This member-only story is on us. Upgrade to access all of Medium.

✨ Member-only story

An Advanced Guide to AWS DeepRacer

Autonomous Formula 1 Racing using Reinforcement Learning

Daniel Gonzalez · Follow
Published in Towards Data Science
16 min read · Jun 15, 2020

Listen Share ••• More

Photo by chuttersnap on Unsplash

Self-driving cars have become a hot field in recent years, with companies such as Tesla pushing the boundary of technology every day. AWS's DeepRacer is making use of this hype, becoming more and more popular and even organizing a league to compete in.

In May of 2020, AWS organized a special event, in which it partnered with Formula 1. The track for this event was the highly complex Circuit de Barcelona-Catalunya. In the time trial category, our team managed to become 12th place, out of nearly 1300 participants.

In this article, we will look at the factors that got our university team to a top 1% ranking in the AWS DeepRacer F1 time trial event. So if you are interested in seeing the advanced techniques that go into training a reinforcement learning model in AWS DeepRacer, this is the right article for you.

We will be covering the following points:

1. A Short Introduction to AWS DeepRacer and our Setup
2. Computing the Optimal Racing Line and Speed
3. Optimizing the Action Space
4. The Reward Function
5. Hyperparameters
6. Continuous Improvement with Log Analysis
7. Automated Race Submissions with Selenium
8. Summary & Next Steps

To follow this article, you do not need an extensive data science background. In fact, our team has a business background, studying Business Analytics at ESADE Business School in Barcelona, Spain. However, basic knowledge about Python is needed to understand this article.

AWS F1 Promo Video

dgnzlz/Capstone_AWS_DeepRacer

Code that was used in the Article “An Advanced Guide to AWS DeepRacer”

github.com

1. A Short Introduction to AWS DeepRacer and our Setup

AWS DeepRacer is a 1/18th scale autonomous racing car that can be trained with reinforcement learning. The model can be trained and managed in the AWS console using a virtual car and tracks. When using the AWS console, the entire infrastructure, including the training of the model and the virtualization of the racing tracks, is managed by AWS.

In contrast to classical machine learning, reinforcement learning is used when you have no data, but an environment from which the agent can learn. In our case, the agent is the car, and the environment is the virtual racing track. By giving the agent rewards for desired actions, the agent learns over time to master the problem in the given environment. To learn more about reinforcement learning, check out this article from my team member Marc Cervera:

Explaining Reinforcement Learning for Beginners based on AWS DeepRacer

High-level explanation of how Reinforcement Learning works with neural networks in autonomous racing

towardsdatascience.com

DeepRacer was built specifically as a learning product for people to learn about machine learning. 3 components play an important role in making its mission a success: the virtual training environment, the physical car, and the league. If you want to learn more about DeepRacer, feel free to visit the official website.

In this article, we will focus on the use of the AWS DeepRacer console, so customization with AWS SageMaker will not be covered. Additionally, only virtual racing will be examined as physical racing requires a different approach. Lastly, we will only consider the time trial format. However, the majority of the described approach can also be used for other racing formats.

DeepRacer's virtual environment (image by author)

2. Computing the Optimal Racing Line and Speed

In virtual races, overfitting the model to the specific track is a way to achieve a good model with an acceptable amount of training. Therefore, to achieve better times and converge faster towards higher speeds with more reliability, we will use a prescriptive approach, in which we will make the car follow the optimal racing line for that track.

To compute the racing line, we will use the K1999 Path-Optimization Algorithm described in Rémi Coulom's Ph.D. Thesis. The algorithm has already been implemented in this GitHub Repo. It works by iteratively decreasing the line's curvature, leading the car to cut curves, decreasing the overall path length. All DeepRacer tracks can be downloaded in the DeepRacer Community's GitHub Repo.

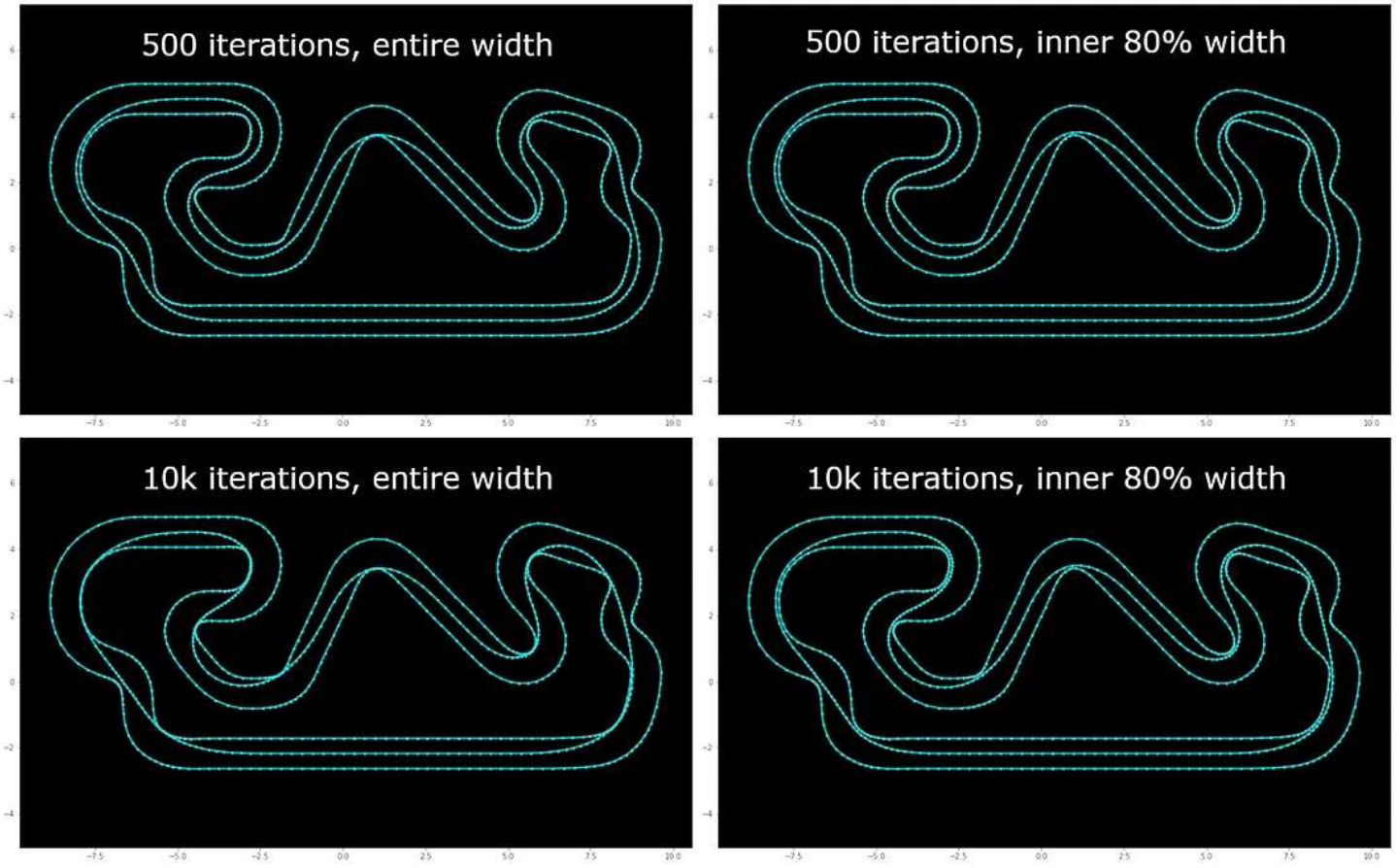
For the F1 track, the result is an array of 258 non-equally spaced coordinate points, which represent the racing line.

