Final Project MGAIA: TrackMania

Anca-Mihaela Matei	s4004507
Giulia Rivetti	s4026543
Evan Meltz	s3817911
Kacper Kadziolka	s4115945
Rajiv Jethoe	s3490750
Luc Schreurs	s1987747

Abstract. This report details the development and analysis of three distinct reinforcement learning (RL) agents: Soft Actor-Critic (SAC), Deep Deterministic Policy Gradient (DDPG), and Proximal Policy Optimization (PPO). These agents are used to navigate a car within the complex environment of the Trackmania game. Leveraging the `tmrl` package, these agents are trained with two different types of sensory inputs: high-dimensional pixel data and structured LIDAR data, both used independently to assess their impact on the agents' performance. Each agent's architecture is adapted to effectively process each particular input type, optimizing learning and decision-making capabilities. We evaluate the agents based on their ability to learn and adapt to the dynamic game environment, comparing their performance metrics. The results highlight both the strengths and weaknesses of each agent and input method, contributing to a deeper understanding of how various reinforcement learning strategies and sensory inputs can be optimized for real-time gaming applications.

Keywords: Reinforcement Learning · Modern game AI · SAC · DDPG · PPO · Trackmania · TMRL

1 Introduction

In this assignment, we implemented the Reinforcement Learning (RL) methods of Soft Actor-Critic (SAC) [5, 8], Deep Deterministic Policy Gradient (DDPG) [9], and Proximal Policy Optimization (PPO) [2] to navigate a predefined track in the simulated racing game Trackmania (TM) [15]. We trained these three RL agents to successfully navigate a basic track while avoiding obstacles and preventing falls. Each agent was trained separately high-dimensional pixel data (image data) and structured LIDAR data to better understand how each input type affects training.

After implementing and training the agents on both data types, we conducted an evaluation and comparison to assess the effectiveness of each method in adapting to and successfully navigating the predefined track. This work was facilitated by the TMRL [14] framework, a comprehensive distributed RL framework for TM designed to assist in the real-time training of Deep RL agents.

In conducting this assignment, we gained practical experience in dealing with various RL methods and how they can be potentially integrated into future video games to enhance realism and engagement.

2 TMRL

As previously mentioned, to conduct this project, we made use of the Trackmania Reinforcement Learning (TMRL) framework to train our agents. "TMRL is a fully-fledged distributed reinforcement learning (RL) framework for robotics, designed to help train deep RL AIs in real-time applications.", according to its creators. [14].

TMRL achieves this by utilizing a pipeline, which we adapted to fit our needs per implemented algorithm. We can train our selected RL algorithms – SAC, DDPG, and PPO – by specifying which pipeline to use in the TMRL `config` file (see documentation). The algorithms collect samples and store them in a replay memory while simultaneously being used to train a policy that maps observations to corresponding actions.

3 Inputs: Pixels vs LIDAR

Both sensory types used in the assignment, high-dimensional pixel data and structured LIDAR data, require a different fundamental architecture for the network. These input types will be explained shortly below.

3.1 LIDAR

Our RL agents that interact with the LIDAR environment, are characterized by a multi-layer perceptron (MLP) to process the LIDAR measurements. In particular, the MLP used by the actor module, depicted in Table 1 has an input layer of size 83, which corresponds to the dimension of the observations. This input is then passed through two hidden layers, which both have 256 neurons and the ReLU activation function. As for the critic's MLP, shown in Table 2, the input layer combines both the observation and the actions dimensions; then, also in this case the two hidden layers have 256 neurons and use the ReLU activation function. Then the critic module is also characterized by an output layer with a single neuron, which is used to output the value corresponding to a given state-action pair.

Layer	Neurons
`Input`	83
`Dense1`	256
`Dense2`	256

Table 1: MLP architecture used for the actor module in the LIDAR environment.

Layer	Neurons
`Input`	83 + dim_act
`Dense1`	256
`Dense2`	256
`Output`	1

Table 2: MLP architecture used for the critic module in the LIDAR environment.

3.2 Pixels

Our pixel-based reinforcement learning (RL) agents utilize a Convolutional Neural Network (CNN) to process high-dimensional image data, complemented by a Multi-Layer Perceptron (MLP) for handling additional inputs such as speed, gear, and RPM. The architecture is designed to effectively integrate both visual and numerical data, enhancing the agent's ability to learn and perform in complex environments.

The image data is processed through a series of convolutional layers in the `VanillaCNN` class. This class initializes the CNN with the structure depicted in Table 3.

Layer	Filters	Kernel Size	Stride
`Conv1`	64	8	2
`Conv2`	64	4	2
`Conv3`	128	4	2
`Conv4`	128	4	2

Table 3: CNN Architecture used in our pixel-processing network.

Layer	Neurons
`Input`	Flat Features + 9
`Dense1`	256
`Dense2`	256
`Output`	1

Table 4: MLP architecture used in our pixel-processing network.

The pixel-based agent processes high-dimensional image data using a series of convolutional (`Conv`) layers. The input to the network is a 64x64 grayscale image, ingested at 20 frames per second. Each convolutional layer is followed by a `ReLU` activation function. The network starts with a convolutional layer having 64 filters, a kernel size of 8, and a stride of 2. This is followed by three more convolutional layers with increasing filter sizes and consistent kernel and stride values. The final output of these layers is a flattened feature vector that captures the spatial hierarchies and patterns from the input images.

