An Optimized DeepRacer Model

Group 18:
Yongkun Deng 480041153 Yeming Chen 480517728 Yanhao Xu 490147935 HangYu Chen 490075948 Riley Vaughan 490421097

***Abstract*—A DeepRacer model can be optimized by using up-scaled waypoints to reward the model for maintaining a racing line, by ensuring that it has a decision space which allows high-speed as well as hairpin turns, by cloning and retraining the model, and by iterating experimentation to determine ideal hyper-parameters.**

***Index Terms*—Amazon DeepRacer, neural networks, way-points, Huber-loss, re-training, deep learning.**

I. Introduction

THIS document details the construction and performance of our group's best model, which was submitted by Yongkun under the name ”Zero-dayGame0018”. With a time of 2:15.813 on the Po-chun Super Speedway, at the time of writing, **it is in first place** on the leaderboard for our class, with only one off-track.

We will first detail the race record, the reward function, the hyper-parameters and the training process used for our best model. Then we will discuss the differences between PPO and SAC policies, before considering each of the improvements that our project model made to a general model.

II. Project Model

A. Race Record

Our model is currently in first place on the final race leaderboard, with a time of 2:15.813, which is 7.779 seconds ahead of the second place time. This was Yongkun's submission. Other group members have submitted different or similar models to the final track leaderboard. A screenshot of our position can be seen in Appendix A.

B. Reward Function

The entire code listing of our reward function can be found in the appendix A and in an attachment to this report. An in-depth explanation of each module of our reward function is contained below. In general, our reward function calculates the nearest waypoint to the car's position, and provides a score based off the difference between the car's current heading and the ideal heading.

1) dist(point1, point2): This helper function takes the two Cartesian co-ordinates as input and returns the Euclidean distance between them using the formula:

$$\text{distance} = \sqrt{(p_{1x} - p_{2x})^2 + (p_{1y} - p_{2y})^2} \quad (1)$$

2) rect(r, theta): This helper function converts Polar co-ordinates of a radius and an angle into Cartesian co-ordinates using the following pair of functions:

$$x = r * \cos(\theta), y = r * \sin(\theta) \quad (2)$$

3) polar(x, y): This helper function acts as the inverse of the previous one, converting Cartesian co-ordinates into polar form, using these equations:

$$r = \sqrt{(x^2 + y^2)}, \theta = \arctan \frac{y}{x} \quad (3)$$

4) angle_mod_360(a): This helper function maps an angle into the interval $[-180, 180]$. For example, using the input 362 would produce an output of 2, while an input of 270 would produce an output of -90. It does so using the following equations:

$$n_0 = \frac{a}{360}, n_1 = \text{a} - n_0 * 360 \quad (4)$$

If n_1 is below 180 degrees, then it is returned. If not, then $n_1 - 360$ degrees is returned.

5) get_waypoints_ordered_in_driving_direction(params): This helper function is used to reverse the entries in the waypoint array if the car is driving clockwise. Po-Chun Super Speedway involves driving counter-clockwise, so this isn't used.

6) up_sample(waypoints, factor=n): This helper function interpolates additional waypoints, upscaling their resolution by the input factor. It uses a list comprehension to generate an output which has n times as many entries as the original waypoints list. For each point p_j in the list of waypoints which is of length l, it generates i additional points n_i in the form $[n_{ix}, n_{iy}]$ using this formula:

$$n_{ix} = \frac{i}{n} * p[(j+1)\%l][0] + (1 - \frac{i}{n}) * p[j][0] \quad (5)$$

$$n_{iy} = \frac{i}{n} * p[(j+1)\%l][1] + (1 - \frac{i}{n}) * p[j][1] \quad (6)$$

7) get_target_point(params): This module first upsamples all of the correctly ordered waypoints by a factor of 8. Next it calculates the distance of every waypoint from the car, and reorders the list of waypoints according to how close they are. It then calculates a distance r as 90% of the track width, and finds that first waypoint with a distance that is greater than r, which it drives towards. That waypoint is used as an optimal direction for the car.

8) get_target_steering_degree(params): This module runs the previous module, and calculates the required heading for the car as angle between the car's current position $[x, y]$ and the target waypoint $[t_x, t_y]$. It does so by using the polar() helper function, and then converting the angle into a direction in the range $[-180, 180]$ using the angle_mod_360() helper function. The output is a simple heading.