Compared to other DeepRacer tracks, the F1 track is quite long. Therefore, to have a

Open in app ↗

at the borders, leading to a loss of grip and control when driving too close to the edge. Thus, to balance reliability and speed on this track, we limit the racing line to the inner 80% of the track width.

The following graphs show the racing line with and without the limitation of only using the inner 80% of the track, in combination with different numbers of iterations.



Computed Racing Line for different number of iterations and usages of track widths (image by author)

Next, we want to calculate the optimal speed. Using a simplified approach, we can calculate the maximum speed for each point on the racing line with

$$v_{max} = \sqrt{\frac{Fr}{m}} = c\sqrt{r}$$

where F is the lateral gripping force, m is the mass of the car, and r is the radius of the curve. As we do not know F or m, we can simplify this equation by assigning a constant c to these unknown values.

The radius on each point of the racing line can be calculated by implying a circle through 3 points on our previously computed racing line: the current point and the points in front and behind that. Solving for the radius, we can calculate the radius with this python function:

```python
# Input 3 coords [[x1,y1],[x2,y2],[x3,y3]]
def circle_radius(coords):

    # Flatten the list and assign to variables
    x1, y1, x2, y2, x3, y3 = [i for sub in coords for i in sub]

    a = x1*(y2-y3) - y1*(x2-x3) + x2*y3 - x3*y2
    b = (x1**2+y1**2)*(y3-y2) + (x2**2+y2**2)*(y1-y3) + (x3**2+y3**2)*(y2-y1)
    c = (x1**2+y1**2)*(x2-x3) + (x2**2+y2**2)*(x3-x1) + (x3**2+y3**2)*(x1-x2)
    d = (x1**2+y1**2)*(x3*y2-x2*y3) + (x2**2+y2**2) * \
        (x1*y3-x3*y1) + (x3**2+y3**2)*(x2*y1-x1*y2)

    # In case a is zero (so radius is infinity)
    try:
        r = abs((b**2+c**2-4*a*d) / abs(4*a**2)) ** 0.5
    except:
        r = 999

    return r
```

circle_radius.py hosted with ❤ by GitHub view raw

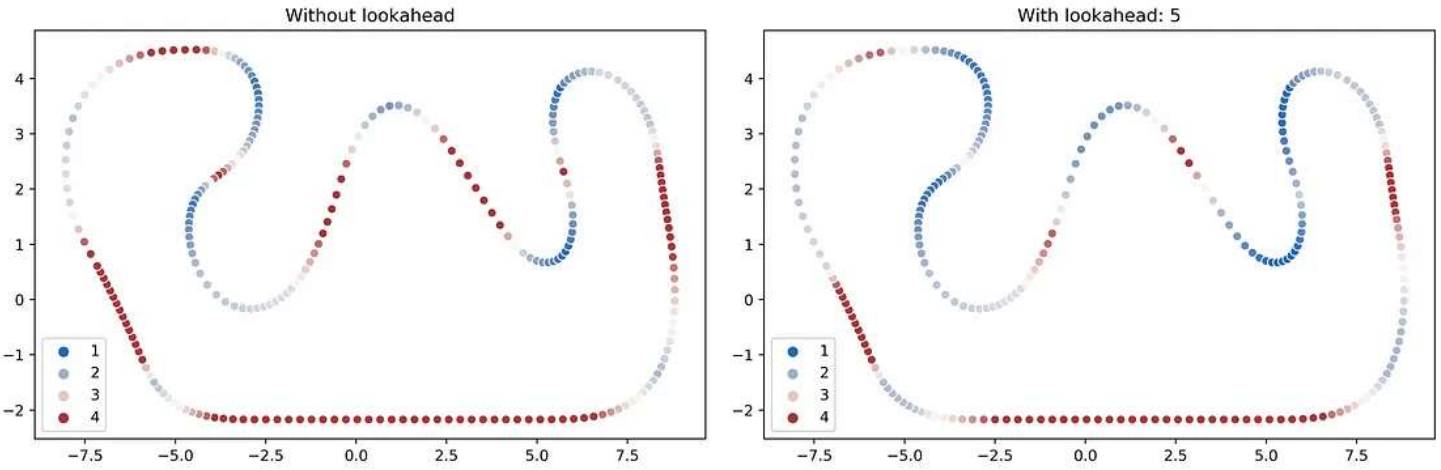
Although the points of the racing line are not equally spaced, the calculated radii are still accurate because of the high density of points. Therefore, any inaccuracies should not be significant enough to make a difference.

Once we have all the radii, we have to figure out *c* through experimentation. We do this by finding the highest possible speed with which the model can complete the tightest turn. For the F1 track, this maximum speed is about 1.3 m/s. We will later use this speed as the minimum speed of the action space.

$$c = \frac{v_{min}}{\sqrt{r_{min}}} = \frac{1.3m/s}{\sqrt{0.59m}} \approx 1.69\ m^{0.5}s^{-1}$$

The value of *c* was calculated on different tracks and was consistently in the range of 1.6 to 1.75.

To find the final optimal speed, we cap the speed at the maximum speed of the car, which was set to 4 m/s by our team for this track. Additionally, we introduce a lookahead factor, which we set to 5. This means that the optimal speed is the minimum of the maximum speeds of the next 5 points. The larger the lookahead value, the sooner the car starts breaking before a curve.



Computed optimal speeds on racing line, with different lookahead values (image by author)

In contrast to the optimal racing line, the optimal speed is only a broad approximation because many factors are not taken into account. To get an exact optimal speed, we would have to consider the exact mass, center of mass, moment of inertia, friction coefficient, cornering stiffness, and maximum acceleration and deceleration rates. We will have to keep this uncertainty in mind when designing the reward function.

