Training Racing Cars using Deep Reinforcement Learning

Amina Tabassum[1] *† **, Siddhartha Dhar Choudhury**[2] †

[1] College of Engineering, Northeastern University
[2] Khoury College of Comptuer Sciences, Northeastern University
360 Huntington Avenue
Boston, Massachusetts 02115
tabassum.a@northeastern.edu, choudhury.si@northeastern.edu

Abstract

The idea of maneuvering a vehicle seems trivial to us, but mimicking the same in an agent is a tough problem mostly because of the enormous amount of possibilities that the agent has to generalize to. Various efforts to incorporate our knowledge of driving in form of computer algorithms have been carried out in the past without appreciable results. In recent times the advancement of deep learning and its application to reinforcement learning has led to promising results in achieving this goal owing to the representational power of neural networks to learn non-linear function mapping between a set of inputs and outputs. In this work, we explore two of these deep-reinforcement learning algorithms to train agents to drive in racetrack environments with varying levels of complexity. In order to teach these agents to control themselves in these environments we use two different ideas from standard reinforcement learning literature - Q learning and policy gradient methods and work with their deep neural network variants (Deep Q Network and REINFORCE). We further dive deep into the levels of learning (end-to-end v/s hierarchical) and the signals (image v/s numerical state) to find the approach that works best in such problems. In order to achieve our goal we also introduce a new simple configurable racetrack environment with tracks that allow plugging in agents trivially to control the cars.

Introduction

The promise of driverless cars (self-driving cars) has intrigued us for decades. The idea of a driverless car is to navigate through any possible situation while driving on the road, without having a human take control of the car. Many efforts have been made to achieve this goal in the past using a variety of algorithms ranging from basic knowledge-base-based agents to planning and learning agents with generalization capabilities. The field has been an active area of study and is still an unsolved problem owing to its level of difficulty[16]. In order to set achievable goals, researchers have set five different levels of self-driving, from L1 to L5, where L1 assists the driver by using sensors to help with the decision-making process more straightforwardly, and L5 is full automation where there is no need for a driver. [3]

It is common for us to learn to drive when we are under the age of 20 and it often does not take much time to learn to drive properly. This is mostly because we use our knowledge of the world around us to navigate various situations posed to us while driving in the real world. Unfortunately, this is not the case with self-driving cars, as they are built (or trained) for a single purpose and they have to learn all these intricate details while learning to drive. This makes solving the problem of self-driving cars an exceptionally challenging one. The number of possible scenarios is unimaginable and it is not possible to train the car in each of those innumerable situations.[5] There have been efforts to solve this problem by trying to come up with specific rules for the various situations, but those failed miserably, mostly because of the sheer amount of possible cases that a driving agent is posed in the real world. The problem thus needs an algorithm that can learn to generalize from the finite experience that it is trained on to those that were never seen before, much like how we use our existing knowledge to generalize on newer situations [9].

The idea of generalizing from some training examples is a popular trend in a field of Artificial Intelligence called Machine Learning. These algorithms (the most commonly used subset of them) work by taking a dataset of inputs and outputs and finding a relationship (or a function) between the two, which eliminates the need for coming up with rules to transform the inputs into required outputs. In recent times, a subset of this field called Deep Learning has emerged and it is now used to solve a wide variety of problems. The central idea of deep learning is an architecture called Artificial Neural Network (which in loose terms mocks the natural neural networks that we have). Neural networks have been proved to learn to approximate any function (linear or non-linear) given enough data containing input and output pairs to learn the relationship from. It does so by learning a set of parameters called weights in these neural network architectures using an algorithm called gradient descent. The greatest feature of this type of learning algorithm is that they require little to no feature engineering in order to learn a function and they do so in an end-to-end fashion without requiring to break the problem down into several smaller steps.

*These authors contributed equally.

†GitHub: https://github.com/frankhart2018/cs-5180-project

As deep neural networks are very good in learning approximate functions that map inputs to outputs, this fits perfectly in our problem with seemingly uncountable possible states (or situations) [16] These neural networks can be paired with existing techniques in reinforcement learning and can be used in control and sequential decision making problems. The most commonly used input to these neural networks are images from cameras in the car, which can then be mapped to a value or action to be taken to control the car. Current state of the art cars use data from additional sensors apart from the image from the onboard cameras. As these networks have good generalization guarantees this removes the chances of error in situations which have not been encountered during training. However, the plain artificial neural network implementations generally do not work well with problems of this size, thus other variants like Convolutional Neural Network (CNN), Recurrent Neural Network (RNN), and Long-Short Term Memory RNN (LSTM-RNN) are used most commonly to solve similar problems.

In this work we will be making use of some of the existing work on combining deep learning with reinforcement learning (into what is now called deep reinforcemnt learning). We will be using two specific algorithms - Deep Q Networks (DQN) and REINFORCE, the former is a deep RL version of Q-learning that replaces the Q-table with a non-linear function approximation (a deep neural network) that maps the set of inputs states to action value estimates that help define the policy at current time step and the second is a policy gradient method where the parameterized policy is represented as a deep neural network which also takes in the state as input and gives the actions to be taken in that particular state. Our focus is to compare these algorithms for the problem of teaching a car to drive in racetrack environments (both simple and complex). We further explore the possibilities of making use of previously learned knowledge in simpler environments to be re-used in complex environments by training the models in a hierarchical fashion. The other comparison is between the types of signals used for training these agents - numerical inputs v/s raw images representing the state of the car in the environment.