After processing the images through the CNN, the flattened feature vector is concatenated with additional numerical inputs such as speed, gear, RPM, and action components. This combined vector is then passed through an MLP. For the critic network (see Table 4), the MLP consists of two hidden layers with 256 neurons each, followed by an output layer with a single neuron. The MLP utilizes ReLU activation functions to introduce non-linearity and facilitate complex function approximation.

This architecture efficiently combines high-dimensional image data with lower-dimensional numerical data, enabling the agent to learn complex behaviours by leveraging both types of input. The CNN extracts spatial features from the images, while the MLP integrates these features with other relevant state information, facilitating comprehensive decision-making during training.

4 Algorithms

In this section, we explore the key algorithms we have implemented and utilized in our project. These algorithms form the backbone of our approach to training agents capable of navigating the complex environment of Trackmania. Each algorithm utilizes different techniques and architectures to balance the exploration-exploitation trade-off. The algorithms discussed include Soft Actor-Critic (SAC), Deep Deterministic Policy Gradient (DDPG), and Proximal Policy Optimization (PPO). We will delve into the theoretical foundation, architectural design choices, and the specific implementation of each algorithm, to highlight their modifications and their effectiveness in this project.

4.1 Soft Actor-Critic

SAC represents a foundational approach in modern reinforcement learning by combining the benefits of both policy-based and value-based methods. The agent learns to determine the optimal actions to take in a given environment state s to maximize cumulative reward R. SAC leverages two key components in a hybrid approach, addressing the limitations of using each method independently. In the SAC framework, the Actor learns a policy to make decisions, while the Critic evaluates the Actor's decisions by computing the value function. This synergy harnesses the strengths of both policy and value functions, making the learning process more stable and efficient, while maintaining a balance between exploration and exploitation [7].

The Actor's role is to make decisions based on the current policy, represented as $\pi_\theta(a|s)$. This notation indicates the probability of taking action a when in state s. By continuously improving this policy, the Actor explores different actions within the action space to potentially maximize the expected cumulative rewards. The policy is defined by parameters, which are adjusted by the Actor network, denoted by θ.

Conversely, the Critic assesses the actions chosen by the Actor by estimating the value function through the state-value function $V(s)$. This estimation of the expected cumulative reward serves as a measure of the quality of the actions suggested by the Actor, thereby directing the Actor towards actions that yield higher expected returns. Equation (1) depicts the expected return R_t from state s under the current policy π.

$$V_\phi(s) - \mathbb{E}[R_t|s_t = s] \tag{1}$$

The learning process for both the policy (Actor) and the value estimates (Critic) occurs simultaneously. The Actor's and Critic's parameters are updated using gradient descent methods. To effectively balance the exploration/exploitation trade-off, which is crucial in complex environments like the Trackmania game, the SAC agent also utilizes entropy regularization.

Algorithm 1 SAC Pseudocode

1: **Input:** Actor policy π_θ, and Critic value function V_ϕ
2: **Init:** Critic parameters ϕ, and Actor parameters θ
3: **for** each episode **do**
4: Sample one trace batch followed by actor policy π_θ
5: Optimize entropy coefficient with backpropagation
6: Calculate critic MSE loss with Bellman backup for Q functions
7: Calculate entropy-regularized actor loss
8: Perform gradient optimization for both policy and value parameters
9: **end for**

Algorithm 1 presents a general, simplified flow of states during the algorithm's execution. The main training loop runs for a predetermined number of episodes as the agent interacts with the environment. The agent

selects actions based on the probabilities generated by the Actor and observes the resulting states. The returns are calculated efficiently while keeping the Q-Networks static to facilitate policy updates. Subsequently, the losses for both the Actor and the Critic are calculated, and their gradients are backpropagated.

We recognize the significant enhancements that can be achieved with the Asynchronous Advantage Actor-Critic (A3C) algorithm. This method introduces parallelism by utilizing multiple workers whose results are combined, allowing the environment to be explored with optimal efficiency.

4.2 Deep Deterministic Policy Gradient

DDPG is a model-free, off-policy actor-critic algorithm that combines the advantages of value-based and policy-based reinforcement learning methods [11]. It was developed specifically to address continuous action spaces, making it particularly suited for applications like controlling vehicles in simulation environments such as Trackmania.

DDPG operates by maintaining two main components: an actor and a critic, similar to the SAC algorithm. The actor in DDPG deterministically approximates the optimal policy and outputs what it considers the best action for any given state. In contrast, according to the previous section the actor in SAC samples actions based on a stochastic policy, allowing for exploration of the action space. This method directly maps states to actions, simplifying the policy but potentially reducing exploration compared to SAC's entropy-enhanced approach. The critic in DDPG evaluates the proposed action by the actor, computing the value function to estimate future rewards. This dual structure allows DDPG to optimize the policy based on the critic's feedback, which assesses the quality of actions taken by the actor.

