TransformerKart: A Full SuperTuxKart AI Controller

CSCE 689 Final Project

Vicente Balmaseda Leah Tomotaki

December 5, 2022

The code can be found at https://github.com/vibalcam/deep-rl-supertux-race
5-minute summary video: https://youtu.be/cvs2AC091fE

Abstract

Reinforcement Learning is a popular solution for autonomous racing simulators, but can be hindered by overly complex state spaces. Decision Transformers can be a powerful tool to overcome this issue due to the Transformer's propensity for simplicity and scalability. Decision Transformers turn reinforcement learning into a sequence modeling problem by processing the sequence of states, actions, and returns with an autoregressive Transformer model. This allows the model to learn offline and generate future behaviors, saving resources while yielding impressive results. Another benefit of the Decision Transformer is its ability to learn from suboptimal trajectories. While the training dataset may consist of many random, suboptimal paths, the Decision Transformer can combine these experiences into an optimal solution. This allows it to surpass its training model and the game's AI–a fact that sets it apart from imitation learning. In this study, we applied a decision transformer to train an agent to drive in the SuperTuxKart environment. SuperTuxKart is a 3D open-source arcade racing game with various tracks (similar to Mario Kart). We first gather a dataset of episodes using a suboptimal baseline controller. For each timestep, a Convolutional Neural Network is used to obtain the state information from the image of the kart on the track. This image embedding is combined with the kart's velocity and the kart's rotation to generate an embedding for the state information. The sequence of states, actions, and rewards-to-go are then used to train the Decision Transformers for the controller in a supervised manner. Multiple controllers have been trained through this process. The result is a robust controller that learns to simultaneously steer, drift, and accelerate. Our solution not only learns a full controller using only information available to the kart, but also shows improvement over the baseline policy used to build the dataset and SuperTuxKart's AI driver, showing potential to succeed in even more complex game environments.

1 Introduction

Decision Transformers are a cutting-edge development in the field of Reinforcement Learning and are well suited for the domain of autonomous automobile racing simulation. Decision Transformers take advantage of "the simplicity and scalability of Transformer architecture" [4] by converting reinforcement learning problems into conditional sequence modeling problems. Transformers use a self-attention mechanism to process sequential data and are commonly used for language modeling problems with great success.

Traditional approaches to autonomous driving and racing simulation, such as Deep Deterministic Policy Gradient, must fit value functions via Bellman Backups and compute gradients. This can take large amounts of memory and specialized frameworks. [4] Other approaches, such as Imitation Learning, can only be as successful as the coach policy. In contrast, Decision Transformers can learn offline from suboptimal trajectories to generate optimal policies at test time. This can yield results that surpass the baseline coach model.

For this project, we tackled the challenge of creating a novel Decision Transformer-based agent that is designed to effectively complete a track in the SuperTuxKart environment. SuperTuxKart is a 3D open-source arcade racing game with various tracks (similar to Mario Kart). To play the game and obtain information from it using Python, we used pystk, which is a heavily modified version of SuperTuxKart that provides a highly efficient and customizable Python interface. This allows it to be used as an environment for experiments. We then trained a Convolutional Neural Network to take the input image of the kart on the track as one of the input states for the model. Lastly, we evaluated our work using an Aim point baseline and the game's AI driver.

2 Related Work

2.1 Autonomous Driving

In recent years, there have been great advancements in the autonomous driving field. Although Super Tux Kart is a game environment that does not directly reflect driving in the real world, there are many similarities in the state and continuous action spaces.

Deep Deterministic Policy Gradients (DDPG) have been proven to be effective in The Open Racing Car Simulator (TORCS) [12]. TORCS differs from Super Tux Kart because it models driving in a complex, real-world environment much more closely and does not have added game features like the item boxes or nitro boost. DDPG is a model-free, off-policy, actor-critic algorithm. It is conducive to continuous action spaces [6], such as degrees of rotation of a steering wheel and acceleration of a car. The deterministic policy gradient is the expected gradient of action values after directly mapping state spaces to action spaces. This requires fewer samples than its stochastic counterpart. It is able to learn off-policy by updating the policy gradient direction from the input of a critic that criticizes the actor's actions at each step. DPG is extended to Deep Deterministic Policy Gradient by using a deep neural network as the function approximator.

The reward function used in this study is a function of the speed along the track axis (positive reward for parallel movement & negative reward for perpendicular movement) and the position on the track (positive reward for staying in the middle of the track) [12]. This model uses sensor input (such as track position, speed, angle, etc) from the simulator as input instead of the raw images [12].