9) score_steer_to_point_ahead(params): This module runs the previous module which outputs the desired heading for the car θ. It compares the current steering angle of the car to the optimal angle, and produces an error:

$$\text{error} = (steering\_angle - \theta)/60 \tag{7}$$

The score used for our reward function is then simply:

$$\text{score} = \max \begin{cases} 1 - \text{error} \\ 0.01 \end{cases} \tag{8}$$

The reason for the limit in this final step is that the optimizer used in SageMaker performs poorly when the reward function outputs negative numbers or numbers too close to zero.

10) reward_function(params): This wrapper function simply calls the previous function and returns the score as a float.

C. Hyperparameters

1) Action Space: Continuous: [0.5, 4]m/s [-30, 30] degrees

The action space is the set of permitted actions for the car to take at any point. DeepRacer allows for the selection of either a discrete or a continuous action space: a discrete action space contains a predefined and limited set of movements, whereas the continuous action space asks the user to set upper and lower bounds for both speed and heading, and permits the complete range of combined actions between those bounds.

Our research found that most implementations of DeepRacer use discrete action spaces that carefully accommodate for a particular track. However, after experimenting with both discrete and continuous action spaces, we decided to use a continuous action space, as it yielded the best results on Po-Chun. We allowed for the full range of headings, [-30, 30], as it helped our model manage around the hairpin turns. We decided upon an abridged range of velocities, [0.5, 4], because we wanted our model to be able to maximise its speed on straight portions of tracks while being able to slow down while cornering. We found that a minimum speed of 0.7 caused too many off-tracks on sharp turns, while a minimum speed of 0.3 caused our model to be significantly slower.

2) Policy Method: Proximal Policy Optimization

For the reasons discussed in section III we chose PPO as our policy method. To briefly summarise, it allowed us to experiment with a wider range of reward functions while maintaining stability and consistent results. While we could have used SAC to attempt to get faster convergences, PPO simply offered more consistent results.

3) Gradient Descent Batch Size: 64

Batch size is the number of training samples in one forward/backward progress, and it is used to speed up the training process. When we use batch size, the inputs will be a series of entries rather than one single entry. The weights will be updated after one batch of data is trained. So, the small batch size will update the weights more frequently and provide more detailed information to the network. However, it will cause a large uncertainty in training, and the learning curve will not be stable. The large batch size takes more samples in one forward/backward progress, which can reduce the computation cost and make the training process more stable and smooth, but it may also cause the loss of features, increasing the memory used during the training. A proper value of batch size will improve efficiency and accuracy.

The batch size we used in our project is 64. It is neither too large nor too small, as our group wants to get convergence within 2 hours. Small batch sizes take a long time to converge, and a large batch size will increase the computation time. Setting the batch size to 64 can help our model converge in a short time while making the training process more stable.

4) Entropy: 0.01

Entropy is the degree of uncertainty during the training process. In DeepRacer, the entropy provides randomness to the selection of action space. The higher value of entropy, the larger degree of uncertainty. This uncertainty and randomness can help the agent find the global optimum rather than the local optimum. The large entropy value will encourage the agent to explore the action space, but it will make the model hard to converge and significantly increase the training time.

The entropy we used in our project is 0.01. The learning progress is very sensitive to the entropy value. Slightly increasing the entropy value makes our model extremely hard to converge. However, the zero entropy value will also prevent us from finding the global optimum, so we use 0.01 as our entropy value to maximise the performance.

5) Discount Factor: 0.5

The discount factor determines how much the future result will affect the current reward. It is highly related to the real-time performance of the reward function. In DeepRacer, the vehicle should consider the future results to decide the current action. The vehicle is expected to achieve continuous rewards rather than discrete rewards. The discount factor is used to help the vehicle achieve long and continuous rewards. However, the large discount factor will significantly increase the training time. And a proper discount factor can improve the performance of the model.

The discount factor used in our project is 0.5. For this project, we want our model achieves the best performance in a short time. The large discount factor will slow down the training process, and a too small discount factor will reduce the training quality. Therefore, our group decided to use 0.5 to achieve the best results in two hours of training

6) Loss Type: Huber

The loss function calculates the difference between the network output and the gourd truth. And the loss value will be used for backpropagation. We have two types of loss functions in DeepRacer: Mean Squared Error and Huber. The function for mean squared error is:

$$\sum_{i=1}^{D} (x_i - y_i)^2 \tag{9}$$

Where x_i is the ith output, and y_i is the ith ground truth. The MSE is quadratic for all loss values. The Huber loss function can be expanded to the following form:

$$L_\delta = \begin{cases} \frac{1}{2}(y-\hat{y})^2 & if\,|(y-\hat{y})| < \delta \\ \delta((y-\hat{y}) - \frac{1}{2}\delta) & otherwise \end{cases} \tag{10}$$