In order to achieve this we introduce a simple racetrack environment built using a similar API to OpenAI's Gym (thus allowing people to get started with this without having to go through the implementation in detail). This environment features configurable racetracks and allows users to add more tracks to test their models on using an easy to use web interface (application) to build, persist and edit new racetracks. It offers a simple API to plug agents in and provides useful visualization tools in order to quantify progress of the learning agent. It also allows the user to make use of either numerical states or get the image at the current time step representing the car in the track environment.

Background

The basic idea behind reinforcement learning is that agent learns on it's own either from experience/trial and error or from model of the environment. It is always comprised of an agent that interacts with environment and returns rewards for taking certain actions[10]. It is widely used in robot manipulations and controls these days.Several solutions have been proposed for roboti manipulation problems but all of these use specialized policy representations and human demonstrations[1].Therefore, researchers focused on using deep reinforcement learning to embed control policy with efficient perceptual representations. Autonomous vehicles is one of the high potential research fields in computer vision , controls and reinforcement learning is also playing great role to achieve level 5 autonomy. Deep Q-networks are applied for autonomous navigation and obstacle avoidance of cars. [6]

Several built in racing car environments are used to solve racing car problems. OpenAI gym is one of the famous open reinforcement learning environment and development tools.[14] It is used as standard benchmark for solving several RL problems because of ease of use. However, we developed our own custom environment comprising of 4 different racetracks. Both state and action space are discrete. State space as discussed in environment dynamic is comprised of road, grass and soil.Whereas, action space is comprised of five discrete actions: Noop,Move left, Move right, Gas and Brake.

Racing car problem has been solved using different reinforcement algorithms such as PPO, DQN , REINFORCE etc.[15]. Some people have also tried to solve it using supervised learning algorithms in simulated environments i.e. CARLA. Racing car training models are used to ensure that car stays on track, avoid collisions with obstacles, pedestrians and other vehicles and control speed.[17] So, in general racing car training model should have following features:

- Adaptive control
- Road/ grass and soil recognition
- Vehicle recognition
- Pedestrian detection etc.

Everyone is trying to address above mentioned problems using different deep learning , reinforcement learning and deep reinforcement learning algorithms. Solving these problems will be a revolution in robotics and especially manipulation and controls domain.In this paper, we will discuss hour proposed solution to control racing car using deep reinforcement learning.

Related Work

Autonomous agents are widely used in several research areas such as Robotics, AI and controls. They are comprised of following basic requirements: Detection, prediction , planning and controls. An agent should detect objects in it's surroundings , predict information based on it's detected information. Planning involves using previous information

and take an optimal action that will maximize reward and hence enable agent reach goal state [2]
[17] discusses solving 2D racing game using reinforcement learning and supervised learning.They created their own environment and used open AI gym environment as well. For self-made environment, reward was +1 when vehicle did not crash and episode ended when car crashed. DQN and REINFORCE was used and model learned very quickly to drive around track. They also trained agent using self supervised learning but results were better using DQN and REINFORCE.
[12] uses DDQN and open AI Gym environment. They modeled a reward system and experimented with a lot of hyper-parameters , model architectures and input image processing to achieve q-value of 1000 after 18,000 trials. They used Keras for implementation and Adam Optimizer with learning rate starting at 0.00001.Four different model architectures were used. Initially, for the basic model, the image was cropped from 96x96 pixels to 50x50 pixel around the car and RGB was converted to gray scale and 10 frames per state were used. To process input frames, smaller CNN model was used and then output of CNN was passed to DDQN. Maximum predicted Q-value achieved using this model was 400 after 21000 iterations. The training speed was improved from 21000 iterations to 9000 iterations using same CNN architecture but with increased number of filters. In this way, different architectures were tried, different hyper-parameters were tuned and hence best model was selected.

[4] uses deepRacer as a platform and they demonstrate how a 1/18th scale car can learn to drive autonomously using RL with a monocular camera. This car is trained using Sim2Real and demonstrates robust reinforcement learning, distributed on-demand computer architecture and robust evaluation method to identify when agent should stop training. In this paper, they have used PPO algorithm to train DeepRacer on multiple tracks.They have achieved sim3real in real track with ¡5 minutes of trainig at slow speeds and 1.6 m/s using moels trained with tuned hyperparameters.
[8] uses TORCS for simulation and implemented two different algorithms: SAC-LSTM and DQN. In this case, SAC and SAC-LSTM performed better than Rainbow DQN. One of the possible reasons can be the continuous action space and exploration methods of two algorithms. SAC maximizes entropy and hence allows the agent to explore uncertain regions of action space. On the other hand, DQN uses discrete action space and hence it's performance was relatively lower than SAC.
[18] discusses the problem of implementing deep reinforcement learning algorithms in real world problems.In this paper, they have used policy gradient algorithm DDPG to handle complex state space and continuous action space(which is case of real world problems). They chose Open Racing Car Simulator (TORCS) as an environment. They designed their own network architecture for both acor and critic inside DDPG to fit DDPG algorithm to TORCS.

