buffer is mature, DDPG produces noticeably higher late-stage rewards than NEAT but pays for that with a larger number of safety violations and occasional training-time instabilities.

- ERL, the hybrid approach, successfully combined the strengths of both. It was learning faster than pure DDPG and achieving superior lap times and rewards, while maintaining a reasonable safety profile. ERL fulfils the promise of hybridisation - it learns as quickly as evolution, finishes as strongly as policy gradients, and does so with fewer crashes than either baseline alone.

From a practical standpoint in autonomous racing, the hybrid method means fewer simulated kilometres to achieve a competent driver and higher ultimate lap-time efficiency. The evidence therefore confirms that combining NEAT with DDPG is more than a conceptual exercise - it

delivers improvements in both learning speed and asymptotic performance, making ERL a compelling avenue for future intelligent driving systems.

Key limitations include hardware constraints that limited parallel NEAT training, the conversion from genome to TensorFlow, and limited training on only one track. Despite these, the findings strongly support hybridization as a practical and effective strategy in autonomous driving AI.

Future research should explore training all components in a unified framework such as PyTorch, iterative actor evolution, and testing in more complex or realistic environments like CARLA.

Ultimately, this work presents a compelling proof-of-concept for combining neuroevolution with reinforcement learning in the pursuit of intelligent, high-performant autonomous control systems.

References

[Ant23] Antoniadis, W., 2023. Activation functions: Sigmoid vs Tanh. Baeldung on Computer Science. Available at: https://www.baeldung.com/cs/sigmoid-vs-tanh-functions [Accessed 5 Sep 2023].

[CLL+09] Cardamone, L., Loiacono, D. and Lanzi, P.L., 2009. Evolving competitive car controllers for racing games with neuroevolution. In: IEEE Congress on Evolutionary Computation, 2009. CEC '09. IEEE. Available at: https://www.researchgate.net/publication/220740175_Evolving_competitive_car_controllers_for_racing_games_with_neuroevolution [Accessed 13 May 2025].

[CLL+10] Cardamone, L., Loiacono, D. and Lanzi, P.L., 2010. Learning to drive in the open racing car simulator using online neuroevolution. IEEE Transactions on Computational Intelligence and AI in Games, 2(3), pp.176–190. Available at: https://www.luigicardamone.it/tesi-pubblicazioni/tciag10learning.pdf [Accessed 13 May 2025].

[Cod19] CodeReclaimers, 2019. NEAT-python's documentation. Available at: https://neat-python.readthedocs.io/en/latest/ (Accessed: 05 September 2023).

[LCL13] Loiacono, D., Cardamone, L. and Lanzi, P.L., 2013. Simulated Car Racing Championship: Competition Software Manual. arXiv preprint arXiv:1304.1672. Available at: https://arxiv.org/pdf/1304.1672.pdf [Accessed 4 Sep 2023].

[LHP+19] Lillicrap, T.P., Hunt, J.J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D. and Wierstra, D., 2015. Continuous control with deep reinforcement learning. arXiv preprint arXiv:1509.02971. Available at: https://arxiv.org/abs/1509.02971 [Accessed 5 Sep 2023].

[Lau16] Lau, B., 2016. Using Keras and Deep Deterministic Policy Gradient to play TORCS. Blog post. Available at: https://yanpanlau.github.io/2016/10/11/Torcs-Keras.html [Accessed 6 Sep 2023].

[Mil25a] Milvydas, Ž., 2025. torcs-autonomous-driving. GitLab repository. Available at: https://gitlab.com/zhygio/public/torcs-autonomous-driving [Accessed 14 Jan 2025].

[Mil25b] Milvydas, Ž., 2025. Autonomous Driving in TORCS [video]. YouTube. Available at: https://youtu.be/NZPY_HJ5nIk [Accessed 14 Jan 2025].

[PCH+17] Peng, Y., Chen, Q., Zhang, M. and Zhang, Y., 2017. Automated state feature learning for actor-critic reinforcement learning through NEAT. In: Proceedings of the Genetic

and Evolutionary Computation Conference (GECCO '17), Berlin, Germany, July 2017. ACM, pp. 515–522. Available at: https://homepages.ecs.vuw.ac.nz/~yimei/papers/GECCO17-Yiming.pdf [Accessed 13 May 2025].

[SM02] Stanley, K.O. and Miikkulainen, R., 2002. Evolving neural networks through augmenting topologies. In: Proceedings of the 2002 Congress on Evolutionary Computation (CEC 2002), pp. 1423–1430. IEEE. Available at: https://nn.cs.utexas.edu/downloads/papers/stanley.cec02.pdf [Accessed 4 Sep 2023].

[Ten25] TensorFlow, 2025. TensorFlow API documentation. Available at: https://www.tensorflow.org/api_docs/python/ [Accessed 15 Jan 2025].

[WJW19] Wang, J., Jia, H. and Weng, Y., 2018. Autonomous Driving using Deep Reinforcement Learning in TORCS. arXiv preprint arXiv:1811.11329. Available at: https://arxiv.org/abs/1811.11329 [Accessed 13 May 2025].