An issue with traditional reinforcement learning approaches, like DDPG, is difficulty representing and generalizing the complex states of a real world road environment. Semantic Segmentation of RGB image inputs have been used to simplify states. Studies show improvement when labeling the input images as road, road marking, and other (obstacles). [10] However, this does not resolve the issue of inefficient use of data by traditional RL models. This requires enormous amounts of sample data and training, which can be time-consuming and costly. [8]

A proposed solution for the autonomous driving problem that addresses these shortcomings is to use a "Transformer-based structure to better capture the relations of the heterogeneous elements and inject predictive information into the scene representation" [8]. This study by Liu, et al. first uses a multi-stage transformer to encode the different elements in the scene captured by the autonomous vehicle camera and model how the different elements are interacting. It then employees a sequential latent transformer to create a latent representation that can be used to train a soft actor-critic model. This method performs better against Data-regularized Q-learning, Soft Actor-Critic, Proximal Policy Optimization, and Rule-based Driver Model baselines.

As shown above, many concepts can be taken from the autonomous driving field to improve our algorithm. However, the application of Reinforcement Learning to the Super Tux Kart environment is much simpler than in the real world. In real-world and simulated autonomous driving, safety and adherence to traffic laws are extremely important. The primary goal of an autonomous vehicle is to deliver passengers to their destinations safely. This is reflected in the reward function. [6]

Conversely, in the Super Tux Kart environment, the goal is to complete the track as quickly as possible. It is acceptable to make dangerous decisions, such as running into other karts or leaving the track, if the end result is faster. Therefore, we can simplify our reward function to ignore safety concerns and traffic laws.

2.2 Training Controllers

Deep Learning controllers are usually trained through imitation learning or reinforcement learning. In imitation learning, we record the observations, actions taken and results that an expert obtains while performing the task. We then train a controller on the gathered data. Thus, we are training a controller to "imitate" the expert in a supervised manner. This greatly simplifies the training process by reducing it to a supervised learning task.

However, imitation learning suffers from a fundamental difficulty. The controllers predictions affect future observations, which violates the i.i.d. (independent and identically distributed) assumption made by most learning approaches. This may lead to the controller not learning how to recover from mistakes since it has not seen these "wrong" scenarios in the experts observations, due to the expert being too "good". Thus, small mistakes made by the controller accumulate over time, leading to poor performance. DAGGER [11] solves this issue following an iterative procedure. DAGGER first trains a policy to imitate an expert. In the following iterations, it uses the policy trained in the previous iteration to generate observations to augment the dataset, and trains a policy on the augmented dataset. The intuition behind DAGGER is to train the policy on the set of inputs it might encounter in order to fix the distribution mismatch and to teach the model to recover from errors. Moreover, DAGGER also mentions the possibility of using a combination between the last trained policy and the expert to augment the dataset. Doing so allows to use the expert to guide the policies, which is most important during the first few iterations, when policies make more mistakes. The resulting model is able to learn to steer on the Super Tux Kart game in order to reduce the average number of falls per lap [1].

Our approach differs from the controller trained with DAGGER mainly in two aspects. First, instead of using imitation learning, we will use a reinforcement learning, which does not require a pre-existing policy, and a sequence modeling approach, which uses a dataset of observations obtained from a variety of non-expert policies. Thus, we will not need an expert to train our model, which reduces the human effort required to train our controller. Second, instead of focusing on reducing the number of falls, we will focus on the race aspect of the game, i.e., completing the track in the least amount of time.

Instead of using imitation learning to train controllers, reinforcement learning can be used. Reinforcement learning approaches have the advantage that they require no human input at all. Instead, the controller will learn while playing the game by itself. Many of the more "traditional" reinforcement learning approaches are not able to efficiently deal with high-dimensional data such as images. DQN [9] was the first model to be able to combine reinforcement learning with deep neural networks, thus allowing to learn control policies from high dimensional input (such as images). DQN is able to train a neural network to learn q values for a policy. Therefore, it is a combination of Q-learning and deep learning. One of its main characteristics is the use of a replay buffer. The replay buffer allows storing past observations to create a dynamic dataset from which the deep neural network can learn from. This buffer allows achieving a greater sample efficiency, decorrelating observations, and smoothing the transition between policies, thus avoiding divergence due to violating the i.i.d. principle and bringing the task closer to supervised learning. Also, it utilizes two neural networks to give consistent targets during temporal difference calculations.