[11] discusses DDQN convergence rate and control performance when applied to highway decision making strategies.[7] also discusses DRL to for racing tracks. [13] disucsses how policy gradient methods are used to solve high dimensional problems such as legged robots and manipulators.

Project Description

In this work we train agents to drive in race track environments of varying complexities. The agent is the only vehicle in the track and it does not learn to compete against other cars that may be present in the track. In order to train the agent we have used two different algorithms - Deep Q Network (DQN) and REINFORCE. Our aim is to compare the learning from both these algorithms, and compare the effectiveness in this simple self-driving car task. We chose these algorithms as they are the simplest algorithms in their category and allows for better comparison between the two class of algorithms - Q Learning (DQN) and Policy Gradient (REINFORCE) methods.

We have considered two different criteria to compare these algorithms - the signal (also referred to as the state) that the agent receives at every time step and the style of learning: end-to-end v/s bootstrapping.

Criteria 1: The Signal

The signal is the input that the agent receives (or in the case of agents powered by deep neural networks, the signal is the input to the neural network) that allows the agent to perceive the environment and learn some policy (actions to take in these states). We have considered two different types of signals - numerical inputs and pixels from the actual track with the car and the surroundings in it. Since, the environments can be arbitrarily large, we confine the agent's view of the environment around it to a certain distance that we call "eye sight" (η). The agent's vision is thus confined to a square of shape $2\eta + 1 \times 2\eta + 1$, this is made clear in Figure 1.

Numerical Inputs The environment is represented as an array ($\mathbb{Z}^{x \times y}$, where x is the height of track and y is the width) in memory and every element (car or surroundings) is represented as a single integer value in this two dimensional array. There are four types of patch that each location in this array can represent - soil (1), grass (2), road (3) and car or the agent (4), and all the values which are out of bounds of the track are padded with 0. At each time step a patch around the car is selected based on the value of η (of shape $\eta \times \eta$) and this is then flattened and returned to the agent as a signal. In order to work with this kind of signal we have used a deep neural network architecture called Multi-Layered Perceptron (MLP) which takes the flattened numerical values as inputs.

Pixels This again selects values around the agent based on the value of η. As each patch is a number in the range of 8-bit integer (used in 8-bit images) we can directly use these values (0, 1, 2, 3, and 4) as a signal, this time without having to flatten the two-dimensional patch. The signal in this case

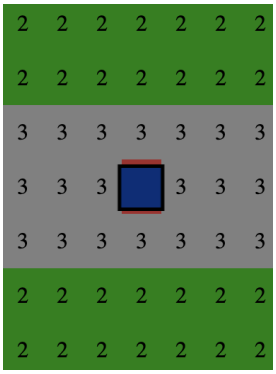


Figure 1: Eye sight defines the area around the agent (the blue patch in this figure) that the agent can perceive. This is an example of a state when the eyesight is three, the agent can thus perceive states within a distance of three in all eight directions. In this, the patches with value 3 (gray-colored patches) represent the road, and the patches with value 2 (green-colored patches) represent the grass.

is thus a single channel image of dimensions $\eta \times \eta$. Since the signal is of image type, we have to use a different architecture to process it, this is called Convolutional Neural Network (CNN).

Criteria 2: Style of learning

We have considered two styles of learning: end-to-end and bootstrapping. The end-to-end version learns in every track it is trained on from scratch. This means that it starts with a fresh neural network with randomly initialized weights every time it sees a new environment during training. On the other hand, the bootstrapped version is trained by using weights from the previous training in a different environment. In order to properly bootstrap we start training with the simplest environments (straight line) and progress towards more complicated environments (starting with simple turns and going on to learn to drive in complicated loop tracks). This thus ensures that the agent can re-use knowledge from the previous environment and hence takes a shorter amount of time to converge to a decent policy (as will be made more clear from the plots in the results section). Figure 2 provides a summary of all the permutations of models.

Environment Dynamics

Before moving on to the details of the algorithms it is important for us to know about the environment dynamics.

States The previous sub-section covers the details of the state - numerical inputs v/s pixels each of which is a real number of shape $\mathbb{Z}^{(2\eta+1)^2}$ in case of the former and $\mathbb{Z}^{2\eta+1\times 2\eta+1}$ in case of the latter.

Actions At each time step the agent can take five different actions:

1. NOOP: This moves the agent forward with the current speed.
2. LEFT: Changes the cars steering angle to a fixed amount in radians (θ) in the negative direction.
3. RIGHT: Changes the cars steering angle to a fixed amount in radians (θ) in the positive direction.
4. GAS: Increases the current speed and moves the car forward with the updated current speed, the change in speed is dictated by the following equation:

$$v_t = v_{t-1} + \delta_{gas} * \alpha$$

$$\delta_{gas} = t - t_{gas}$$

   Where v_x is the velocity at time x, α is the constant with which speed increases every time, t is the current time step and t_{gas} is the first time step since when the agent is taking the GAS action continuously (i.e. the more the agent takes this action the greater the speed). The max speed is capped by v_{max} which is a constant.
5. BRAKE: Decreases the current speed and moves the car forward with the updated current speed using the above mentioned equation (the only difference being a difference with the previous time steps velocity instead of addition).

Hence, the action is chosen from one of $\mathbb{Z}^5$.