Where the y is the ground truth, the $\hat{y}$ is the output from the model. The δ is the threshold. From the function above, we can observe that the loss is squared for a small value and linear for a large value. This feature allows the Huber loss function to be less sensitive to large updates. In contrast, for the MSE, all kinds of updates are considered equally. Based on the features of these two loss functions, the Huber can be helpful if the model is hard to converge. The Huber will be more focused on good outputs, which can help the model converge faster. But the Huber function is also more complex than the MSE, so if the convergence is good, the MSE can be used to improve the efficiency.

We used the Huber loss function in our project. During the experiment, our group found that the model was hard to converge during the training and the batch size we used was slightly smaller than the standard value. So, the results returned from the model are not very stable. We decided to use the Huber loss function to help the convergence of our model. The Huber loss function will be less sensitive to large changes, so we believe it can address our problem.

7) Learning Rate: 0.0003

The nature of the neural network is gradient descent. And the learning rate controls how much contribution of gradient descent to the existing neural network weights. The small value of the learning rate will help find the local optimum, but it will significantly increase the convergence time and prevent us from finding the global optimum. The large learning rate will cause the model hard to converge. So, a proper learning rate value can reduce the training time and lead us to a good result.

The learning rate we used in this project is 0.0003. The value is a little bit small at the beginning of the training. However, after one hour of training, the 0.0003 learning rate is still powerful for the training. The large learning rate will be helpful at the beginning but will also prevent us from finding the optimal point. The small learning rate will make the training process more stable, but it will also slow down the progress. Setting the learning rate to 0.0003 can ensure both accuracy and efficiency.

8) Number of experience episodes between each policy-updating iteration: 20

The experience episode is the period between the vehicle reset point and the off-track point. The number of experience episodes between each policy-updating iteration determines the weight updated frequency. The agent can learn the whole path in a few resets for simple tracks. So, a small number of experience episodes is enough for the training, and the training process will be fast. However, for the complex tracks, the small number of experience episodes will make the vehicle update the weight when it has not collected enough information. This kind of update is useless. A large number of experience episodes allows the vehicle to have many resets to learn the track. So, the number of experience episodes between each policy-updating iteration is associated with the difficulties of the tracks.

We used the Po-Chun Super Speedway as our training track. It is a long and challenging track. Our group set the number of experience episodes to 20 to improve the training quality. During the training, the vehicle will get off track easily, and the small number of experience episodes will make a lot of useless updates to the model. So our group decided to use 20 experience episodes to ensure training efficiency.

9) Number of Epochs: 10

The number of epochs is the number of turns the whole training dataset passes through the neural network. It determines the training length. The larger number of epochs will lead to a longer training time. And training the network in a large number of epochs will help the model learn more information from the input dataset, and the model will be more stable. However, it may also cause an overfitting problem and significantly extend the training time. Moreover, the number of epochs is also associated with the batch size. A larger batch size can reduce the training time but may cause the loss of some information, and a small batch size takes longer time to train, but it can collect more details of the training data, so it is a good idea to use small epochs and small batch size or large epochs and large batch size.

We trained our model in 10 epochs. The batch size we used in this project is 64, which is smaller than the standard value, so our group decided to use small epochs to achieve the best performance. The small batch size increased the training time for one epoch, but it also provided more information, so we used the small number of epochs to ensure the overall training time and performance.

D. Training Process

During the training process, the most significant factor to evaluate the model performance in the training graph is the average percentage completion (red line). It indicates the distance percentage in the whole track completed before the car gets off track during the evaluation process. As our first training trial, we aim to create a basic model with a general framework of the training, so we used a larger batch size and learning rate and reduced epochs number to ensure the model can train faster. Meanwhile, with a higher entropy the model can have a larger probability to explore by itself, which means the model will be more general and suitable for most situations. Figure 1 below shows the result of such a training approach.

To improve the model, we make a small adjustment to the model hyperparameter so that the car can be suitable for driving through continuous bends. We use a smaller batch size so that each training will focus on a shorter track and a smaller learning rate to ensure convergence. But in Figure 2, there are some zero points for average percentage training, the reason for those is when DeepRacer car enters the hairpin turns, the car will always be off track. Meanwhile, Figure 2 also illustrates the overfitting of the model with a tendency to converge. So we decided to change our reward function to improve the final performance.