3. Optimizing the Action Space

As DeepRacer's action space is discrete, some points in the action space will never be used, e.g. a speed of 4 m/s together with a steering angle of 30 degrees. Additionally, all tracks have an asymmetry in the direction of curves. For example, the F1 track is driven clockwise, leading to more right than left turns. Out of these 2 reasons, it is beneficial to optimize the action space. We can either opt for faster convergence by removing the actions that will not be used or for more precise driving if we keep the same number of actions but assign them more intelligently. We choose the latter. For the car setup, we use a single camera and a 3-layer Convolutional Neural Network because anything more complex does not improve performance for time trial and would only increase the time to converge.

We follow a 5-step approach:

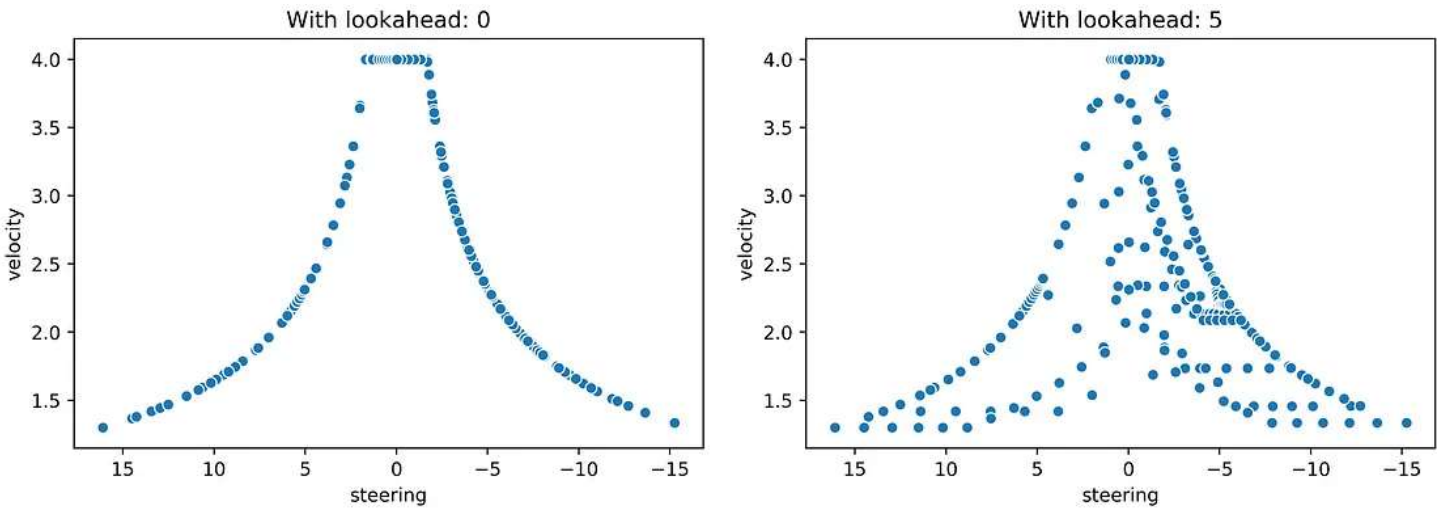
1. Calculate the Steering Angle
2. Add Gaussian Noise
3. Apply K-Means Clustering
4. Manually Add Actions
5. Export to S3

3.1 Calculate the Steering Angle

Until now, we only have the radius for each point on the racing line. This turning radius has to be converted to a steering angle with

$$r = \frac{L}{sin(\alpha)} \Leftrightarrow \alpha = arcsin\left(\frac{L}{r}\right)$$

where α is the steering angle, L=0.165m is the axle distance, and r is the radius of the curve.



All actions on the optimal racing line and optimal speed, for different lookahead values (image by author)

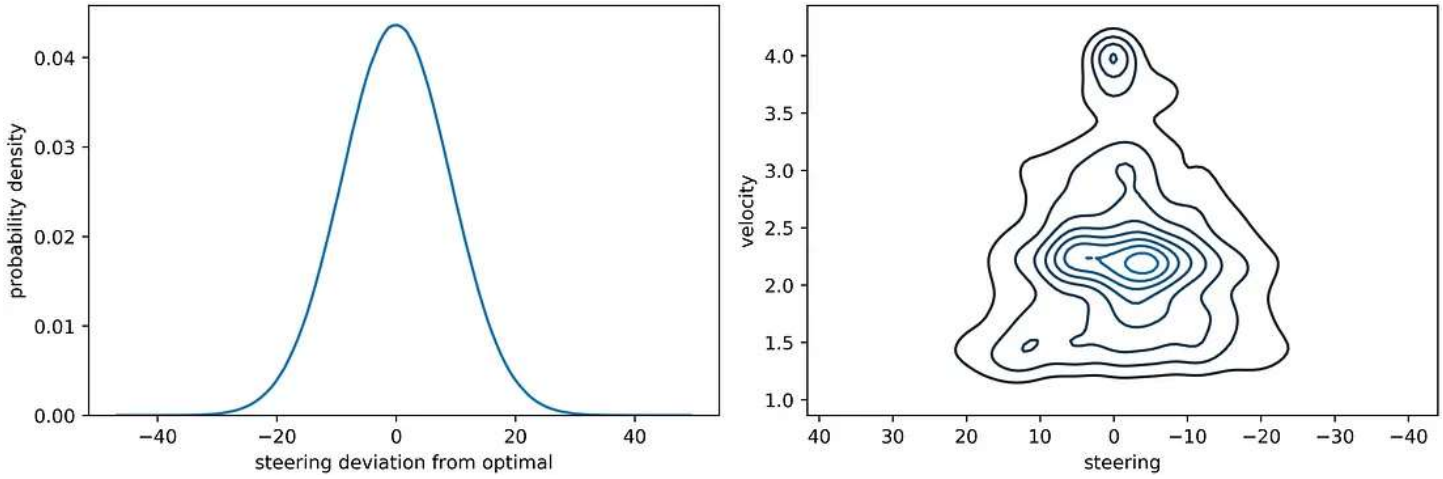
3.2 Add Gaussian Noise

In a perfect world, the model would always follow the optimal racing line and speed. However, this is never the case, especially not when training of the model has just begun. Therefore, to represent the uncertainty in driving and give the car more flexibility to correct previous decisions, we add Gaussian noise to each action. We only apply Gaussian noise to the steering, not the speed, because correcting previous decisions is primarily driven through the steering, not speed.