While DQN is capable of dealing with high-dimensional state spaces, it can not handle continuous action spaces. More recent techniques include Deep Deterministic Policy Gradients (DDPG) [7], which is capable of learning policies in high-dimensional, continuous action spaces. This model follows an actor-critic approach, being based on deterministic policy gradient (DPG) with insights from DQN. It also utilizes batch normalization from deep learning to deal with different physical units and ranges on different feature information from states (for example, velocity vs distance).

A different approach to reinforcement learning is abstracting it as a sequence modeling problem. Decision Transformer [4] takes advantage of the advances in language modeling using the transformers architecture to tackle reinforcement learning tasks. This model trains on a dataset of offline trajectories by sampling minibatches of timestep length and outputs optimal actions. Each step is represented as a sequence of rewards-to-go (reward remaining to be obtained by the trajectory), state, and action. The use of rewards-to-go allows the model to specify the expertise of the output policy. Also, training on collected experience in a sequence modeling manner allows avoiding bootstrapping, and thus, avoiding one of the "deadly triad".

2.3 Game Playing

Most of the reinforcement learning literature has been benchmarked on the Arcade Learning Environment (ALE) [2]. ALE is a platform for evaluating domain-independent models on different environments that emulate Atari games. ALE also provides reward functions for training reinforcement learning models on them. These environments together with other reinforcement learning benchmarks are included in OpenAI Gym [3]. OpenAI Gym provides a convenient and general interface to ease testing reinforcement learning algorithms on a wide variety of environments. More importantly, it offers an extendable interface that can be used to create new environments. We have taken advantage of this feature to create an OpenAI environment that wraps the Super Tux Kart game.

Super Tux Kart is a 3D open-source arcade racing game with various tracks (similar to Mario Kart). This game has been used as an evaluation environment for previous research, such as the DAGGER algorithm that trains a controller using imitation learning. NeuralKart [5] develops an autopilot that trains and plays without human intervention on Mario Kart 64. To accomplish this task, it utilizes a search AI with complete control over the emulator to generate a labeled training set. This labeled training set is then used by a Convolutional Neural Network (CNN) to train a steering policy. Lastly, the resulting controller is trained on an augmented dataset generated by running the search AI over randomly sampled states. Our approach differs in that it is solely trained on game data accessible to the kart (image, velocity, and rotation), not having access to the emulator during training or evaluation. Moreover, our approach uses reinforcement learning and decision transformers instead of imitation learning.

3 Environment

This study takes place in the **SuperTuxKart**[1] environment. SuperTuxKart is a 3D open-source arcade racing game with various tracks (similar to Mario Kart). To play the game and obtain information from it using Python, we will use **pystk**[2] Pystk is a heavily

[1]Try out SuperTuxKart: `https://supertuxkart.net/`
[2]pystk docs: `https://pystk.readthedocs.io`

modified version that provides a highly efficient and customizable Python interface. This allows it to be used as an environment for experiments. Since this environment was not available in OpenAI Gym, we first implemented an environment that connects pystk to Gym. We then set some environment-specific variables for the experiments, such as the number of laps, the rescue time (the time after which a car is "rescued" when stuck), and the tracks that will be used for training and testing. This environment can be represented as a Markov Decision Problem as shown below.

3.1 States

Each state will be defined by the image of the kart on the road, the kart's velocity, and the kart's rotation. Including the velocity and rotation of the kart makes the state Markovian because we do not need to reference previous states to determine the current trajectory of the kart. The velocity and rotation of the kart are properties of the pystk Kart class.

Starting state is the race's starting line. The terminal state is the race's finish line after k loops (k is an environment variable).

3.2 Actions

The actions the kart can take are accelerate, drift, and steer. These are defined in the pystk Action class.

- Steer: Determines the change in the kart's rotation. This is a continuous action space with values between -1 and 1, inclusive.
- Accelerate: Determines the speed that the kart accelerates. This is a continuous action with values between 0 and 1, inclusive. The model can also be trained with acceleration set to a default constant of 1 for simplicity.
- Drift: A boolean value representing whether or not the kart is drifting. Drifting can speed up turns, but may cause the kart to drift off course. Initially, the model was trained without the drift action, but this was later implemented. Drifting can be turned on or off when running the model.

We decided not to include the break, fire, nitro, or rescue actions that are provided by pystk. The game automatically calls the rescue action when the kart goes in the wrong direction for too long or gets stuck. The agent will not be allowed to call the rescue action since it would allow the kart to orient itself in the optimal direction without learning to drive. Moreover, the players in the SuperTuxKart game have no control over the rescue action. We did not include the fire and nitro actions for simplicity because they require interaction with the other players and very situational actions.