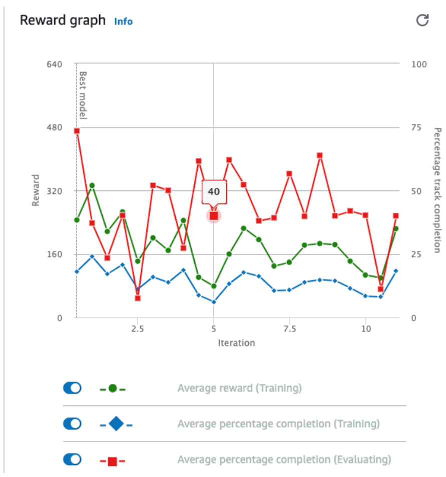


Fig. 1: Training process 1

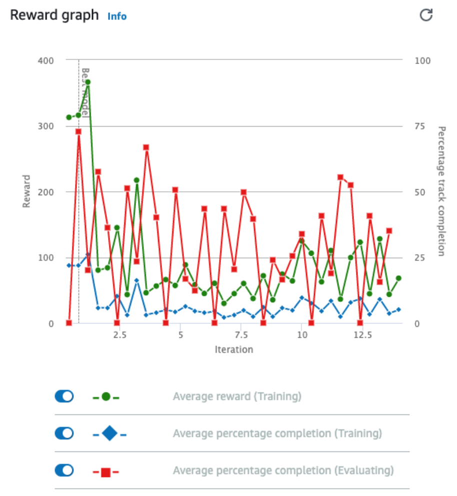


Fig. 2: Training process 2

After changing the logic of the reward function, we can see that the training is improved apparently (Figure 3). We set a smaller discount factor to ensure the driving style. Since it is the first trial of new reward function, the result could be a little bit lower, but it can be improved by further training. To ensure fast training with more iterations we set a greater value for the learning rate and reduced the gradient descent batch size.

To improve the previous model, we still made some minor adjustments to ensure the car completion of the track and narrow the sharp decrease of the previous graph. A larger gradient descent batch size and a smaller learning rate are used for improvement to achieve stable training and shorten the overall training time. Figure 4 demonstrates a much more stable result compared to the previous one.

III. Analysis of PPO and SAC

A. General Introduction

The mainstream of reinforcement learning can be divided into two approaches: on-policy methods and off-policy methods. Off-policy methods evaluate and/or improve a separate

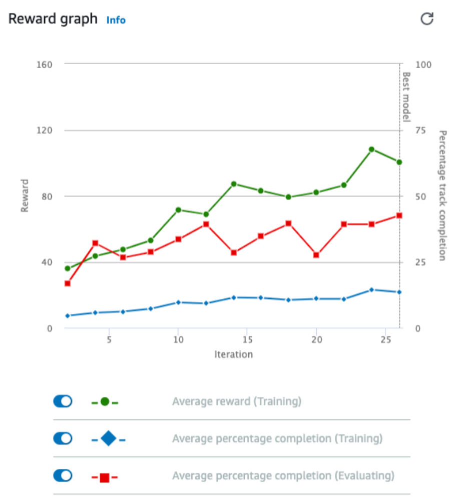


Fig. 3: Training process of our final reward function.

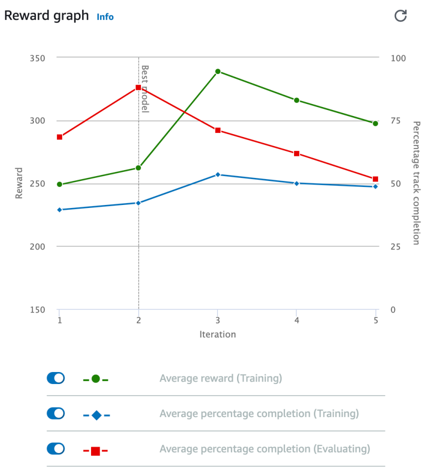


Fig. 4: Training process of our final reward function with a smaller learning rate and a larger batch size.

policy from the one used to create the data, in contrast to on-policy methods, which aim to evaluate or improve the policy that is used to make decisions [1]. Due to this feature, the agent in on-policy methods always pledges to explore and seeks for the best possible policy that still engages in exploration, whereas the agent in off-policy methods learns a deterministic optimal policy from the exploration which can be different or unrelated to the policy followed [1].