Algorithm 2 DDPG Pseudocode

Input: initial policy parameters θ, Q-function parameters ϕ, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters: $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$
repeat
    Observe state s and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{\text{low}}, a_{\text{high}})$, where $\epsilon \sim \mathcal{N}$
    Execute action a in the environment
    Observe next state s', reward r, and done signal d to indicate whether s' is terminal
    Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
    if s' is terminal **then**
        Reset environment state
    end if
    if it's time to update **then**
        for however many updates **do**
            Randomly sample a batch of transitions $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
            Compute targets $y(r, s', d) = r + \gamma(1 - d)Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$
            Update Q-function by one step of gradient descent using

$$\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_\phi(s, a) - y(r, s', d)\right)^2$$

            Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \mu_\theta(s))$$

            Update target networks with

$$\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1 - \rho)\phi$$

$$\theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1 - \rho)\theta$$

        end for
    end if
until convergence

DDPG enhances the stability of learning by incorporating target networks for both the actor and the critic. These target networks are essentially delayed copies of the main networks and are updated at a slower rate. This method ensures that the learning targets do not change too frequently, preventing the learning process from becoming unstable. By consistently maintaining these targets, the algorithm avoids the risks associated with abrupt changes in policy evaluation that could lead to divergent or oscillating behaviors.

Another important aspect relates to the exploration-exploitation trade-off. In DDPG, the exploration necessary for effective policy learning in continuous action spaces is achieved by adding noise to the actor's outputs. Specifically, we used Ornstein-Uhlenbeck noise (OUNoise). This type of noise is particularly useful in environments like driving simulations, where actions need to exhibit a degree of continuity rather than abrupt changes. The OUNoise helps maintain exploration, which is coherent over time, thus enabling the policy to explore more effectively in environments that have a smooth underlying structure.

The efficiency of DDPG's training process is significantly enhanced by the use of a replay buffer, a crucial component for off-policy reinforcement learning algorithms. In the implementation of DDPG for Trackmania using the package, the management of the replay buffer is conveniently handled by the package itself. This integration facilitates a seamless and effective use of past experiences to improve the learning process.

The `tmrl` package comes equipped with a built-in mechanism to store and retrieve experiences (state, action, reward, next state) in the replay buffer. As a result, the primary development effort did not involve setting up the buffer from scratch but rather focusing on optimizing the training function.

The DDPG algorithm presented in [6] is outlined in Algorithm 2, providing a detailed step-by-step procedural format for implementation, where γ is the discount factor, ρ is named poliak and has values between $[0, 1]$ and $\mu_\theta(s)$ represents the deterministic policy that the algorithm aims to learn.

4.3 Proximal Policy Optimization

PPO (Proximal Policy Optimization) is a policy optimization gradient-based algorithm that was introduced to improve the training stability of reinforcement learning agents. The core feature of PPO is its ability to prevent excessively large updates to the policy, which can destabilize the learning process. This is achieved by computing the ratio between the current policy and the previous one, which indicates how much the policy has actually changed. This ratio is then clipped within a certain range to remove the incentive for the current policy to go too far from the old one, thus favouring more controlled updates [13]. PPO is characterized by two components:

1. The **Actor**, which outputs the probability distribution for the next action given a certain state. Therefore the actor represents the policy that the agent follows to make decisions.
2. The **Critic** which estimates the expected cumulative reward from that state [12].

The goal of the algorithm is to learn a policy that maximizes the obtained cumulative rewards given the experience during training. The algorithm aims to make the policy more likely to select actions with a high "advantage", where the advantage function measures how much better an action is compared to the critic's prediction of the expected reward [12]. By focusing on actions with higher advantages, PPO effectively guides the policy towards more rewarding behaviour.

Algorithm 3 shows the pseudocode of PPO, where the main steps of the algorithm are reported. Initially, the algorithm follows the same steps as DDPG, choosing the action provided by the policy and receiving back from the environment the corresponding reward and state. The core difference of PPO lies in the training step. After having sampled a mini-batch from the replay buffer, the algorithm proceeds by computing the advantage function $A(s_t, a_t)$. Then, the ration function r_t is computed, which represents the probability of taking action a_t at state of s_t in the current policy divided by the previous one. Therefore, it is essentially used to estimate the divergence between old and current policy. Both the advantage and the ration functions are used to compute the loss L_{clip}, along with the clipped factor, which is used to clip r_t, so that in this way we do not have a too large policy update, since the current policy can't be too different from the older one. After that, the total loss is computed as the difference between the clipped loss L_{clip} and the value loss L_{value}. Finally, the network's parameters are updated with stochastic gradient descent.

Algorithm 3 PPO Pseudocode

Input: initial policy parameters θ, Q-function parameters ϕ, empty replay buffer $\mathcal{D}$
repeat
  Observe state s and select action a
  Execute action a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  for each epoch **do**
    Sample a minibatch from the trajectory buffer
    Compute advantages $A(s_t, a_t)$
    Compute current policy probability ratio: $r_t = \frac{\pi_\theta^{actor}(s_t,a_t)}{\pi_\theta^{actor_{old}}(s_t,a_t)}$
    Compute surrogate objective: $L_{clip} = \hat{\mathbb{E}}_t \left[\min(r_t * A(s_t, a_t), clip(ratio, 1-\epsilon, 1+\epsilon)A(s_t, a_t))\right]$
    Compute value loss: $L_{value} = \hat{\mathbb{E}}_t \left[(V_\theta(s_t) - (r_t + \gamma \cdot V_\theta(s_{t+1}))\right]^2$
    Compute total loss: $L_{total} = L_{value} - L_{clip}$
    Update actor-critic network parameters with stochastic gradient descent: $\theta_{actor} \leftarrow \theta_{actor} - \alpha \cdot \nabla_\theta L_{total}$
    $\theta_{critic} \leftarrow \theta_{critic} - \alpha \cdot \nabla_\theta L_{value}$
    Update target network parameters: $\theta_{actor_{old}} \leftarrow \theta_{actor}$
    $\theta_{critic_{old}} \leftarrow \theta_{critic}$
  end for
until convergence