3.3 State Transition Probabilities:

The simulator defines the stochasticity in the environment. This is introduced to the environment through the obstacles, other karts, items, and the simulator's physics (which are unknown).

3.4 Rewards

Note: The discount factor of past rewards is **1** because the environment is episodic, meaning that it has a fixed maximum length. Thus, the sum of rewards is also finite.

- **-1 for each timestep:** This will encourage the agent to take the least time possible.
- **+1 for each meter advanced along the road:** This reward will only be given once, meaning that it will be given with respect to the furthest distance along the road reached. This will provide a positive reward the agent can use to learn that it should keep moving forward along the road.

4 Controller

In this section, we present the architecture of the controller and how the state information is processed to predict what actions to take.

The architecture of the controller is a Decision Transformer [4] combined with a Convolutional Neural Network to process the image information. The Decision Transformer transforms the Reinforcement Learning task into a sequence modeling task and utilizes the autoregressive modeling capabilities of Transformers and the Attention mechanism to learn complex controllers.

The controller's inputs are the environment state information (image, velocity, and rotation), and its outputs are the environment actions (steer, drift, and acceleration). The controller does not predict the brake action, which is set to false since a similar effect can be obtained by reducing the acceleration. However, if desired, the controller could be readily extended to predict these or any other actions, such as using objects.

4.1 State Embedding Layer

This layer takes the different state information and transforms each of them independently into an embedding vector with the same shape (hidden size) so they can be processed by the Decision Transformer. The inputs and outputs of the embedding layer are summarized in Table 1.

Table 1: Input and output of the state embeddings (shape given in parenthesis)

Input	Output
Image (height, width, 3)	State embedding (3 hidden size)
Velocity (3)	
Rotation (4)	

The velocity and rotation information embedding layers consist of a linear layer followed by a ReLU activation and batch normalization. The image embedding layer consists of a pre-processing layer that resizes the image to 96×128 ($height \times width$) followed by a fully Convolutional Neural Network (CNN), a linear layer, ReLU activation, and batch normalization. Lastly, the velocity, rotation, and image embeddings are concatenated to form the state embedding.

4.2 Decision Transformer

The backbone of the controller is the Decision Transformer. This model takes a sequence of timesteps and predicts the next action to take using a Transformer. Each timestep is given as a tuple of returns-to-go (final return minus accumulated reward), state, and action taken. The inputs and outputs of the decision transformer are summarized in Table 2.

Table 2: Input and output of the Decision Transformer (shape given in parenthesis)

Input	Output
State (episode_length, state_dim) Actions (episode_length, action_dim) Rewards-to-go (episode_length, 1) Timesteps (episode_length)	Actions (episode_length, action_dim)

4.3 Output Layer

The output layer splits the actions tensor into a single tensor for each action (steer, acceleration, drift, and brake) and transforms them to the correct scale. For evaluation, the predicted actions for the last timestep is used as the actions taken by the controller. Table 3 presents the output functions used for each action.

Table 3: Output transformations done on each action

Action	Output Function
Steer	Tanh ($[1, 1]$)
Acceleration	Sigmoid ($[0, 1]$)
Drift	(value $>$ 0).int() (boolean)
Brake	(value $>$ 0).int() (boolean)

5 Experiments

In this section, we give an overview of the complete training process, from data collection to the actual training of the model. We will finish by presenting and discussing the results obtained by the controller.

For our experiments, we have trained a controller on a single track (lighthouse), leaving training on multiple tracks as future work. The controller is trained to finish the track in the minimum time possible.

The overall process for training is the following:

1. Gather dataset of episodes
2. Calculate rewards-to-go

3. Train controller model on the dataset
4. Optional: Augment the dataset using the trained controller and continue training
5. Evaluate the model on the environment

5.1 Data Collection

We first need to obtain a dataset on which to train our model. This could be done iteratively by evaluating our model and training it. However, in order to make the training process faster, we are going to use the baseline as our "coach". Therefore, we run the baseline on the environment for a certain number of episodes and gather the state, action, and rewards data for each timestep. An important fact is that our aim-point baseline model is not optimal, which means that we will be learning from a suboptimal model to gather the data. Once we have gathered the data for all the episodes, we calculate the rewards-to-go. This can be easily done since we have the reward information for each timestep.

Baseline. The aim-point baseline model works as follows. It takes the image information and predicts the center of the road at a fixed distance forward, which we will call the aim point. It then uses a preset set of heuristics to calculate the actions to take from the predicted aim point and the velocity of the kart. Therefore, the aim-point baseline relies heavily on manually set instructions which are not optimal.