Rewards At each time step the agent receives a reward dictated by the following equation:

$$c_{i,j} + \mathbb{1}_{|a_{old}|=1} * r_{same} + \mathbb{1}_{s \in S^G} * r_{win}$$

$$c_{i,j} = \begin{cases} 1 & \text{(i, j) = ROAD patch} \\ -3 & \text{(i, j) = GRASS patch} \\ -5 & \text{(i, j) = SOIL patch} \end{cases}$$

Where a_{old} is a set of the last five actions taken by the agent in the environment, and r_{same} is the constant negative reward that is given to the agent for taking the same action in the last five time steps. This extra term is to force exploration early on during training. S^G is the set of goal states (all the patches that make up the finish line), and r_{win} is the reward for completing the track (i.e. passing through the finish line)

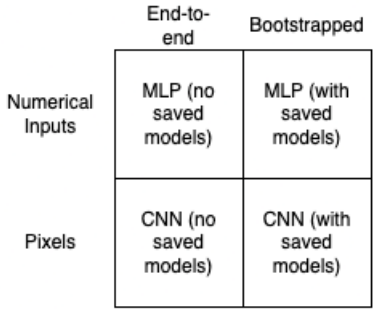


Figure 2: A summary of all possible permutations of models trained using the two criteria mentioned above. The end-to-end trained models do not require saving models, but the bootstrapped version does require that, as they use the model from previous training in the current track as the complexity grows

Task type This is an episodic task, with an episode ending when the agent reaches the finish line (which means it successfully completed a lap in the track).

Now we have sufficient knowledge of the environment dynamics, and can move to the algorithms:

Deep Q-Network

DQN is a deep neural network which takes states as input and outputs the Q-values of actions. When agent interacts with environment, the experience tuple can be highly correlated and Q-learning selects the experiences with maximum Q-value so bias will also be higher. In order to solve this problem, we store experience tuples and randomly select small batches of data and hence avoid the q-values from diverging or oscillating.

We have used two different neural network architectures:

- Multi Layered Perceptron (MLP) based network
- Convolutional Neural Network (CNN)

MLP

MLP based network has an input layer of dimensions $(2\eta + 1)^2$, two hidden layers of dimension (128) and output layer of five action nodes. All layers are activated using Rectified Linear Unit (ReLU) and algorithm is trained using Adam (Adaptive Moment Estimation) optimizer.

CNN

The CNN network has input layer that transforms single channel input to 3 channels followed by one more hidden convolutional layer with 5 as the output channel. This is then flattened and passed to a linear hidden layer of dimensions $(5 * 2\eta - 1 * 2\eta - 1$, and finally to the output layer with five actions nodes . We have activated all layers by Rectified Linear Unit (ReLU) activation followed by softmax for output layer.

Training

We have used Adam optimizer and mean square error loss function to train algorithm. We have used following equations to compute loss function and update weights of neural network:

$$w \leftarrow w - \alpha \nabla_w Ł(s, a, s'; w)$$

$$\nabla_w Ł(s, a, s'; w) \approx -(r + \gamma \max_a Q_w(s', a') - Q_w(s, a) \nabla_w Q_w(s, a))$$

REINFORCE

As discussed earlier, the REINFORCE algorithm is driven by a deep neural network that takes the state at the current time step (that the agent experiences) and maps it to an action to be taken on the current time step. In our case since, we have five different actions, all of our neural networks have a output layer of size five.

We have defined two different neural network architectures - a MLP based network and a CNN based network. The same network is used in both end-to-end and bootstrapped training versions of the agent.

MLP The Multi Layered Perceptron (MLP) has an input layer of dimension $(2\eta + 1)^2$, two hidden layers of fixed dimension (128) and an output layer consisting of five nodes. Each of these hidden layers are activated using Rectified Linear Unit (ReLU) functions. The final output layer is activated using softmax function which allows the model to maintain an epsilon greedy policy (as opposed to hardmax function) before the agent converges to a good enough policy.

CNN The Convolutional Neural Network (CNN) has an input layer that transforms the single channel input into 3 channels, this is followed by one more hidden convolutional layer with 5 as the output channel. This is then flattened and passed to a linear hidden layer of dimensions $(5 * 2\eta - 1 * 2\eta - 1$, found empirically) and finally to the output layer with five nodes (as above). We have used batch normalization in between the convolution layers followed by Leaky-ReLU activation for all the hidden layers (the linear hidden layer is not followed by batch norm as expected). Finally, the output layer is again activated using a softmax function.

Training This algorithm is trained using Adam (Adaptive Moment Estimation) optimizer and a Negative Log Likelihood (NLL) loss function. The training is dictated by the following equations:

$$\theta \leftarrow \theta + \alpha\gamma^t G \nabla ln\pi(A_t|S_t, \theta)$$

$$G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$$

Where θ represents the parameters of the model (NN), α is the learning rate, γ is the discounting factor, G is the return, R_x is the reward at time step x, and π is the current policy.

Experiments

We have trained our agents using two different algorithms: Deep Q Networks (DQN) and REINFORCE. For each of these techniques, we trained our agents on three different racetrack environments. Due to computational constraints, we could only train the agents that learned good enough policies for the numerical inputs but could not learn even decent policies using image input (due to the small-sized CNNs that we trained). Another choice to cut on the computational cost is choosing a maximum number of time steps that an agent can experience in a single episode in this environment. Our experimentation is divided into four different models (summary in Figure 2) for each of these three tracks resulting in twelve different results overall.