5 Lecture Topic Relation

In keeping with the requirements of this assignment, our project relates to and incorporates several lectures covered in this course namely:

- **Learning and Optimization (Lecture 3):** This lecture is the cornerstone of our entire project as this introduces the topic of RL. It goes into detail regarding how agents can learn from interactions with their environment to optimize their decision-making processes. This lecture delves into the basic principles of RL, highlighting its strengths as well as its shortcomings. We used this knowledge when implementing our agents to better improve their track navigation skills over time, and to adapt strategies to maximize rewards. This lecture also describes the potential of Deep Learning and RL in the gaming industry and was a source of inspiration for the idea of attempting to train such an agent on a game like Trackmania.
- **Experimentation (Lecture 5):** This lecture provided us with the framework for testing and comparing the performance of the RL agents. It gave us information which changed our perception of how such an experiment should be conducted. This includes not having pre-defined expectations like which algorithms will perform better and why, which will lower the occurrence of success bias, as well as gaining an understanding of the tediousness of actually obtaining positive results. In addition, we applied several other concepts mentioned throughout our project such as adding noise to agents to encourage exploration actors and random batch sampling. Through this systematic type of experimentation, we can quantify the effectiveness of different learning strategies and refine our approaches if needed.
- **Learning from Pixels (Lecture 6):** This lecture is critical to our project, as it lays the foundation for our primary method of interaction with the Trackmania environment. It details how processing visual data from the game is possible and the techniques for doing so. This visual data is turned into the necessary feedback required by the agent to make decisions regarding navigation on the track. Although we also train agents using LIDAR data, which can be less computationally expensive, it does not match the accuracy provided by high-dimensional pixel data. Therefore, applying the concepts presented in this lecture is essential for creating agents that can effectively learn to navigate the game environment.

6 Experiments

In this section, the experimental setup and results will be discussed. In the experimental setup, we will explain in detail what track we use, what hyperparameters we use and why and explain the overall process. In the results, we will discuss the comparison between an agent trained on pixel and LIDAR data. This way we can observe and discuss the differences in performance when training on different data.

6.1 Experimental Setup

For the experimental setup, we trained all four agents on both the pixel and LIDAR data:

- Soft Actor-Critic (SAC) Double Critic
- Soft Actor-Critic (SAC) Single Critic
- PPO
- DDPG

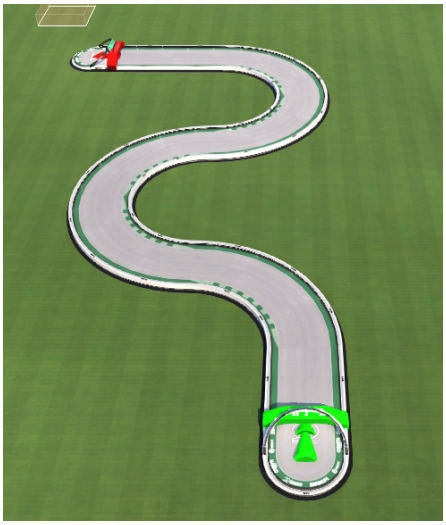


Fig. 1: Image of our custom track used to train all our agents for a fair comparison.

We created a custom track with enough complexity to evaluate if the agent is learning over time (see Figure 1). The track is short to facilitate training, with 64x64 pixel frames at 20 frames per second being processed by the network. This volume of data is substantial for our PCs, necessitating a smaller track. However, we included corners and bends to ensure the RL-agents learn navigation skills. While the corners are not highly challenging, they provide sufficient complexity for the agent to learn effectively.

Hyperparameter Search Looking for suitable hyperparameters is always a challenge in Machine Learning and often requires some form of intuition [1], but in Deep Reinforcement Learning it is even harder we noticed. Training takes a long time, as can be seen in 6. A lot of problems were encountered in finding suitable hyperparameters, mainly, the sheer time it takes before improvement can be observed by a change in hyperparameters. Methods like Grid or random search were considered for hyperparameter optimization, but they were quickly discarded due to their significant time budget needs. Usually in our RL problem it takes 1 epoch with 100 steps in each epoch to observe the agent learn. One epoch can be 10 minutes, but it can also be 2-3 hours depending on the model. With 8 distinct agents and 4 distinct model architectures that needed to be trained we went with hyperparameters that are often standard for these models or we used the same hyperparameters that were already provided to us, which we will talk about in the next subsection.

Hyperparameter selection For the SAC model we were already provided with carefully chosen hyperparameters by the TMRL authors. They took a long time to tune their model they say, so we made good use of their hard work. All of our models used the Adam optimizer, all models were trained for 10 epochs. Batch sizes were all 256 with exception of PPO, which was 100, this is because of the extreme amount of time PPO takes to train. For the DDPG model we chose very similar hyperparameters as for the SAC agent.