First, we have to determine the desired standard deviation of the Gaussian noise. Then, we generate an array of Gaussian noise, which will later be added to the existing data points.

```
from scipy import stats

# Find standard deviation so that probability of >15 degrees steering is 5%
steering_sd = -15 / stats.norm.ppf(0.05)

# Create array of noise based on normal distribution
resample_size = 100000
steering_noise = np.random.normal(0,steering_sd,resample_size)
```

gaussian_noise.py hosted with ❤ by GitHub view raw



Gaussian noise distribution for steering + KDE plot for actions infused with Gaussian noise (image by author)

3.3 Apply K-Means Clustering

The maximum amount of actions, when using the DeepRacer console is 21. We use K-Means clustering on the Gaussian noise infused actions to calculate 19 actions. The last 2 actions will be manually added in the next step. We use K-Means because this allows us to use the Euclidian distance for the 2 dimensions speed and steering — the closer the points are to one another, the more similar they are. The centroid of a cluster will represent one action. In case you are not familiar with K-Means, this article explains it very well.

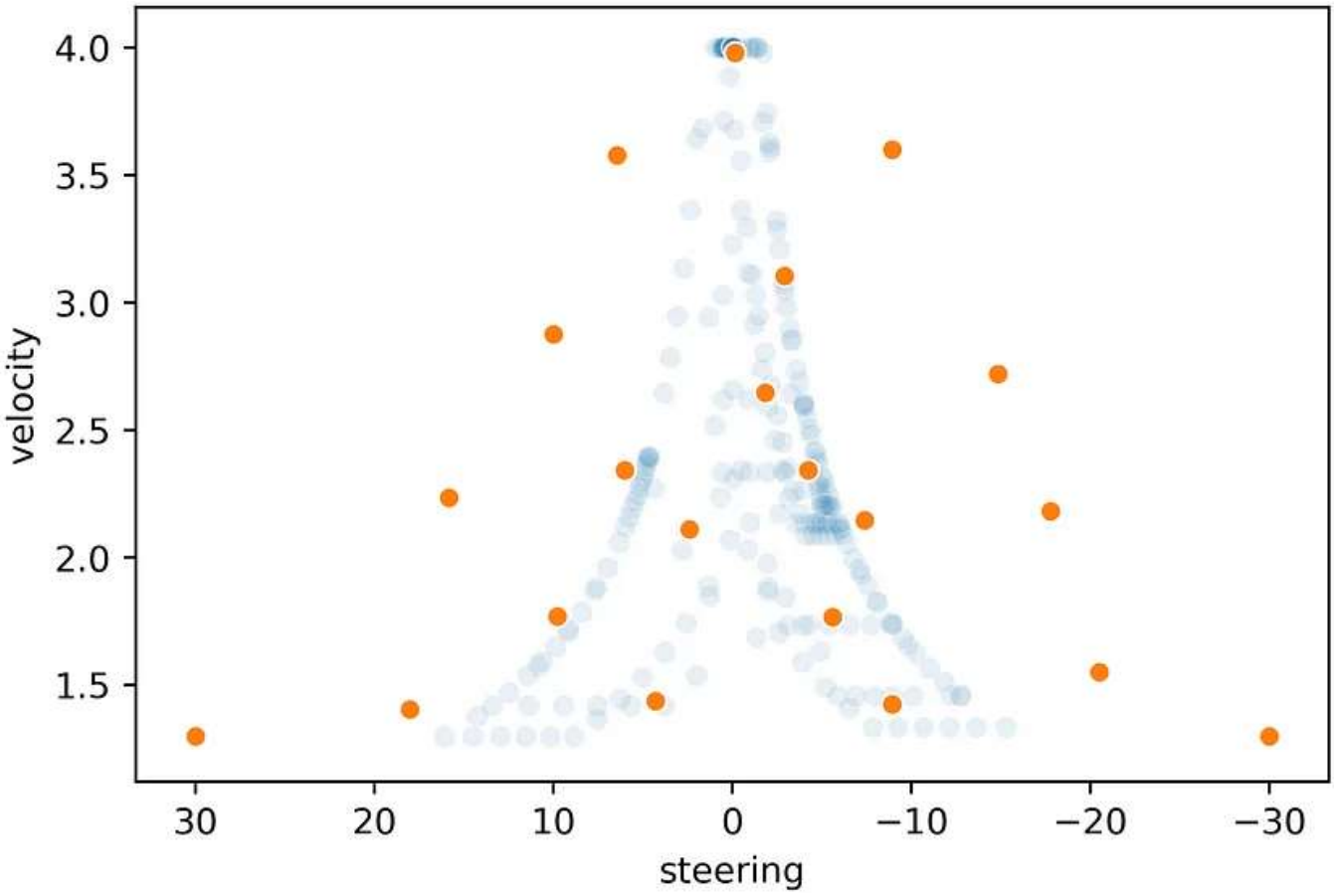
```python
from sklearn.preprocessing import MinMaxScaler
from sklearn.cluster import MiniBatchKMeans

# all_actions is a DataFrame with 2 columns: speed and steering
X = all_actions

# Rescale data with minmax
minmax_scaler = MinMaxScaler()
X_minmax = minmax_scaler.fit_transform(X)

# K-Means Clustering
n_clusters = 19
model = MiniBatchKMeans(n_clusters=n_clusters).fit(X_minmax)

# Interpretable Centroids will represent the action space
X_minmax_fit = minmax_scaler.fit(X)
X_centroids = X_minmax_fit.inverse_transform(model.cluster_centers_)
```

kmeans_actionspace.py hosted with ❤ by GitHub view raw

To see which additional pre-processing steps we applied to the initial data points, please refer to our GitHub Repo.

3.4 Manually Add Actions

Between each update of the model, multiple episodes are conducted. For example, if this value is set to 20, the car will start each episode 5% further down the track, compared to the previous episode. Therefore, the car will rarely start exactly on the racing line or its direction. To give the car the possibility to turn into the desired direction at the start of each episode, we want to add 2 additional actions: (min. speed, 30°) and (min. speed, -30°).



Final action space of size 21 (image by author)

To conclude, the method described in chapter 3 is a first approach and is better than the predefined actions. However, asymmetric actions may force the car to decide between turning and higher speed. For example, looking at the action space plot above, the car cannot drive at 4m/s and steer 3° at the same time. If it wants to steer at high speeds it has to reduce the speed. Thus, further experimentation with the action space might lead to better results.

3.5 Export to S3

After finding our desired action space, we have to export it to S3, in which the model metadata for DeepRacer is saved. We follow these simple steps:

1. Create a model in the console with 21 actions, and train this model for 5 minutes. Be aware that we cannot change the number of actions afterward, only the speed and steering for each action

2. Open the S3 folder aws-deepracer-xxx/model-metadata/modelname

3. Download model_metadata.json and replace the existing actions with the desired actions
4. In S3, replace the old model_metadata.json with the new file
5. Clone the previously created model in the AWS DeepRacer console. This clone will use our desired action space for training

4. The Reward Function

4.1 Challenges in Designing a Reward Function

Designing the reward function can be seen as the most challenging part of reinforcement learning. This is due to the large range of complexity, which reward functions can have. On the one end, a reward function with 5 lines of code can get us around the track eventually, albeit at low speeds and with lots of zig-zagging. On the other end, there are reward functions with hundreds of lines of code, for example when specifically telling the model where the racing line is.