Proximal Policy Optimization (PPO) [2] is a recently proposed on-policy policy gradient method in reinforcement learning. PPO is an improved and simplified version of Trust Region Policy Optimization (TRPO) [3]. One of the major improvements of PPO is the enhancement of importance sampling. Importance sampling is a widely used technique in reinforcement learning that can convert on-policy methods into partially off-policy to alleviate the need for extensive sampling. Compared to TRPO, PPO proposed two methods to further increase the sample efficiency of importance sampling: clipping and KL penalty. Clipping reduces the probability ratio of policy updates to form a pessimistic bound of the performance of the policy, whereas KL penalty adds penalties on KL divergence to limit excessive policy updates of the model [2]. PPO also reports that clipping achieves better performance compared to KL penalty, and clipping is more commonly used in modern applications.

Soft Actor-Critic (SAC) [4] is another widely used approach in reinforcement learning. SAC leverages the high sample efficiency of off-policy methods and adds entropy to the maximization goal to increase the range of exploration. In addition, SAC uses actor-critic algorithm [5] with stochastic actor in its updating algorithm. Actor-critic algorithm combines the advantages of actor-only method (i.e., only relies on policy-based functions) and critic-only method (i.e., only relies on value-based functions). Actor-critic algorithm is mainly used in on-policy policy gradient methods, which is usually more stable than off-policy methods [6]. SAC utilizes this property to improve the stability of its algorithm.

B. Differences between PPO and SAC

PPO is an on-policy method and SAC is an off-policy method, which means PPO is more stable than SAC, but has a lower sample efficiency compared to SAC. The difference between on-policy method and off-policy method is discussed in details in section III-A. In reinforcement learning, exploration and exploitation are two fundamental and essential elements. [7] defines exploration as search for new knowledge and exploitation as continuously use of previously obtained knowledge to improve the performance. There is a trade-off between exploration and exploitation. Too high exploitation will lead to insufficient learning of the environment, which may result in getting stuck in a local optimum or even worse. In contrast, high exploration may find the global optimum, but can significantly increase the overall training time, which is unacceptable in time-limited situations. To ensure the balance between exploration and exploitation, PPO leverages entropy regularization which encourages exploration; hence the algorithm is more likely to find the global optimum. SAC utilizes entropy maximization to achieve a similar result. In addition, compared to entropy regularization, entropy maximization is more likely to abandon policies that opt for unpromising behavior. This further increases the sample efficiency of SAC which usually leads to faster convergence compared to PPO. However, SAC is more sensitive to the reward function, particularly the reward scale [4]. Large reward scale leads to insufficient exploration and may get stuck in a local optimum, whereas small reward scale result in not exploiting the reward, and causes severe performance degradation. In contrast, PPO is more robust to different reward functions. This is also the main reason why we chose PPO as the final algorithm. We have tested PPO and SAC in DeepRacer model. The results are shown in Figure 5 and 6.

We use the same settings for PPO and SAC (e.g., the same hyperparameters, reward function). It is obvious that due to the high reward scale, SAC hardly explore the environment. Even we set the SAC α value to 1 (i.e., the maximum value that prefers exploration), SAC still stuck in local optima for a long