Tracks

As mentioned, we have used three different tracks (Figure 3), the training is done in the sequence mentioned below:

1. Simple: This is a 5 x 10 (H x W) environment where the agent starts at index [1, 2] (based on array-indexing) and has to reach the goal state at any of [1, 6], [2, 6], or [3, 6]. The goal of this environment is to teach the agent to learn to drive straight on the road without deviating. We have made it so small by choice, as we wanted the agents trained on this track to generalize to a longer straight-track environment. The test environment for this is shown in Figure 4.
2. Ellipse: This is an elliptical-shaped environment. This introduces the agents to the concept of taking turns. The agent starts just after the finish line and has to move in an anti-clockwise direction till it reaches the goal state.
3. Circle: This is the toughest environment, as the agent has to continuously take turns in this and it has to do so without getting out of the track as much as possible. The agent again starts just after the goal line and has to move in anti-clockwise direction till it reaches the goal state.

Environment Parameters

The environment parameters refer to the choice of reward and the constants that decide the dynamics of the environment, the following table (Table 1) summarizes these:

Parameter name	Value
Steering angle	0.6 radians
Alpha (change in speed)	0.5
Base speed	1.0
Max speed	5
Max time steps	1000
On win reward	50
On road reward	1
On grass reward	-3
On soil reward	-5
Same action reward	-10

Table 1: Table of environment parameters

Training

The MLP models are kept the same for both end-to-end training and bootstrapped version of the learning schemes. The only difference between these two is the saving and loading of models before training on each track. For the end-to-end training, we start with a new neural network with randomly initialized weights every time, whereas for the bootstrapped version the model the weights are loaded from the previous iteration of training and only the first model (i.e. the one trained on the simple environment Figure 3a) get's randomly initialized weights.

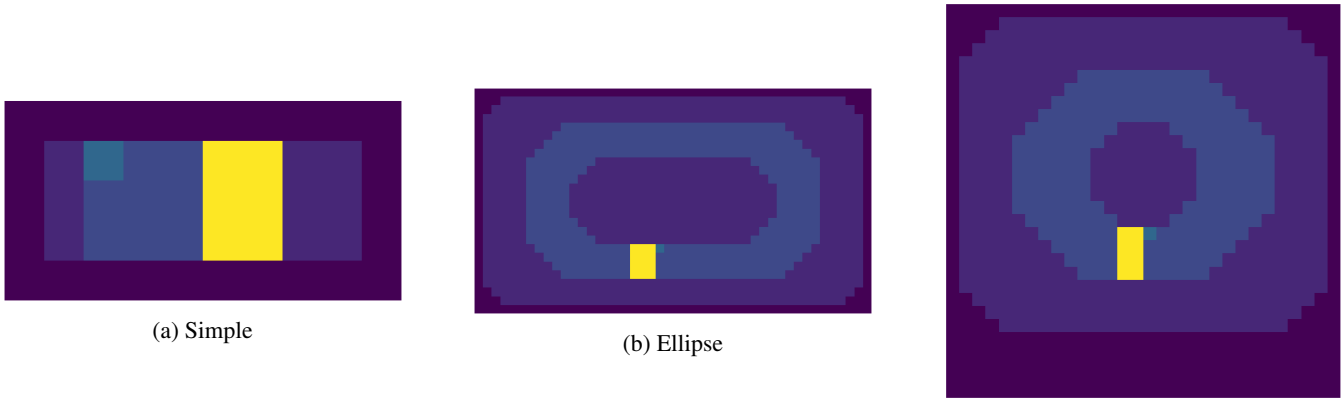
Bootstrapped model The order in which the agent is trained on these three environments does not matter at all for end-to-end training as it learns everything from scratch, but in the case of bootstrapping it does matter a lot. We feel that the order of difficulty arises with the number of turns to make and hence have ordered them as: simple (Figure 3a), ellipse (Figure 3b) and circle (Figure 3c).

Learning to drive in a straight path is the most basic of operations that an agent can learn, hence we also present the agent with a separate test environment (Figure 4), and only if it learns to drive in this testing path correctly, we move on to the subsequent models.

Deep Q-Network

As mentioned before, based on agent's eye sight, input layer receives numerical input followed by two hidden layers of dimensions 128 followed by output layer of dimensions 5. Table 2 shows the model architecture and Table 4 shows the hyperparameters used.

Table 4 demonstrates different hyperparameters tuned for different tracks.We are implementing both end-to-end and bootstrapped training variants based on learning scheme. It took 200 time steps for simple track, 5000 time steps for elliptical and circular track and 7000 time steps for bootstrapped version. Initially, we started from higher epsilon value (exploration factor) i.e. 1 and then reduced it to 0.01. Update frequency of target policy is 200 and update frequency of behavior policy is 40.



(a) Simple

(b) Ellipse

(c) Circle

Figure 3: The tracks that we are using to train our agents

Figure 4: This is a much longer environment to test driving in a straight line

REINFORCE

As mentioned before, we have two different variants of this training based on the learning scheme: end-to-end and bootstrapped version. Both of these share the same model architecture. The input layer gets a flattened version of the numerical inputs based on agent's "eye sight" (η) and is of dimension $(2 * \eta + 1)^2$ which is followed by two hidden layers each of which has 128 nodes and finally there is an output node with five nodes representing the five actions that the agent can take at each time step. Table 2 shows the model architecture in form of a table and Table 3 shows the list of hyperparameters used.