As can be seen in Table 8, these are the hyperparameters selected. As said before, the SAC agent's hyperparameters were pre-determined by the TMRL authors. The DDPG agent almost uses the same hyperparameters with slightly different actor and critic learning rates. The PPO implementation has the most

Hyperparameter	Value
LR actor	0.000005
LR critic	0.00003
γ	0.99
polyak	0.995
α	0.02
$\beta 1$, $\beta 2$ actor	0.997, 0.997
$\beta 1$, $\beta 2$ critic	0.997, 0.997

Table 5: Hyperparameters for all models that used the SAC architecture.

Hyperparameter	Value
LR actor	0.00005
LR critic	0.0003
γ	0.99
$\beta 1$, $\beta 2$ actor	0.9, 0.999
$\beta 1$, $\beta 2$ critic	0.9, 0.999

Table 6: Hyperparameters for all models that used the DDPG architecture

Hyperparameter	Value
Learning rate actor	0.0003
Learning rate critic	0.001
γ	0.99
λ	0.95
Clip ratio	0.2
Train iteration actor	10
Train iteration critic	10
$\beta 1$, $\beta 2$ actor	0.997, 0.997
$\beta 1$, $\beta 2$ critic	0.997, 0.997

Table 7: Hyperparameters for all models that used the PPO architecture

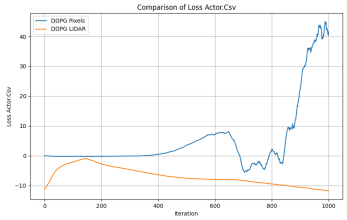
Table 8: All hyperparameters used for all models and their variations

hyperparameters, and in turn performed the worst out of all these models, as can be seen in 3a, 3b, 3c. We went with the suggestions from [4] to guide the hyperparameter selection process for the PPO implementation. But according to [3][10] PPO is sensitive to hyperparameters explaining its performance. The sensitivity was noticed during training and further tuning by way of seeing different behaviours of the agent by changing a single hyperparameter a slight bit. The decision was made not to tune by hand and just select the parameters set out in [4] even though the performance would be bad.

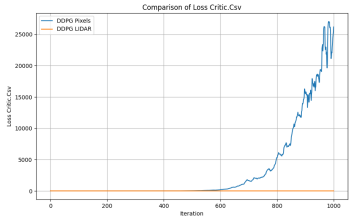
In PPO, the training iterations for the actor and critic has to be of significant size to facilitate the learning process. [4] recommends 80 iterations for both, such that the actor and critic can learn. This, however took too much time and we had to scale it down a significant amount to make training feasible.

6.2 Experimental Results

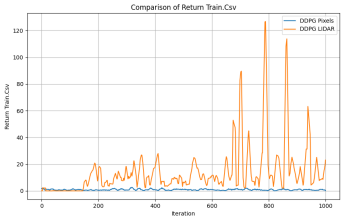
In this section we will discuss the results of our testing process, which mainly includes training our 8 agents on our own custom track. While logging all necessary metrics to Weights and Biases, which is an online monitoring platform for machine learning applications. This was set-up by the TMRL authors as a convenient tool for the participants in the competition. A smoothing window of 10 steps was applied to all graphs, this was mostly done to make the "return_train" graphs more legible, but other graphs do benefit from this smoothing as well.



(a) Actor loss for both LIDAR and Pixel DPPG Agents



(b) Critic loss for the LIDAR and Pixel DDPG agents



(c) Returns for the Pixel and LIDAR DDPG agents

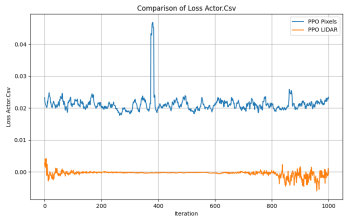
Fig. 2: Overall results of training the DDPG LIDAR/Pixel agents. You can see the LIDAR agent performing much better as seen in Figure 2a and Figure 2b. Here you can see that the LIDAR agent's loss keeps going down while the Pixel agent's loss keeps rising. The same can be said for the critic loss and the returns are much higher as well for the LIDAR agent

The analysis of Figure 2 indicates that the LIDAR agent significantly outperforms the pixel agent. This conclusion is supported by the steady decrease in actor loss over time, as shown in Figure 2a. A gradual

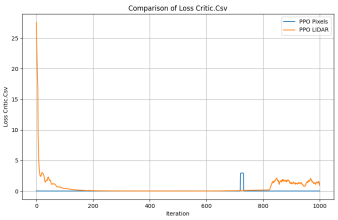
reduction in actor loss suggests that the Q-value evaluations by the critic network are accurate, which is crucial for stable learning. In contrast, if the Q-value evaluations were poor, we would expect to see erratic and oscillating actor loss values.

Further examination of Figure 2b reveals that the critic loss for the LIDAR agent remains stable around zero. While the pixel agent's critic loss is obscured due to its high value, the inference behavior of the pixel agent suggests that it is not learning effectively. This ineffectiveness is likely due to inadequately tuned hyperparameters, which are especially critical given the high dimensionality and complexity of pixel data. The time constraints of the project prevented a more thorough and well though out hyperparameter tuning process.