Exploration. In order to increase exploration and augment the dataset, we have added noise to the coach model. The noise is added by sampling two values from a uniform distribution, one of them is added to the velocity and the other is multiplied to the aim point.

Dataset. The dataset collected had 150 episodes with approximately 600 timesteps per episode. The dataset has been collected by running two sets of episodes. For the first set, the baseline was able to drift. For the second set, drifting was disabled. Each set of data was collected using the following noise values:

- 15 episodes without adding noise
- 30 episodes with (0.05, 2.5) noise values (velocity and aim point respectively)
- 30 episodes with (0.1, 5) noise values (velocity and aim point respectively)

5.2 Training

Image embeddings. Before training the model on the dataset, we pretrain a CNN autoencoder on the dataset images. We will then use the encoder part of the autoencoder as the image embedding layer[3]. Doing so allows faster training of the model as a whole. Moreover, this layer is reused for all the models trained.

Training loop. For training the model, we iterate through the episodes randomly. For each episode, we reconstruct the state (images, velocity, and rotation), action, timestep, and rewards-to-go sequences and pass them through our model.

Loss functions. Table 4 presents the loss functions used for each of the predicted actions. The total loss is the sum of the losses for each action (this can be generalized to a weighted sum if we want to give more importance to learning a certain action). The reasoning behind using these losses is the following. The steering and acceleration actions are real values, thus using a regression loss such as MAELoss or MSELoss is a reasonable option. The drift and brake actions are boolean values, thus their prediction can be tackled as a binary classification task, making the Binary Cross Entropy loss a reasonable option (BCEWithLogitsLoss combines a Sigmoid layer and a Binary Cross Entropy loss).

Table 4: Loss functions used for training each action

Action	Output Function
Steer	MSELoss
Acceleration	MSELoss
Drift	BCEWithLogitsLoss
Brake	BCEWithLogitsLoss

Training time. We also used lazy loading for the dataset, meaning that the images were not all loaded into memory (our system did not have enough memory). Instead, we loaded into memory the paths for the dataset into and loaded each episode when it was going to be used. Using an NVIDIA RTX 2060, each epoch/episode took about 45 seconds to train (200 epochs take around 2.5 hours). Each epoch trains on a single episode due to memory constraints (batch size of 1). Each episode has a length of 600-700 timesteps.

[3]This step pretrains the CNN, which then continues training as the image embedding together with the rest of the model (the CNN weights are not frozen)

Learning rate warm-up. During training, the training loss decreased rapidly while the validation loss did not. This was solved by using a learning rate warm-up, which helped the DecisionTransformer perform well while using a reasonable learning rate so training time does not increase too much.

5.3 Hyperparameters

The controller's hyperparemeters come from the Decision Transformer and CNN. The Decision Transformer has been obtained from Huggingface[4]. We have built the CNN ourselves following an architecture with residual connections.

The hyperparameters have been selected using previous knowledge of values that tend to work well and some trial and error (mostly tuning the learning rate and the number of epochs). Table 5 presents some of the hyperparameters used for the models. More information about the hyperparameters can be found in the agents.cnn and the agents.decision_transformer files.

Table 5: Some of the hyperparameter values used for the controller

Hyperparameter	Value
Learning rate	1e-4
Epochs	300-400
Learning rate warm-up epochs	100
Hidden size	128
Max episode length	2000
Image input size	96,128
CNN output channels per block	16, 32, 64, 128
CNN convolutions per block	3
CNN stride per block	2
CNN kernel size	3
CNN residual connections	yes
CNN activation function	ReLU
Normalization	BatchNorm

5.4 Results

CNN autoencoder. Two CNN autoencoders have been trained, one transforming the

[4]See `https://huggingface.co/docs/transformers/model_doc/decision_transformer`

images to grayscale and another one using RGB images. In our experiments the RGB CNN works best. However, we think that the grayscale CNN could obtain better results when training the controller in multiple tracks at the same time.

Models. A summary of the controllers trained is presented in Table 6. The following controllers have been trained and evaluated:

- **NoDriftModel.** This model learns to steer. It is trained from scratch using the CNN pretrained encoder and the dataset portion with drift disabled.
- **DriftModel.** This model learns to steer and drift. It is trained on the full dataset by taking the NoDriftModel with best validation loss and training it further (continue training steer and additionally train drift)
- **FullModel.** This model learns to steer, drift, and accelerate. It is trained from scratch using the CNN pretrained encoder on the full dataset.