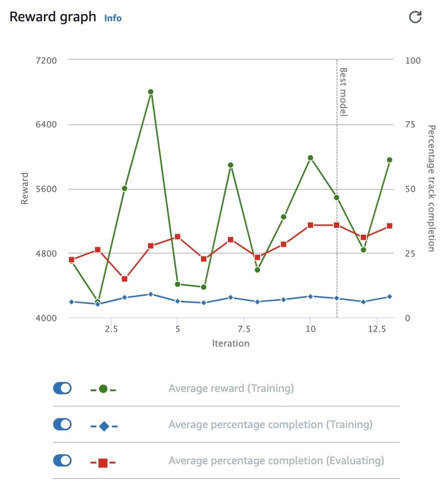


Fig. 5: Training plot of PPO algorithm

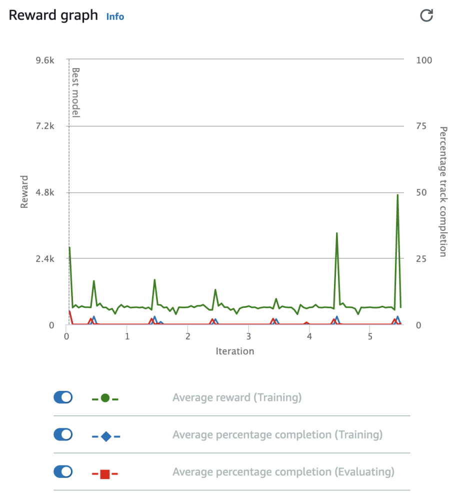


Fig. 6: Training plot of SAC algorithm

time. Conversely, PPO keeps exploring the environment which is more likely to find the global optimum. Finally, according to the AWS DeepRacer official documents, PPO is applicable for both discrete and continuous action space, and SAC is only applicable for continuous action space. We believe this is due to the fact that implementing discrete actions for SAC is not an easy task compared to PPO. However, there are also some variants of SAC that can be used with discrete action space, such as [8].

IV. Improvements to a General Model

A. Path decision of the car

In the first approach, we encourage the car to stay near the centerline of the track at all times and help the car to speed up/down based on the direction of the path. It is done by:

1) Minimize the difference between the heading of the car and the current direction of the path, shown in Fig 7.
2) Add future steps to detect if there will be a corner in the future so that the car could have the ability to “foresee” the future path, shown in Fig 8.

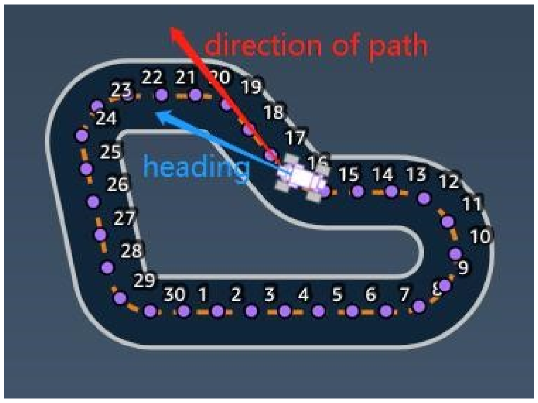


Fig. 7: Path Selection 1

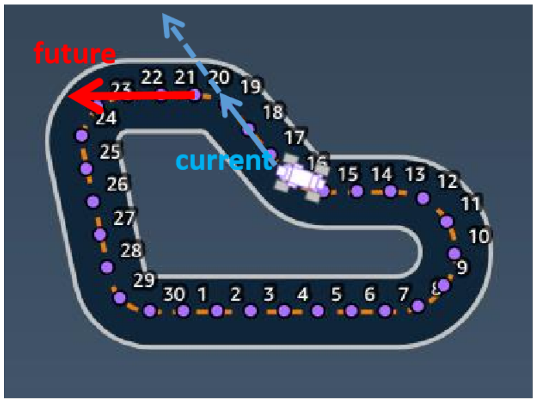


Fig. 8: Path Selection 2

The main drawback for this approach is that although it could help the car to adjust the speed when there is a corner, the car has to make a big steering angle when passing a sharp turn because we encourage the car to stay at the centre line all the time. So even if we trained for a long time, there is still a high probability for the car to be off track.

As for improvement, we made the car as the center of a circle with radius r and calculated the intersection of that circle with the centre line of the path, shown in Fig 9. Then the car aims and travels at the intersection (future way-point) directly, and the path is equivalent to making a tangent line [9]. This method used the similar idea as the first approach, but we have to add more way-points (up-sample function) and need to adjust the radius of the circle.

In this case, the car no longer needs to travel on the center-line every time, but could pass the corner in a tangent line. It is an improvement because it could not only check the direction of the future path and reduce the probability for the car to make a large steering angle, but also could reduce the total distance which the car needs to travel.

B. Rewards setting

In the first approach, rather than multiplying the sub-rewards, we decided to encourage the car to do the right thing by "adding sub-rewards". The reason is that if the previous sub-reward is close to zero, the model will not care about

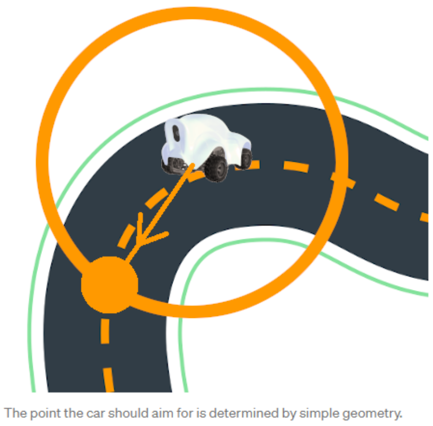

The point the car should aim for is determined by simple geometry.