The main objective when coding a good reward function for DeepRacer is this: for a given progress, the lower the time, the more reward the car should receive. For example, if 2 episodes both achieved 50% progress, but one was faster than the other, the faster episode should receive more reward. However, we also want to reward other aspects, such as the closeness to the optimal racing line. Therefore, balancing the different goals is the most challenging part of designing a reward function. Now let's explore how we can deal with these challenges and design an effective reward function.

4.2 Aspects of Our Reward Function

The reward function, we used to get to 12th place, has 5 main aspects:

1. Default reward
2. Closeness to the racing line
3. Speed difference to the optimal speed

4. Completing the lap in fewer steps
5. Punishment for obviously wrong decisions

First, we define a default reward, i.e. a minimum reward that is always given to the car, except for when it takes an obviously bad decision. As going off track results in zero reward, the higher the default reward is, the more crashing hurts the car, so the car will be more risk-averse. However, setting the default reward too high violates our objective that fewer steps equal more reward. Therefore, the default reward should not be too high.

Second, we add a reward for closeness to the racing line. Calculating this reward relies on the racing line we computed in chapter 2. Adding this reward reduces zig-zagging and is especially helpful when a new model has just started training. Yet, if this reward is too high, the car will only tediously follow the racing line, but not care about speed.

Third, we add a reward for closeness to the optimal speed. We are not using "more speed equals more reward" because if we have a wrong proportion of rewards, the car will predominantly care about going fast, and never be able to pass the first curve in training. So, it is easier to define an optimal speed, although we know that it only is an approximation.

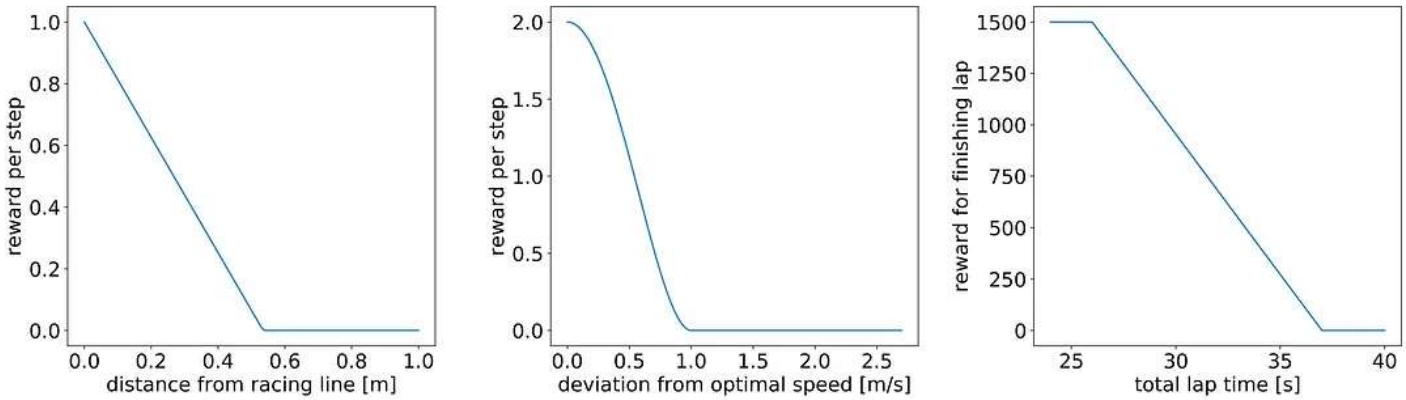
Fourth, we add a significant reward when the car completes a lap, so when it reaches 100% progress. The fewer steps it used, the higher this reward is. We can base the reward on the number of steps because the model takes 15±0.5 steps per second. We start giving a reward once the time is something the model can easily manage to do, while we cap the reward at a time that is equal to the fastest time on the leaderboard. We experimented with not only giving a reward when completing the lap but at more frequent intervals. However, that led to a model caring too much about speed.

Lastly, we set the total reward to almost zero for obviously wrong decisions, which are:

- being off track,
- having a direction heading, which deviates from the direction of the racing line by more than 30 degrees, or

- having a speed, which is 0.7 m/s slower than the optimal speed.

The cutoff values of 30 degrees and 0.7 m/s worked well for us, but further experimentation may lead to better results. Additionally, this punishment to almost zero makes our reward function discrete. In theory, continuous reward functions make the model learn faster because even if the car is doing something horrible, a slightly less horrible state should score lightly better. So as for the cutoff values, further experimentation may also lead to better results.



Aspects 2, 3, and 4 of our reward function (image by author)

To see our entire reward function, please refer to our GitHub Repo.

4.3 Adding vs. Multiplying Sub-Rewards

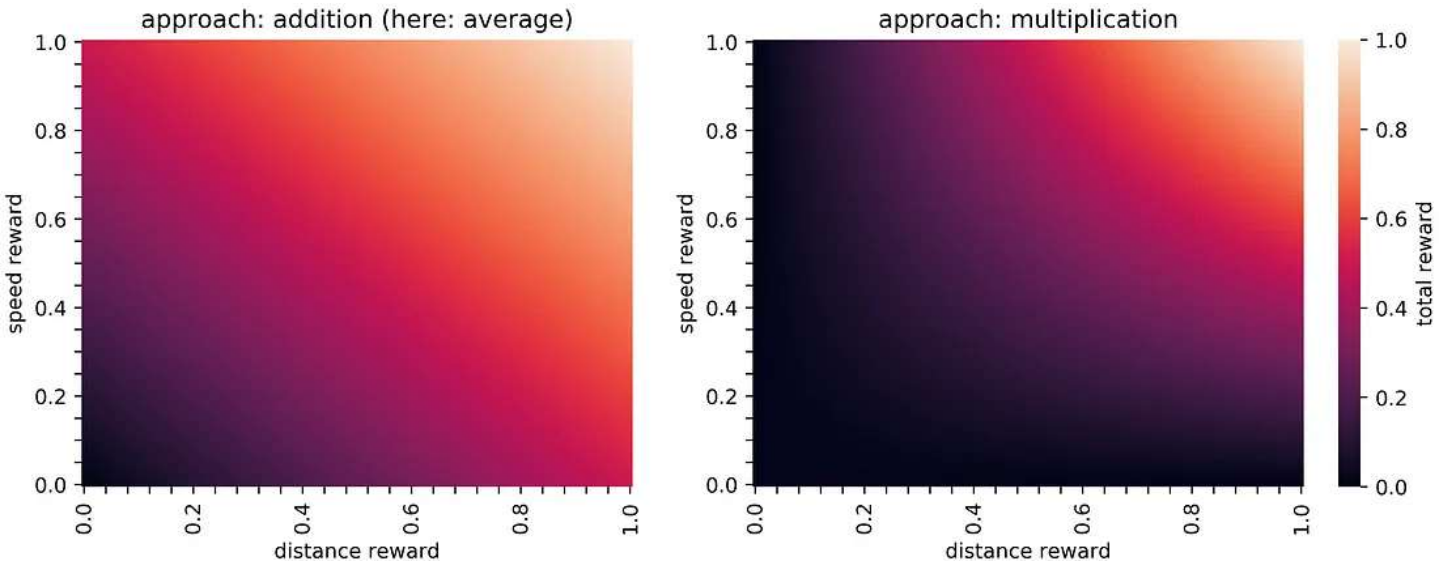
We define a sub-reward as an aspect of the reward function, such as the reward for closeness to the racing line. Within the reward function, there are 2 main ways to combine sub-rewards to the total reward: adding or multiplying them.