The hyperparameters table shows the values for the number of episodes of training for the end-to-end version on different tracks and the combined number of episodes for the bootstrapped version. A simple hack that we used in order to save us the trouble of saving and loading models for the bootstrapped version is to swap the training environment after a certain number of steps. The total number of time steps it took for the model to learn all three environments is 910. During this time it spent 200 episodes on the simple track, the next 304 episodes on the ellipse track, and the final 406 episodes on the circle track.

Results for DQN

The results for DQN for three different tracks (straight path, circle and elliptical track) are summarized in Figure 5 (end-to-end training) and Figure 6 (bootstrapped training). For straight and elliptical track, trend is pretty obvious that

Layer Name	Dimension (num nodes)
Input layer	$(2\eta + 1)^2$
ReLU	N/A
Hidden layer 1	128
ReLU	N/A
Hidden layer 2	128
ReLU	N/A
Output layer	5
Softmax	N/A

Table 2: Table of MLP model architecture

Hyperparameter	Value
Episodes (in simple track)	200
Episodes (in ellipse track)	500
Episodes (in circular track)	820
Episodes (bootstrapped)	910
Learning rate	1e-2
Discount factor	0.99
Epsilon	1e-5
Eye sight	2

Table 3: Table of hyperparameters for REINFORCE

Hyperparameter	Value
Episodes (in simple track)	220
Episodes (in ellipse track)	310
Episodes (in circular track)	450
Episodes (bootstrapped)	700
Learning rate	1e-3
Discount factor	0.99995
Epsilon start value	1
Epsilon end value	1e-2
Epsilon duration	250
Replay buffer size	100,000
Frequency of behaviour policy update	40
Frequency of target policy update	200
Batch size	32
Eye sight	2

Table 4: Table of hyperparameters for DQN

agent is exploring in initial episodes and once it has learnt about the environment,the average return increases.For circular track, there is suddent drop in reward. This is because, at this point , the agent lost track and entered into grass. Hence, the average return reduced because of negative reward. However, the agent recovered even after getting off the road hence, average return again increases. For bootstrapping training, the trick we used is to switch track after certain number of episodes and observe if agent explores new path or just exploits previous knowledge about the environment.

With reference to Figure 6, initially the track is straight path. There is sudden dip in average return after 150 and 250 episodes. This is because of switching environments and agent could not immediately exploit it's previous knowledge so, average return decreases. Figure 5a clearly demonstrates that agent took 150 episodes to learn about the environment. So, we switched environment after 170 episodes from straight track to elliptical. Agent took some time to transfer it's knowledge about the environment and then it uses it's previous information and gives average return that it demonstrated for elliptical path in Figure 5b.Since, it quickly learned this environment . We switched environment after 300 episodes to circular track and again there is dip in average return. It is important to notice that it utilised bootstrapped information more efficiently in elliptical track than circular one. Another important point to notice is sudden drop on average return after 780 episodes. We observed this trend in Figure 5c as well. This is because, our agent is not perfectly trained for circular path and it gets off the track. However , it still manages to recover path. We need to tune hyperparameters of architectures to make it more efficient.

Results for REINFORCE The results of REINFORCE are summarized in Figure 7 (end-to-end training) and Figure 8 (bootstrapped training). A trend that we see in the bootstrapped version is the sudden drop in the average returns, this is because of the switching of tracks and the agent's inability to transfer the knowledge when there is a sudden change in the environment. As we see that after 200 episodes and 504 episodes there is a considerable dip in the average returns, and then as the agent learns in these environments the average returns increases. The peaks in Figure 8 show the maximum average value that the agent achieved in each of these environments.

It is pretty evident that the agent performs better with the bootstrapped version (Figure 8), with fewer amount of training for each of the tracks it is able to perform pretty well. It is seen that for the ellipse and circle environments the bootstrapped agent actually performed better than the end-to-end trained equivalents, this is mostly because it was able to re-use its parameters from the previous tracks in the current track. The less number of down spikes in both ellipse and circle shows the fact that it didn't have to spend a lot of time re-learning how to take complicated turns as it did in the end-to-end training version (Figure 7b and Figure 7c).