[WZM+24] Wang, R., Zhao, Y., Ma, H. and Zhang, D., 2024. Research on energy scheduling optimization strategy with compressed air energy storage. Sustainability, 16(18), p.8008. Available at: https://www.mdpi.com/2071-1050/16/18/8008 [Accessed 14 Jan 2025].

Appendices

Appendix A – NEAT settings

```
[NEAT]
fitness_criterion     = max
fitness_threshold     = 200000
pop_size              = 30
reset_on_extinction   = True

[DefaultGenome]
# node activation options
activation_default      = tanh
activation_mutate_rate  = 0.0
activation_options      = tanh

# node aggregation options
aggregation_default     = sum
aggregation_mutate_rate = 0.0
aggregation_options     = sum

# node bias options
bias_init_mean          = 0.0
bias_init_stdev         = 1.0
bias_max_value          = 30.0
bias_min_value          = -30.0
bias_mutate_power       = 0.5
bias_mutate_rate        = 0.7
bias_replace_rate       = 0.1

# genome compatibility options
compatibility_disjoint_coefficient = 1.0
compatibility_weight_coefficient   = 0.5

# connection add/remove rates
conn_add_prob           = 0.5
```

```
conn_delete_prob        = 0.5

# connection enable options
enabled_default         = True
enabled_mutate_rate     = 0.15

feed_forward            = True
initial_connection      = full_direct

# node add/remove rates
node_add_prob           = 0.4
node_delete_prob        = 0.4

# network parameters
num_hidden              = 0
num_inputs              = 12
num_outputs             = 2

# node response options
response_init_mean      = 1.0
response_init_stdev     = 0.0
response_max_value      = 30.0
response_min_value      = -30.0
response_mutate_power   = 0.0
response_mutate_rate    = 0.0
response_replace_rate   = 0.0

# connection weight options
weight_init_mean        = 0.0
weight_init_stdev       = 1.0
weight_max_value        = 30
weight_min_value        = -30
weight_mutate_power     = 0.5
weight_mutate_rate      = 0.8
weight_replace_rate     = 0.1
```

```
[DefaultSpeciesSet]
compatibility_threshold = 3.0

[DefaultStagnation]
species_fitness_func = max
max_stagnation       = 10
species_elitism      = 2

[DefaultReproduction]
elitism            = 2
survival_threshold = 0.2
```

Appendix B – DDPG settings

BUFFER_SIZE = 100000

BATCH_SIZE = 32

GAMMA = 0.99

TAU = 0.001 # Target Network HyperParameters

LRA = 0.0001 # Learning rate for Actor

LRC = 0.001 # Learning rate for Critic

action_dim = 2 # Steering, Acceleration/Brake

state_dim = 12 # Number of sensors

EXPLORE = 100000.0

max_steps = 1000

OU_steps = 200000

OU_mu = 0.0

OU_theta = 0.6

OU_sigma = 0.3

Appendix C – ERL NEAT settings

```
[NEAT]
fitness_criterion     = max
fitness_threshold     = 1000000000
pop_size              = 30
reset_on_extinction   = True

[DefaultGenome]
# node activation options
activation_default      = tanh
activation_mutate_rate  = 0.0
activation_options      = tanh

# node aggregation options
aggregation_default     = sum
aggregation_mutate_rate = 0.0
aggregation_options     = sum

# node bias options
bias_init_mean          = 0.0
bias_init_stdev         = 1.0
bias_max_value          = 30.0
bias_min_value          = -30.0
bias_mutate_power       = 0.5
bias_mutate_rate        = 0.4
bias_replace_rate       = 0.1

# genome compatibility options
compatibility_disjoint_coefficient = 1.0
compatibility_weight_coefficient   = 0.5

# connection add/remove rates
conn_add_prob           = 0.5
conn_delete_prob        = 0.5
```

```
# connection enable options
enabled_default         = True
enabled_mutate_rate     = 0.2

feed_forward            = True
initial_connection      = full_nodirect

# node add/remove rates
node_add_prob           = 0.7
node_delete_prob        = 0.3

# network parameters
num_hidden              = 0
num_inputs              = 12
num_outputs             = 2

# node response options
response_init_mean      = 1.0
response_init_stdev     = 0.0
response_max_value      = 1.0
response_min_value      = 1.0
response_mutate_power   = 0.0
response_mutate_rate    = 0.0
response_replace_rate   = 0.0

# connection weight options
weight_init_mean        = 0.0
weight_init_stdev       = 1.0
weight_max_value        = 30
weight_min_value        = -30
weight_mutate_power     = 0.5
weight_mutate_rate      = 0.5
weight_replace_rate     = 0.1

[DefaultSpeciesSet]
compatibility_threshold = 1.0
```

```
[DefaultStagnation]
species_fitness_func = mean
max_stagnation       = 10
species_elitism      = 2

[DefaultReproduction]
elitism            = 2
survival_threshold = 0.2
```