In our experience, adding sub-rewards together works better than multiplying sub-rewards. We believe that this is the case because if one sub-reward is close to zero, the model will not care about improving the other sub-rewards when using the multiplication approach.

For example, we tested a multiplication approach for 2 sub-rewards: closeness to the racing line, and closeness to the optimal speed. If the car is far away from the racing line but makes a good decision regarding speed, the car will still get zero reward, even though it took a good decision. Therefore, in our experiments, models

with a multiplication approach were never able to complete an entire lap, let alone a fast lap. From this, we learned that we have to reward the car for good speeds even if it is not on the optimal racing line.

The following plots visualize both approaches. In the multiplication approach, we can see that if the car is far off the racing line, improving the speed does not matter as much anymore.



Visualizing both approaches for combining sub-rewards (image by author)

5. Hyperparameters

Hyperparameters have quite a steep learning curve, so mastering them takes a lot of time. Therefore, we recommend to get your head around action spaces and reward functions first, before experimenting with hyperparameters.

During our experiments, we learned that starting with default hyperparameters, but higher batch size because of the long track, works well for the first hours of training. As soon as the model starts to converge, we reduce the entropy and learning rate. However, hyperparameters highly depend on the reward function, action space, and track, so we encourage you to experiment a lot.

Hyperparameter	Advantage of higher values	Disadvantage of higher values	Default
Batch Size	More stable updates	Slower training	64
Epochs	Policy updates have higher effect	Slower training	10
Learning Rate	Faster training	May struggle to converge	0.0003
Entropy	More experimentation may lead to better results	May struggle to converge	0.01
Discount Factor	More stable model	Slower model	0.999
Episodes per Iteration	Improves model stability	Slower training	20

A note on entropy: sometimes, a model performs worse after cloning. The reason for this is that entropy decreases over time during training as the model becomes more confident in its decisions. However, when the model is cloned, the entropy is reset to the hyperparameter value, which is defined when setting up the training.

6. Continuous Improvement with Log Analysis

The aspects we covered until now should only be a starting point for you. Each track, action space, and model behaves differently. This is why analyzing the logs after each training is so important.

Fortunately, the DeepRacer Community wrote a Log Analysis Tool on GitHub, with which training sessions, evaluations, and the action space can be analyzed. We highly recommend to use them. There are multiple resources, such as this blog post, which explain everything you need to know about the log analysis tool.

The overall goal of log analysis is to try out different variations of reward functions, hyperparameters, and action spaces and see which variations lead to the best time, progress, improvement, or convergence. This iterative approach takes time. So, you should plan your experiments considering your time and budget constraints. For example, our team of 3 conducted 477 different training sessions, accumulating a total of 2950 hours of training in April and May.

To decide which variation we want to pursue further, we look at the time and progress. Models that perform well on these two aspects, compared to similar experiments, are further pursued. When creating a new model, we follow a 3 step process:

1. Make sure that the model makes progress on the track. Early on, rule out models that struggle with completing one single entire lap
2. Once at least some laps have been completed, focus on lower lap times. The progress does not have to be high in this step, as long as at least some laps are being completed
3. Once lap times have converged, optimize for higher progress. Although optimizing for progress often increases lap times because the model becomes more risk-averse, this part is important to get a reliable model that can complete 3 laps in a row

A health warning about the cost of training: We as students were only able to train for so many hours because May 2020 was free of charge for the F1 event. So please keep an eye on the billing dashboard as it is very easy to run up a large bill.

7. Automated Race Submissions with Selenium

After having trained a model, with which we are happy, we submit it to the race. A good model will have a balance between speed and reliability. Therefore, it will not complete 100% of laps or have consistent lap times. As only the best time out of all submissions counts for the final time, we can submit our model multiple times to improve our ranking in the race. This can either be done manually or automatically through a web scraping tool.

Multiple web scraping packages for python are available. This article describes the 3 most popular ones: Scrapy, Selenium, and Beautiful Soup.

Using Selenium, we coded a function, which auto-submits a model to a race for a specified amount of time. The advantage of this is, that we are only using the console, not SageMaker nor the AWS CLI. As a bonus, we also coded a function, which automatically conducts experiments with hyperparameters. This can be used to conduct multiple experiments overnight without having to manually set them up every couple of hours.

To see the code for our Selenium functions, please refer to our GitHub Repo.

8. Summary & Next Steps

Only using the AWS DeepRacer console to train models, we showed how to compute the optimal racing line and speed, optimize the action space with K-Means clustering, design a good reward function, analyze logs to continuously improve the model, and auto-submit the model to the race. Using all these tools and spending a bit of time to adapt them to your situation, you will be able to improve your DeepRacer rankings soon. In general, we were able to show that balancing the different goals is the main challenge in applied reinforcement learning. In the case of DeepRacer, these goals are speed, reliability, and fast learning.

As a next step, once you have a well enough understanding of DeepRacer, you could try training models in AWS SageMaker or even a local setup. These 2 options will give you higher flexibility and possibly lower costs than using the DeepRacer Console. All the necessary resources are located in the DeepRacer Community's GitHub. Additionally, feel free to check out the community's website or YouTube channel. The community is always willing to help if you get stuck, need some advice, or simply want to learn more about DeepRacer.