Table 6: Actions predicted by controllers

Model	Steer	Acceleration	Drift	Brake
NoDriftModel	Predicted	1	False	False
DriftModel	Predicted	1	Predicted	False
FullModel	Predicted	Predicted	Predicted	False

We have compared the controllers trained with the following set of already given controllers:

- **Baseline.** Aim point baseline controller used as coach to gather the dataset. This is the model from whose experience our controllers are learning
- **NoDriftBaseline.** Aim point baseline with drift disabled.
- **AI.** The game's AI. We have used the AI level 0, since levels 1 and 2 use objects, which have not been used for this experiment

Evaluation. To evaluate the controller we have to give the model a target reward which will be used by the Decision Transformer to make its predictions. The target reward to use depends on the track and the maximum reward achievable. We have used a target reward of 500 for lighthouse.

We have run each of the models 20 runs and obtained summary results. Fig. 1 presents the median number of steps obtained per model, and Fig. 2 presents the median rewards obtained. Table 7 presents more detailed statistics for the evaluation results. We can observe that the model with the lowest median number of steps is the DriftModel, which is also very consistent compared with the other models. It is followed by the FullModel, which has a lower median value and is less consistent but achieves better results in some runs. Both controllers surpass the game's AI and the baseline from whose experience they have learned.

Controller's performance. We have trained a controller that learns from a suboptimal policy to fully control a kart in the Super Tux Kart environment. The model learns to predict which steer, drift, and acceleration actions to take using only information accessible by the player's kart (image, and the current velocity and rotation). Moreover, the controller surpasses the coach policy used to gather its training data without using any other data. This is an essential difference between the Decision Transformer with respect to Imitation Learning. Our controller does not just imitate the coach's actions, but it improves on them. It is able to discern positive and negative patterns in the experiences it has seen and combine them to obtain a more optimal policy.

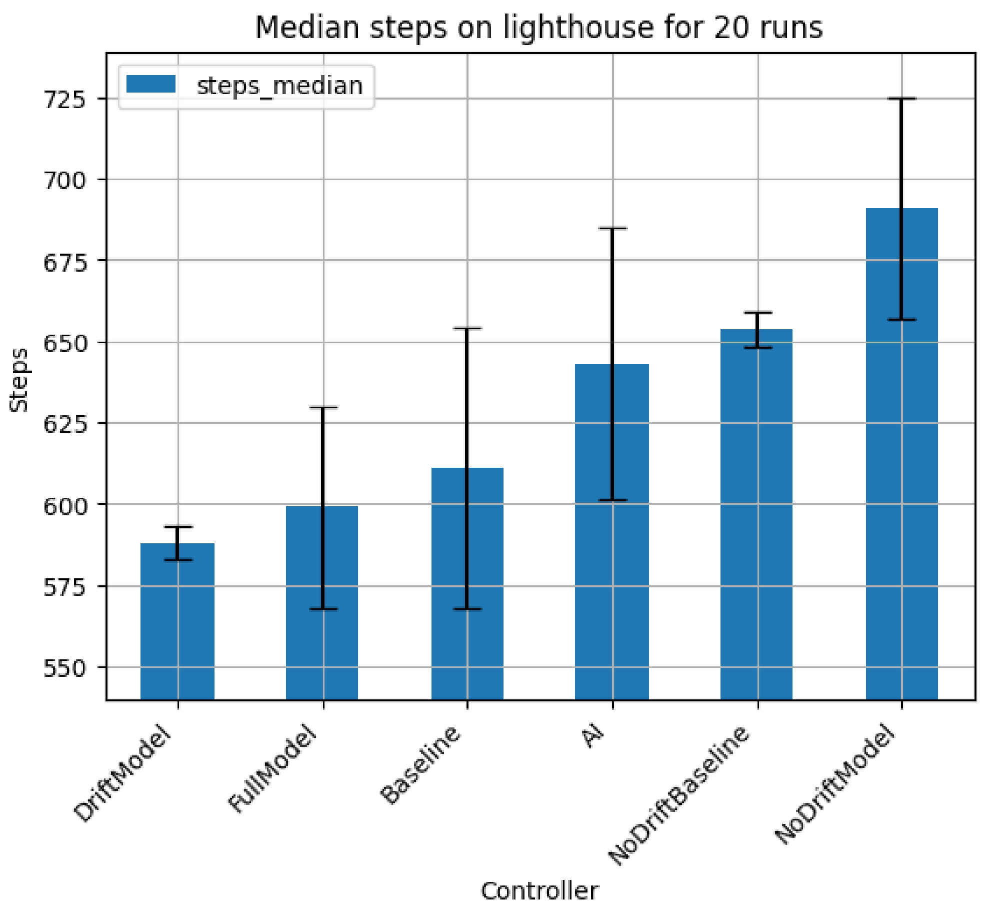
Table 7: Number of steps per model for 20 runs on lighthouse track