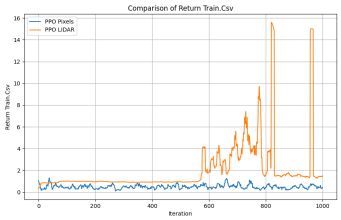
Figure 2c shows that the pixel agent consistently returns values close to zero, both during training and inference. This indicates that the pixel agent fails to move from its starting position. In contrast, the LIDAR agent demonstrates a steady improvement in returns, albeit with some instability. This suggests that while the LIDAR agent's learning process may not be entirely optimal or smooth, it is capable of completing the track during inference, reflecting a higher overall learning effectiveness.



(a) Actor loss of the Pixel and LIDAR PPO agents



(b) Critic loss for the LIDAR And Pixel PPO agents



(c) Returns for the Pixel and LIDAR PPO agents

Fig. 3: Overall results of training the PPO LIDAR/Pixel agents. The results show that basically no learning has been done. This can be seen in both 3a, 3b. Here the losses don't go down which means no learning is occuring. Furthermore, the returns do not go up significantly. LIDAR does do a bit better though.

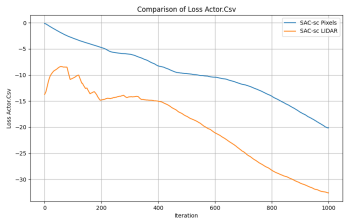
As previously discussed, the performance of the PPO agent is notably poor, likely due to its high sensitivity to hyperparameters [3]. We used guidelines from [4] to select these parameters, but hyperparameter tuning is highly dependent on the specific training environment. Given the high-dimensional nature of pixel data, a more meticulous tuning process was necessary. Unfortunately, both the LIDAR and pixel agents trained with PPO did not perform well.

The losses for both agents indicate a lack of learning, with oscillating values clearly visible in Figure 3a. A similar pattern is observed in Figure 3b. The returns graph in Figure 3c shows that returns remain close to zero throughout the training process. Although the LIDAR agent exhibits occasional spikes in returns, suggesting brief exploration of potentially useful strategies, these gains are not sustained due to suboptimal hyperparameter settings we think.

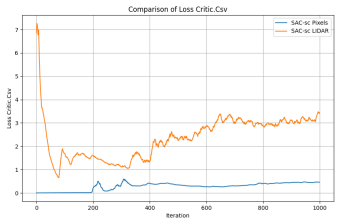
The poor performance of the PPO agents can be attributed to insufficient hyperparameter tuning, a critical factor given the algorithm's sensitivity. The limited time spent on tuning these parameters likely contributed to the observed performance issues. During inference, both models fail to perform effectively; they either move backwards, remain stationary, or repeatedly drive into barriers. This behavior, particularly the repetitive actions, suggests that the agents may be stuck in local optima, failing to explore more effective strategies due to the poor hyperparameter configuration.

The SAC Single Critic architecture demonstrates strong performance consistently from start to finish. This conclusion is supported by Figures 4a and 4b, where both the LIDAR and pixel agents show a steady decline in actor losses. The pixel agent begins with a loss of zero, while the LIDAR agent starts lower but still shows significant improvement. The smoother actor loss curve for the pixel agent suggests a more stable learning process compared to the LIDAR agent.

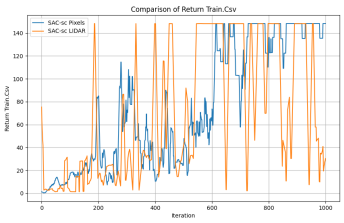
This stability is further corroborated by the critic loss graph, which shows a sharp decrease in critic loss for both agents. However, the critic loss for the LIDAR agent begins to increase and oscillate over time,



(a) Actor loss for both LIDAR and Pixel SAC-SC agents



(b) Critic loss for both LIDAR and Pixel SAC-SC agents

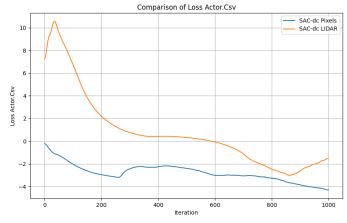


(c) Both returns over 10 epochs of the SAC-SC LIDAR and Pixel agents

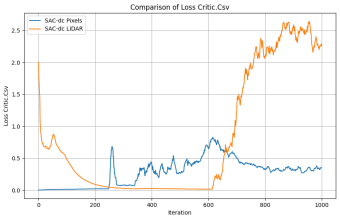
Fig. 4: Overall results of training the SAC-SC LIDAR/Pixel agents. From 4a and 4b we can tell that the pixel agent does slightly better because of the more steady and lower decrease of the actor loss and the slow increase of the critic loss. In the 4c we can also see that LIDAR has a much more unstable graph, which shows that LIDAR does well, but is very unstable, Pixel is also unstable but much less so.

indicating potential difficulties in learning the environment's complexities. This instability might be due to the Single Critic architecture's lack of the stabilization factor present in the Double Critic architecture used in other SAC agents.