F1 ProAm Event - May Qualifier

1292 racers

Search by racer alias

1 2 3 4 5 6 7 ... 65

Rank	Racer	Time	Gap to 1st	Video	Off-track
1	JJ	01:17.517		Watch	-
2	GT-DevelopersIO	01:21.264	+00:03.747	Watch	-
3	Mentaiko-DevelopersIO	01:22.601	+00:05.084	Watch	1
4	RayG	01:23.696	+00:06.179	Watch	-
5	OneDudeDesign	01:23.724	+00:06.207	Watch	-
6	Giacomo	01:24.693	+00:07.176	Watch	-
7	JMajor	01:26.296	+00:08.779	Watch	-
8	Robin-Castro	01:26.451	+00:08.934	Watch	-
9	AkramDweikat	01:29.951	+00:12.434	Watch	-
10	Karl-NAB	01:30.827	+00:13.310	Watch	-
11	ShinyThings-DBS	01:31.236	+00:13.719	Watch	-
12	ESADE-MIBA-3	01:31.775	+00:14.258	Watch	-
13	ESADE-MIBA-2	01:32.867	+00:15.350	Watch	-
14	Fumiaki	01:33.521	+00:16.004	Watch	-

Final Placement in the time trial category of the F1 event in May 2020 (image by author)

A big thank you to my team members Natalia Korchagina and Marc Cervera, without whom our team would have never achieved its result. Also, thank you to Lyndon Leggate and Tomasz Ptak from the DeepRacer community for their amazing helpfulness. Finally, thank you to our ESADE professors, who allowed us to participate in DeepRacer as a university project — we learned so much about reinforcement learning along the way!

Reinforcement Learning | Aws Deepracer | Machine Learning | Formula 1 | Self Driving Cars

Follow

Written by Daniel Gonzalez

50 Followers · Writer for Towards Data Science

AI enthusiast - linkedin.com/in/dg4

More from Daniel Gonzalez and Towards Data Science

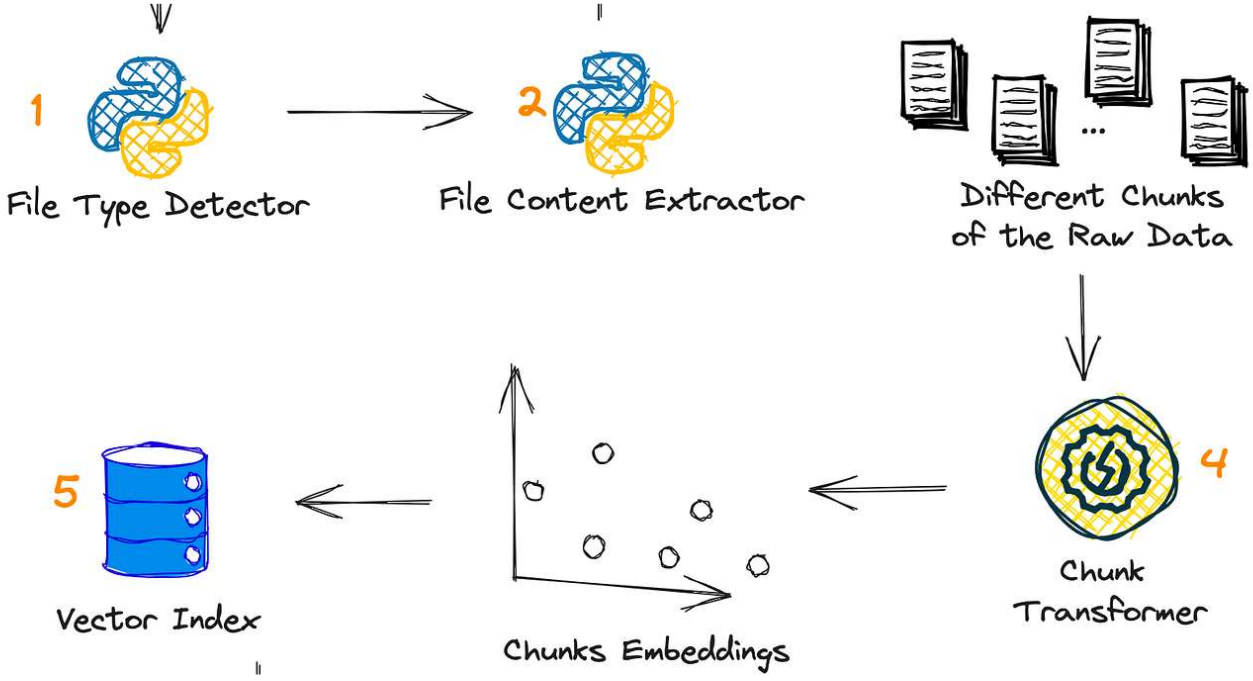
Bex T. in Towards Data Science

130 ML Tricks And Resources Curated Carefully From 3 Years (Plus Free eBook)

Each one is worth your time

✨ · 48 min read · Aug 1

3.4K 11



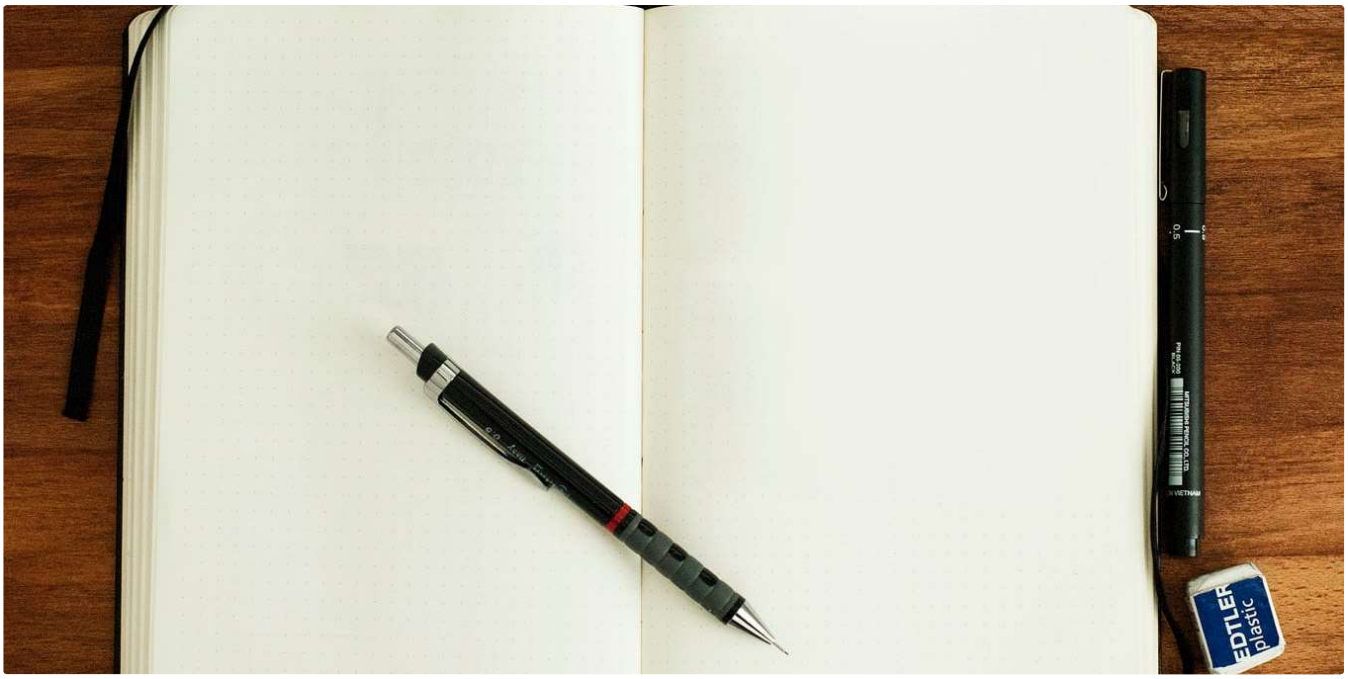
Zoumana Keita in Towards Data Science

How to Chat With Any File from PDFs to Images Using Large Language Models—With Code