Models	Mean	Median	Std	Max	Min	Range
DriftModel	592.85	588	8.87	614	583	31
FullModel	603.8	599	27.34	681	568	113
Baseline	608.4	611	25.97	664	568	96
AI	637.45	643	18.74	676	601	75
NoDriftBaseline	660.65	653.5	14.05	693	648	45
NoDriftModel	681.7	691	17.44	704	657	47

Limitations. Although the controller can learn from a suboptimal policy, it is heavily reliant on the policy used for exploration. In order to learn that a certain action improves the policy, it needs to gather data taking that action. This is especially important to learn to take situational actions such as using objects, which are very rare due to the different objects available and the different situations they can be used in. This need for exploration can be mitigated by adding noise, using an exploratory policy, or using an augmentation approach such as DAGGER.

Another limitation of the model is the high quantity of data needed to train on. While using a Transformer increases the modeling capability, it also entails gathering a significant amount of data for training. This can be especially limiting to train in a real

Figure 1: Median number of steps for 20 runs on lighthouse track (error bars are the maximum and minimum values)

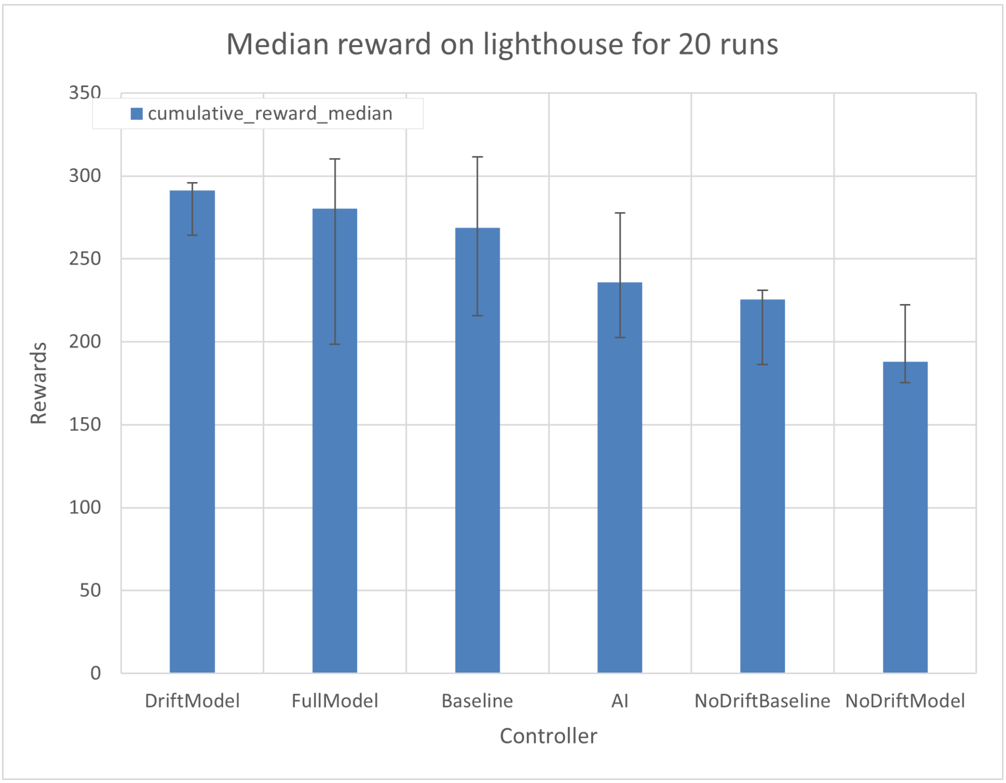


environment instead of a simulated one.

6 Conclusions and Future Work

Throughout this study, we have shown the capabilities of Decision Transformers in the racing simulator domain, specifically in the SuperTuxKart environment. As seen in the Results section (Sec. 5.4), the Full Model and Drift Model perform better than both the Aim Point Baseline and the game's AI driver. This is due to the model's ability to learn from random suboptimal policies by identifying positive and negative experiences and combining the information to take decisions. The resulting solution is a full controller

Figure 2: Median sum of rewards for 20 runs on lighthouse track (error bars are the maximum and minimum values)



(steer, drift, and acceleration) that can handle a complex high-dimensional state space of images, kart velocities, and kart rotations, as well as a continuous action space of steering and acceleration values with situational actions such as drifting.