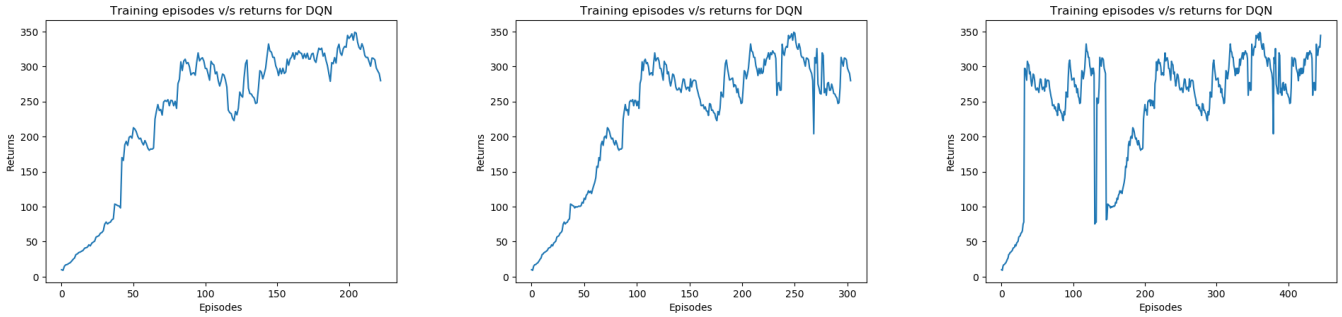
Conclusion

It is evident from our experimentation that in this use case, policy gradient methods (REINFORCE) perform better than Q-learning algorithms (DQN). The agent was able to learn faster and achieve better average returns at the end of the training with REINFORCE. This holds true for both types of training schemes that we used: end-to-end and bootstrapping.

As far as the learning scheme is concerned it is shown that the bootstrapped version performs better than end-to-end learning, especially in terms of the time required to learn a decent enough episode in complex environments (with multiple turns). The agent is thus able to re-use the information from the previous training and generalize on the more complex environments. On the other hand, in the case of end-to-end training, the agent has to learn the intricacies of the environment from scratch every time and spends some time learning to drive straight before learning about more complex actions like making turns.

Future Work

We plan to extend our work by training the CNN models and using the much more complicated loop tracks that we have defined in our environment. We also intend to focus on more sophisticated policy gradient algorithms like Advantage Actor-Critic (A2C) and Deep Deterministic Policy Gradients. Another possible extension to this work could be to make the environment more realistic by converting both state and action spaces to continuous values instead of discrete steps.



(a) Simple (b) Ellipse (c) Circle

Figure 5: DQN results - average returns over the entire training episode

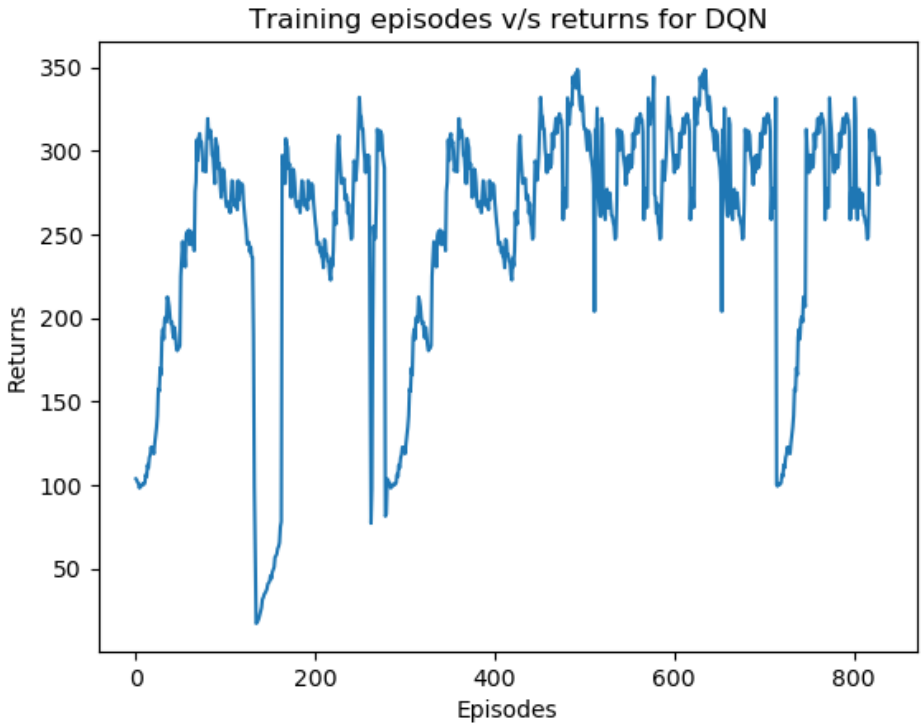
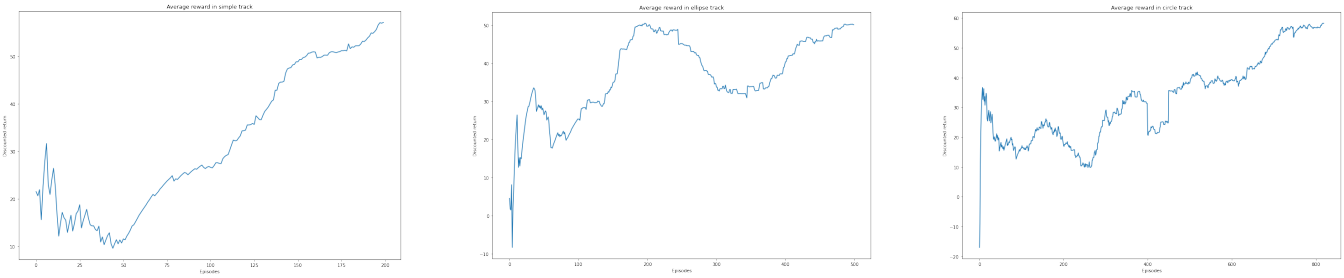