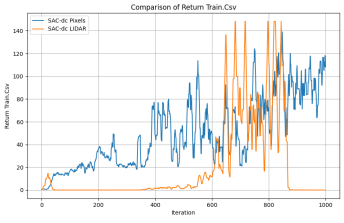
The returns graph in Figure 4c reveals that despite some variability, both agents achieve consistently high returns on the track. The pixel agent exhibits more variability in returns than the LIDAR agent, although it is hard to tell which performs better from the graph, possibly due to the pixel agent's greater need for the stabilization effect provided by the double critic network. During inference, both LIDAR and pixel agents are capable of completing the track. However, the pixel agent drives significantly slower than the LIDAR agent, indicating that the pixel agent requires more training. This difference in performance can be attributed to the high dimensionality of pixel data processed by a CNN architecture, in contrast to the simpler MLP used for the LIDAR agent. The LIDAR agent, although more aggressive and prone to bumping into walls, ultimately completes the track more efficiently. The SAC Double Critic, as provided by TMRL, performed very well as



(a) Actor loss of the Pixel and LIDAR SAC-DC agents



(b) Critic loss of the Pixel and LIDAR SAC-DC agents



(c) Overal returns of the Pixel and LIDAR SAC-DC Agents

Fig. 5: Overall results of training the SAC-DC LIDAR/Pixel agents. From 5a and 5b we can see that the pixel agent again does better because the actor loss starts at zero and slowly declines while the critic loss increases much more for the LIDAR agent. We can also see that in 5c that around 850 steps/8.5 epochs the LIDAR agents stops getting any returns.

expected. However, when comparing it to our own implementation using the LIDAR agent, notable differences emerge. Figure 5a illustrates the LIDAR agent's more turbulent training process, characterized by peaks and valleys in the graph. This turbulence is further evidenced by the LIDAR agent starting with a significantly higher loss compared to the pixel agent, which began with a loss around zero.

Around epoch 8.5, the LIDAR agent begins to diverge, as shown by the zeroing out of the return graph in Figure 5c. This divergence can be predicted by examining Figure 5b, where the critic loss for the LIDAR agent starts to increase drastically and steadily around epoch 6. The significant increase in critic loss indicates

that the critic network is struggling to accurately estimate Q-values. When the critic loss is high, the Q-values used to update the actor's policy are unreliable, leading to poor policy updates and, ultimately, a degradation in the agent's performance, as reflected by the zeroing returns.

This zeroing out could be attributed to catastrophic forgetting, where the critic network fails to retain useful knowledge and suddenly loses the ability to provide accurate value estimates. This issue can arise from overly aggressive updates, high learning rates, or insufficient regularization. In our case, it is more likely due to the former rather than the latter, given the observed pattern of critic loss.

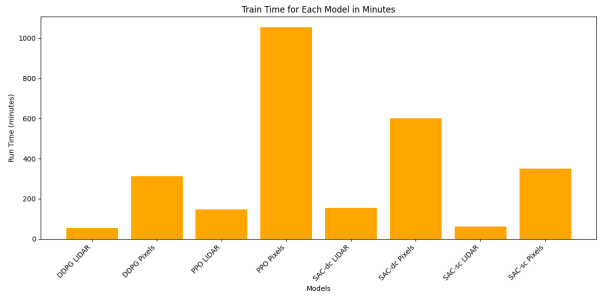


Fig. 6: All full training times of all our trained agents plotted in minutes

In Figure 6 we can see all our trained agents' training time plotted against each other in minutes. An overall observation can be made that training of the LIDAR agents takes much less time than training the Pixel agent variants. This makes sense, since pixel agents are training on 64x64 images of the game with the CNN network ingesting 20 frames a second. While the LIDAR's data input is much smaller. The significant difference between pixel and LIDAR training time could be factor taken into account when choosing which agents to train. it is all dependant on how complex and varied your environment is.

6.3 Experiments Conclusion

In this section, we discussed the experimental setup and results of training various reinforcement learning (RL) agents using both pixel and LIDAR data. We created a custom track with sufficient complexity to test the learning capabilities of the agents. The track was designed to be short to manage the computational load, especially considering the high data input rate from pixel data (64x64 images at 20 frames per second). Four types of agents were trained: Soft Actor-Critic (SAC) Double Critic, SAC Single Critic, Proximal Policy Optimization (PPO), and Deep Deterministic Policy Gradient (DDPG), each with pixel and LIDAR data.

The results highlight significant performance differences between models trained on pixel versus LIDAR data. The LIDAR-based DDPG agent outperformed its pixel counterpart, showing a steady decrease in actor loss and improved returns, while the pixel agent struggled with high and unstable losses. Similarly, PPO agents exhibited poor performance due to the algorithm's sensitivity to hyperparameters, with both pixel and LIDAR agents failing to show significant learning. The SAC Single Critic agents demonstrated better performance, with both types showing a steady decrease in actor loss, though the pixel agent had a smoother learning curve. The SAC Double Critic agents initially performed well, but the LIDAR agent showed instability and a sharp increase in critic loss, leading to a complete drop in returns around epoch 8.5. Overall, LIDAR agents trained faster and more efficiently than pixel agents, suggesting that the lower dimensionality of LIDAR data is advantageous for RL training. This efficiency could be critical when deciding which data type to use, depending on the complexity and variety of the training environment.

7 Discussion and Conclusion

The experiments conducted in this study provide a comprehensive comparison between reinforcement learning agents (RL) trained on pixel data and those trained on LIDAR data. The agents implemented include Soft Actor-Critic (SAC) with both single and double critics, Deep Deterministic Policy Gradient (DDPG), and Proximal Policy Optimization (PPO). Each of these agents was trained on a custom designed track that presented sufficient complexity to challenge their learning capabilities.