Fig. 9: Path Selection 3

improving the other sub-rewards when using the multiplication approach [10].

But even if we choose to add the sub-rewards, it only provides some kind of average rating for the last couple of frames that lead the car to the place it is now. And it is hard to choose the weights of each sub-reward (e.g. It is hard to decide which one is more important: stay on the center line or speed up).

As for the improvement, we only care about the direction of the car because pointing to the correct direction is the most direct method to teach the car so that it could immediately tell the learning algorithm if the current steering direction is good or not. And there is no averaging over multiple frames needed.

C. Model clone and Learning rate adjustment

The most important indicator in the reward graph is the average percentage completion during evaluation (the red points in the reward graph), which represents how far the car can progress before being off track during evaluation runs.

In the early training process, we tried to make the learning fast by using a larger learning rate so that the car could prioritize exploring the action space. But after training the model and improving the performance for two hours, our focus tended towards making minor adjustments to the driving style to improve the reliability, which means try to stabilize learning. It is done by cloning the model (and maintaining the weights) previous trained and decreasing the learning rate from 0.0003 to 0.00003. After that, we continue training the model for 2 hours. Those two training graphs are shown in Figure 3 and 4.

D. Discount factor trade-off

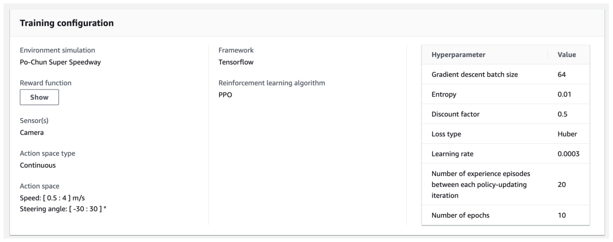
As we have mentioned previously, the discount factor determines how much the future result will affect the current reward. The network with a larger discount factor would need a longer time to converge, but a smaller discount factor will reduce the training quality which means we have to make a trade-off between these two values.

In this case, we decided to use 0.5 as the discount factor rather than 0.999, so that our model could have better performance in a limited time (less than 4 hours).

APPENDIX

```python
import math

def dist(point1, point2):
    return ((point1[0] - point2[0]) ** 2 + (point1[1] - point2[1]) ** 2) ** 0.5

# thanks to https://stackoverflow.com/questions/20924085/python-conversion-between-coordinates
def rect(r, theta):
    """
    theta in degrees
    returns tuple; (float, float); (x,y)
    """
    x = r * math.cos(math.radians(theta))
    y = r * math.sin(math.radians(theta))
    return x, y

# thanks to https://stackoverflow.com/questions/20924085/python-conversion-between-coordinates
def polar(x, y):
    """
    returns r, theta(degrees)
    """
    r = (x ** 2 + y ** 2) ** 0.5
    theta = math.degrees(math.atan2(y, x))
    return r, theta

def angle_mod_360(angle):
    """
    Maps an angle to the interval -180, +180.
    Examples:
    angle_mod_360(362) == 2
    angle_mod_360(270) == -90
    :param angle: angle in degree
    :return: angle in degree. Between -180 and +180
    """

    n = math.floor(angle / 360.0)
    angle_between_0_and_360 = angle - n * 360.0
    if angle_between_0_and_360 <= 180.0:
        return angle_between_0_and_360
    else:
        return angle_between_0_and_360 - 360


def get_waypoints_ordered_in_driving_direction(params):
    # waypoints are always provided in counter clock wise order
    if params["is_reversed"]:  # driving clock wise.
        return list(reversed(params["waypoints"]))
    else:  # driving counter clock wise.
        return params["waypoints"]


def up_sample(waypoints, factor=10):
    """
    Adds extra waypoints in between provided waypoints
    :param waypoints:
    :param factor: integer. E.g. 3 means that the resulting list has 3 times as many points.
    :return:
    """
    p = waypoints
    n = len(p)
    return [
        [
            i / factor * p[int((j + 1) % n)][0] + (1 - i / factor) * p[j][0],
            i / factor * p[int((j + 1) % n)][1] + (1 - i / factor) * p[j][1],
        ]
        for j in range(n)
        for i in range(factor)
    ]


def get_target_point(params):
    waypoints = up_sample(get_waypoints_ordered_in_driving_direction(params), 8)

    car = [params["x"], params["y"]]
    distances = [dist(p, car) for p in waypoints]
    min_dist = min(distances)
    i_closest = distances.index(min_dist)
    n = len(waypoints)

    waypoints_starting_with_closest = [waypoints[(i + i_closest) % n] for i in range(n)]

    r = params["track_width"] * 0.9

    is_inside = [dist(p, car) < r for p in waypoints_starting_with_closest]
    i_first_outside = is_inside.index(False)

    if i_first_outside < 0:
        # this can only happen if we choose r as big as the entire track
        return waypoints[i_closest]

    return waypoints_starting_with_closest[i_first_outside]


def get_target_steering_degree(params):
    tx, ty = get_target_point(params)
    car_x = params["x"]
    car_y = params["y"]
    dx = tx - car_x
    dy = ty - car_y
    heading = params["heading"]

    _, target_angle = polar(dx, dy)

    steering_angle = target_angle - heading

    return angle_mod_360(steering_angle)


def score_steer_to_point_ahead(params):
    best_stearing_angle = get_target_steering_degree(params)
    steering_angle = params["steering_angle"]

    error = (
        steering_angle - best_stearing_angle
    ) / 60.0  # 60 degree is already really bad

    score = 1.0 - abs(error)

    return max(
        score, 0.01
    )  # optimizer is rumored to struggle with negative numbers and numbers too close to zero


def reward_function(params):
    return float(score_steer_to_point_ahead(params))
```