Figure 6: DQN results-Bootstrapping



(a) Simple (b) Ellipse (c) Circle

Figure 7: REINFORCE results - average returns over the entire training episode

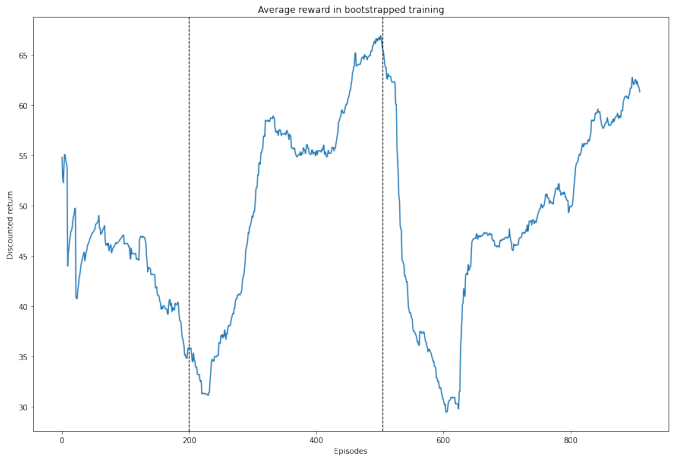


Figure 8: REINFORCE results - bootstrapped learning, the dashed lines represent the points when the environment was switched. The first dashed line (at 200 episodes) marks the end of simple track and switches to ellipse, the second dashed line (at 504 episodes) marks the end of ellipse track and the swapping with circle track

References

[1] Amarjyoti, S. 2017. Deep Reinforcement Learning for Robotic Manipulation-The state of the art.

[2] Aradi, S. 2022. Survey of Deep Reinforcement Learning for Motion Planning of Autonomous Vehicles. *IEEE Transactions on Intelligent Transportation Systems*, 23(2): 740–759.

[3] Badue, C.; Guidolini, R.; Carneiro, R. V.; Azevedo, P.; Cardoso, V. B.; Forechi, A.; Jesus, L.; Berriel, R.; Paixão, T. M.; Mutz, F.; de Paula Veronese, L.; Oliveira-Santos, T.; and De Souza, A. F. 2021. Self-driving cars: A survey. *Expert Systems with Applications*, 165: 113816.

[4] Balaji, B.; Mallya, S.; Genc, S.; Gupta, S.; Dirac, L.; Khare, V.; Roy, G.; Sun, T.; Tao, Y.; Townsend, B.; Calleja, E.; Muralidhara, S.; and Karuppasamy, D. 2020. DeepRacer: Autonomous Racing Platform for Experimentation with Sim2Real Reinforcement Learning. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, 2746–2754.

[5] Daily, M.; Medasani, S.; Behringer, R.; and Trivedi, M. 2017. Self-Driving Cars. *Computer*, 50(12): 18–23.

[6] Fayjie, A. R.; Hossain, S.; Oualid, D.; and Lee, D.-J. 2018. Driverless Car: Autonomous Driving Using Deep Reinforcement Learning in Urban Environment. In *2018 15th International Conference on Ubiquitous Robots (UR)*, 896–901.

[7] Folkers, A.; Rick, M.; and Büskens, C. 2019. Controlling an Autonomous Vehicle with Deep Reinforcement Learning. In *2019 IEEE Intelligent Vehicles Symposium (IV)*, 2025–2031.

[8] Güçkıran, K.; and Bolat, B. 2019. Autonomous Car Racing in Simulation Environment Using Deep Reinforcement Learning. In *2019 Innovations in Intelligent Systems and Applications Conference (ASYU)*, 1–6.

[9] Kim, J.; Lim, G.; Kim, Y.; Kim, B.; and Bae, C. 2019. Deep Learning Algorithm using Virtual Environment Data for Self-driving Car. In *2019 International Conference on Artificial Intelligence in Information and Communication (ICAIIC)*, 444–448.

[10] Li, Y. 2017. Deep Reinforcement Learning: An Overview.

[11] Liao, J.; Liu, T.; Tang, X.; Mu, X.; Huang, B.; and Cao, D. 2020. Decision-Making Strategy on Highway for Autonomous Vehicles Using Deep Reinforcement Learning. *IEEE Access*, 8: 177804–177814.

[12] Maniyar, Y.; and Manoj, N. 2017. 2Racetrack Navigation on OpenAIGym with Deep Reiforcement Learning. 1–7.

[13] Peters, J.; and Schaal, S. 2006. Policy Gradient Methods for Robotics. In *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2219–2225.

[14] Ramesh, N.; and Mourya, S. ???? Reinforcement Learning using the CarRacing-v0 environment from OpenAI Gym.

[15] Ravichandiran, S. 2020. *Deep Reinforcement Learning with Python: Master classic RL, deep RL, distributional RL, inverse RL, and more with OpenAI Gym and TensorFlow*. Packt Publishing Ltd.

[16] Remonda, A.; Krebs, S.; Veas, E.; Luzhnica, G.; and Kern, R. 2021. Formula rl: Deep reinforcement learning for autonomous racing using telemetry data. *arXiv preprint arXiv:2104.11106*.

[17] Tejar, H.; Storozev, M.; and Saks, J. 2017. 2D Racing game using reinforcement learning and supervised learning. 1–10.

[18] Wang, S.; Jia, D.; and Weng, X. 2018. Deep Reinforcement Learning for Autonomous Driving.