From the results, it is evident that the type of input significantly influences the learning efficiency and performance of the RL agents. The LIDAR based agents consistently outperformed their pixel based counterparts across most metrics. This can be attributed to the lower dimensionality and structured nature of LIDAR data, which enables faster and more stable learning. For instance, the LIDAR based DDPG agent showed a steady decline in actor loss and a corresponding improvement in returns, indicating effective learning and policy optimization. In contrast, the pixel-based DDPG agent showed high and unstable losses, suggesting difficulties in learning from the high dimensional pixel data.

The performance of the PPO agents highlighted the sensitivity of this algorithm to hyperparameters. Both the pixel and LIDAR PPO agents failed to demonstrate significant learning, with losses remaining high and returns close to zero. This underperformance highlights the need for thorough hyperparameter tuning, especially for algorithms like PPO that are inherently sensitive to parameter settings. The PPO's poor performance can be primarily attributed to the lack of extensive hyperparameter tuning due to time constraints.

The SAC agents, particularly the single critic architecture, demonstrated better performance overall. Both the pixel and LIDAR SAC-SC agents showed a steady decrease in actor loss, though the pixel agent exhibited a smoother learning curve. The critic loss patterns suggested that the LIDAR agent struggled more with the complexities of the environment, leading to oscillations in critic loss. This instability was less pronounced in the pixel agent, potentially due to the different processing architectures used for pixel and LIDAR data.

Interestingly, the SAC Double Critic (SAC-DC) architecture, which was expected to provide superior performance due to its design, showed a divergent behavior for the LIDAR agent. Around epoch 8.5, the LIDAR agent's returns dropped to zero, coinciding with a sharp increase in critic loss. This suggests that the critic network failed to provide reliable Q-value estimates, leading to poor policy updates. The pixel agent, however, maintained a relatively stable performance, although starting with a higher initial loss.

In conclusion, the study reveals that LIDAR data generally provides an advantage in training RL agents due to its lower dimensionality and structured nature. However, the choice of algorithm and careful tuning of hyperparameters are crucial for optimizing performance. The SAC-SC architecture emerged as the most robust across different data types, while PPO's performance highlighted the challenges associated with hyperparameter sensitivity. These findings offer valuable insights into the design and training of RL agents for complex environments, suggesting that a balance between data type, network architecture, and parameter tuning is essential for achieving optimal performance. Additionally, during inference, it was observed that pixel models perform inference but fail in the subsequent round. We think this has something to do with the TMRL package, but we did not have enough time to troubleshoot this issue.

References

[1] Marc Claesen and Bart De Moor. "Hyperparameter search in machine learning". In: *arXiv preprint arXiv:1502.02127* (2015).

[2] John Schulman et al. *Proximal Policy Optimization Algorithms*. Aug. 2017. DOI: 10.48550/arXiv.1707.06347. arXiv: 1707.06347 [cs]. (Visited on 05/29/2024).

[3] John Schulman et al. *Proximal Policy Optimization Algorithms*. Accessed: 2024-05-01. 2017. URL: https://openai.com/index/openai-baselines-ppo/.

[4] AurelianTactics. *PPO Hyperparameters and Ranges*. https://medium.com/aureliantactics/ppo-hyperparameters-and-ranges-6fc2d29bccbe. Accessed: 2024-05-05. 2018.

[5] Tuomas Haarnoja et al. *Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor*. Aug. 2018. DOI: 10.48550/arXiv.1801.01290. arXiv: 1801.01290 [cs, stat]. (Visited on 05/29/2024).

[6] OpenAI. *Deep Deterministic Policy Gradient*. https://spinningup.openai.com/en/latest/algorithms/ddpg.html. Accessed: 2024-05-1o. 2018.

[7] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2018. ISBN: 9780262039246.

[8] Tuomas Haarnoja et al. *Soft Actor-Critic Algorithms and Applications*. Jan. 2019. DOI: 10.48550/arXiv.1812.05905. arXiv: 1812.05905 [cs, stat]. (Visited on 05/29/2024).

[9] Timothy P. Lillicrap et al. *Continuous Control with Deep Reinforcement Learning*. July 2019. DOI: 10.48550/arXiv.1509.02971. arXiv: 1509.02971 [cs, stat]. (Visited on 05/29/2024).

[10] Ashley Hill et al. *Stable Baselines3: Reliable Reinforcement Learning Implementations*. https://stable-baselines3.readthedocs.io/en/master/modules/ppo.html. Accessed: 2024-05-05. 2021.

[11] Aske Plaat. *Deep Reinforcement Learning*. Springer Nature Singapore, 2022. ISBN: 9789811906381. DOI: 10.1007/978-981-19-0638-1.

[12] Brian Pulfer. *PPO — Intuitive guide to state-of-the-art Reinforcement Learning*. https://medium.com/@brianpulfer/ppo-intuitive-guide-to-state-of-the-art-reinforcement-learning-410a41cb675b. 2022.

[13] Thomas Simonini. *Proximal Policy Optimization (PPO)*. https://huggingface.co/blog/deep-rl-ppo. 2022.

[14] Yann Bouteiller, Edouard Geze, and Andrej Gobe. *TMRL, Reinforcement Learning for Real-Time Applications*. Version 0.6.1. Apr. 10, 2024. URL: https://github.com/trackmania-rl/tmrl (visited on 04/11/2024).

[15] *Trackmania*. Ubisoft. URL: https://www.ubisoft.com/en-gb/game/trackmania/trackmania (visited on 05/30/2024).