Listing 1: Reward Function code.



Training configuration

Environment simulation
Po-Chun Super Speedway

Reward function

Sensor(s)
Camera

Action space type
Continuous

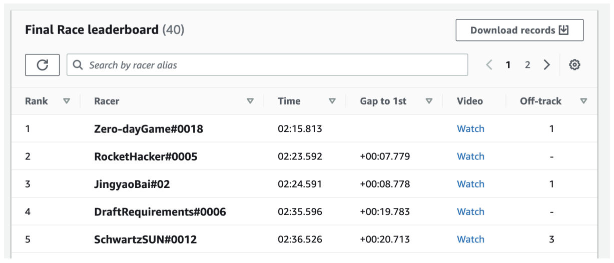
Action space
Speed: [0.5 : 4] m/s
Steering angle: [-30 : 30] °

Framework
Tensorflow

Reinforcement learning algorithm
PPO

Hyperparameter	Value
Gradient descent batch size	64
Entropy	0.01
Discount factor	0.5
Loss type	Huber
Learning rate	0.0003
Number of experience episodes between each policy-updating iteration	20
Number of epochs	10

Fig. 10: Hyperparameters



Final Race leaderboard (40)

Rank	Racer	Time	Gap to 1st	Video	Off-track
1	Zero-dayGame#0018	02:15.813		Watch	1
2	RocketHacker#0005	02:23.592	+00:07.779	Watch	-
3	JingyaoBai#02	02:24.591	+00:08.778	Watch	1
4	DraftRequirements#0006	02:35.596	+00:19.783	Watch	-
5	SchwartzSUN#0012	02:36.526	+00:20.713	Watch	3

Fig. 11: Final leaderboard.

References

[1] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[2] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[3] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *International conference on machine learning*. PMLR, 2015, pp. 1889–1897.

[4] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," in *International conference on machine learning*. PMLR, 2018, pp. 1861–1870.

[5] V. Konda and J. Tsitsiklis, "Actor-critic algorithms," in *Advances in Neural Information Processing Systems*, S. Solla, T. Leen, and K. Müller, Eds., vol. 12. MIT Press, 1999. [Online]. Available: https://proceedings.neurips.cc/paper/1999/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf

[6] J. Peters and S. Schaal, "Reinforcement learning of motor skills with policy gradients," *Neural networks*, vol. 21, no. 4, pp. 682–697, 2008.

[7] F. Vermeulen and H. Barkema, "Learning through acquisitions," *Academy of Management journal*, vol. 44, no. 3, pp. 457–476, 2001.

[8] P. Christodoulou, "Soft actor-critic for discrete action settings," *CoRR*, vol. abs/1910.07207, 2019. [Online]. Available: http://arxiv.org/abs/1910.07207

[9] F. Tandetzky. AWS Deepracer — How to train a model in 15 minutes. (2019, Dec 17). [Online]. Available: https://medium.com/twodigits/aws-deepracer-how-to-train-a-model-in-15-minutes-3a0dca1175fb

[10] D. Gonzalez. An Advanced Guide to AWS DeepRacer. (2020, Jun 15). [Online]. Available: https://towardsdatascience.com/an-advanced-guide-to-aws-deepracer-2b462c37eea