Moving forward, the next steps will be to test our model against more baselines to prove the validity of our solution. We also hope to expand the scope of this project to include more SuperTuxKart game features, such as nitro boost and prize boxes, because Decision Transformers are capable of learning complex state spaces. Both of these features require item collection and specifically timed actions. Eventually, we would also like to train our model to race against other AI-driven players. This adds much more complexity due to the high level of stochasticity introduced by the other agents, but has been proven to be possible in works by Wurman, et al. [13].

References

[1] aistats11anon. *Comparison of Imitation Learning Approaches on Super Tux Kart.* Nov. 2010. URL: https://www.youtube.com/watch?v=V0OnpNnWzSU (visited on 11/05/2022) (cit. on p. 4).

[2] Marc G. Bellemare et al. "The Arcade Learning Environment: An Evaluation Platform for General Agents". In: *Journal of Artificial Intelligence Research* 47 (June 2013), pp. 253–279. ISSN: 1076-9757. DOI: 10.1613/jair.3912. arXiv: 1207.4708 [cs]. URL: http://arxiv.org/abs/1207.4708 (visited on 10/27/2022) (cit. on p. 6).

[3] Greg Brockman et al. *OpenAI Gym.* June 2016. arXiv: 1606.01540 [cs]. URL: http://arxiv.org/abs/1606.01540 (visited on 10/27/2022) (cit. on p. 6).

[4] Lili Chen et al. *Decision Transformer: Reinforcement Learning via Sequence Modeling.* June 2021. arXiv: 2106.01345 [cs]. URL: http://arxiv.org/abs/2106.01345 (visited on 09/19/2022) (cit. on pp. 2, 5, 8).

[5] Harrison Ho, Varun Ramesh, and Eduardo Torres Montano. "NeuralKart: A Real-Time Mario Kart 64 AI". In: (), p. 9. URL: http://cs231n.stanford.edu/reports/2017/pdfs/624.pdf (cit. on p. 6).

[6] Bangalore Ravi Kiran et al. "Deep Reinforcement Learning for Autonomous Driving: A Survey". In: *CoRR* abs/2002.00444 (2020). arXiv: 2002.00444. URL: https://arxiv.org/abs/2002.00444 (cit. on pp. 3, 4).

[7] Timothy P. Lillicrap et al. *Continuous Control with Deep Reinforcement Learning.* July 2019. arXiv: 1509.02971 [cs, stat]. URL: http://arxiv.org/abs/1509.02971 (visited on 10/27/2022) (cit. on p. 5).

[8] Haochen Liu et al. *Augmenting Reinforcement Learning with Transformer-based Scene Representation Learning for Decision-making of Autonomous Driving.* 2022. DOI: 10.48550/ARXIV.2208.12263. URL: https://arxiv.org/abs/2208.12263 (cit. on p. 3).

[9] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning.* Dec. 2013. DOI: 10.48550/arXiv.1312.5602. arXiv: 1312.5602 [cs]. URL: http://arxiv.org/abs/1312.5602 (visited on 10/26/2022) (cit. on p. 5).

[10] Błażej Osiński et al. "Simulation-Based Reinforcement Learning for Real-World Autonomous Driving". In: *2020 IEEE International Conference on Robotics and Automation (ICRA)*. May 2020, pp. 6411–6418. DOI: 10.1109/ICRA40945.2020.9196730 (cit. on p. 3).

[11] Stephane Ross, Geoffrey J. Gordon, and J. Andrew Bagnell. *A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning*. Mar. 2011. DOI: 10.48550/arXiv.1011.0686. arXiv: 1011.0686 [cs, stat]. URL: http://arxiv.org/abs/1011.0686 (visited on 10/26/2022) (cit. on p. 4).

[12] Sen Wang, Daoyuan Jia, and Xinshuo Weng. *Deep Reinforcement Learning for Autonomous Driving*. 2018. DOI: 10.48550/ARXIV.1811.11329. URL: https://arxiv.org/abs/1811.11329 (cit. on pp. 2, 3).

[13] Peter R. Wurman et al. *Outracing champion Gran Turismo Drivers with deep reinforcement learning*. Feb. 2022. URL: https://www.nature.com/articles/s41586-021-04357-7#citeas (cit. on p. 17).