Complete guide to building an AI assistant that can answer questions about any file

✦ · 9 min read · Aug 5

1.2K 12



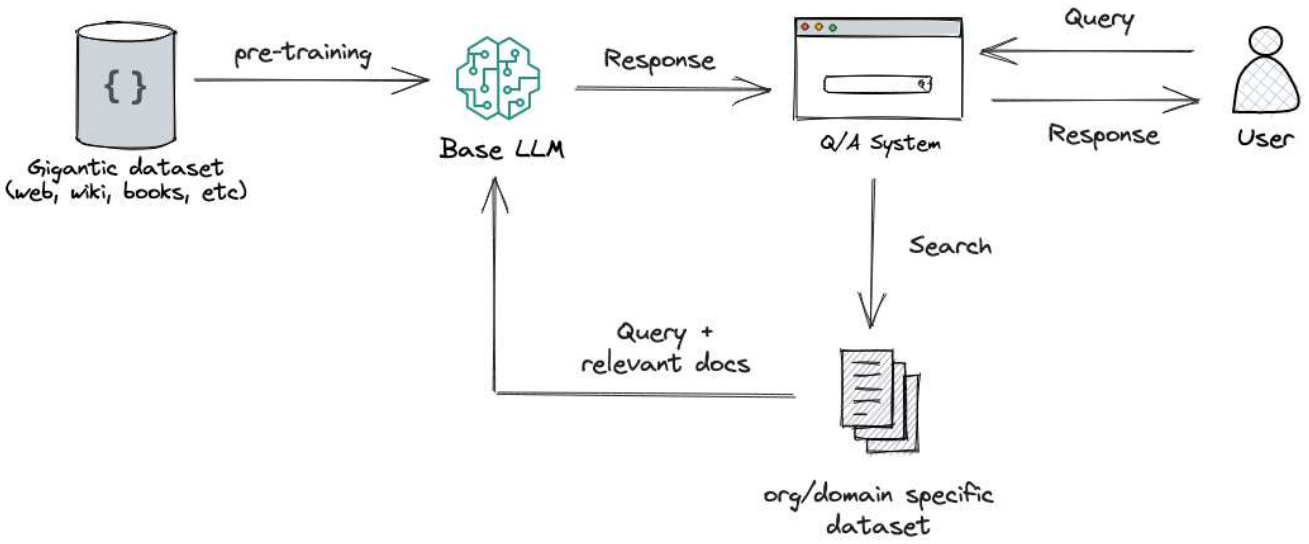
Cameron R. Wolfe, Ph.D. in Towards Data Science

Advanced Prompt Engineering

What to do when few-shot learning isn't enough...

✦ · 17 min read · Aug 7

849 8



Heiko Hotz in Towards Data Science

RAG vs Finetuning—Which Is the Best Tool to Boost Your LLM Application?

The definitive guide for choosing the right method for your use case

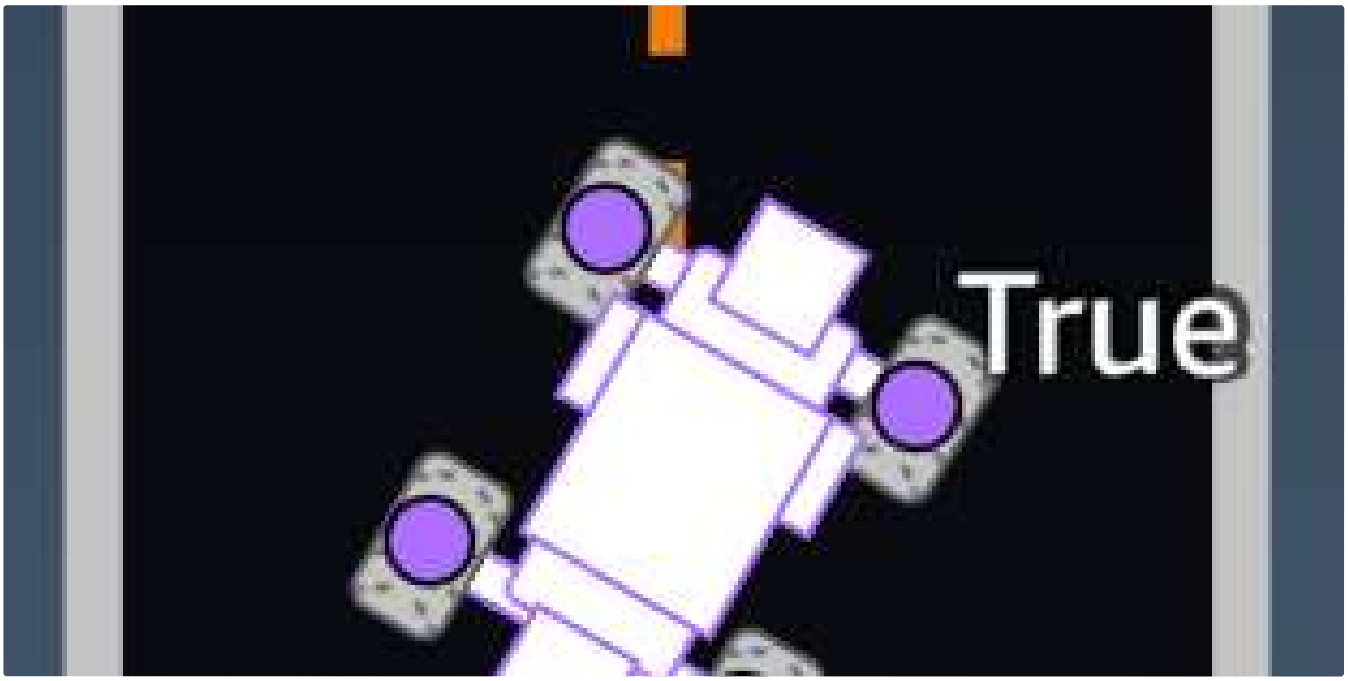
✦ · 19 min read · Aug 25

1K 13

See all from Daniel Gonzalez

See all from Towards Data Science

Recommended from Medium



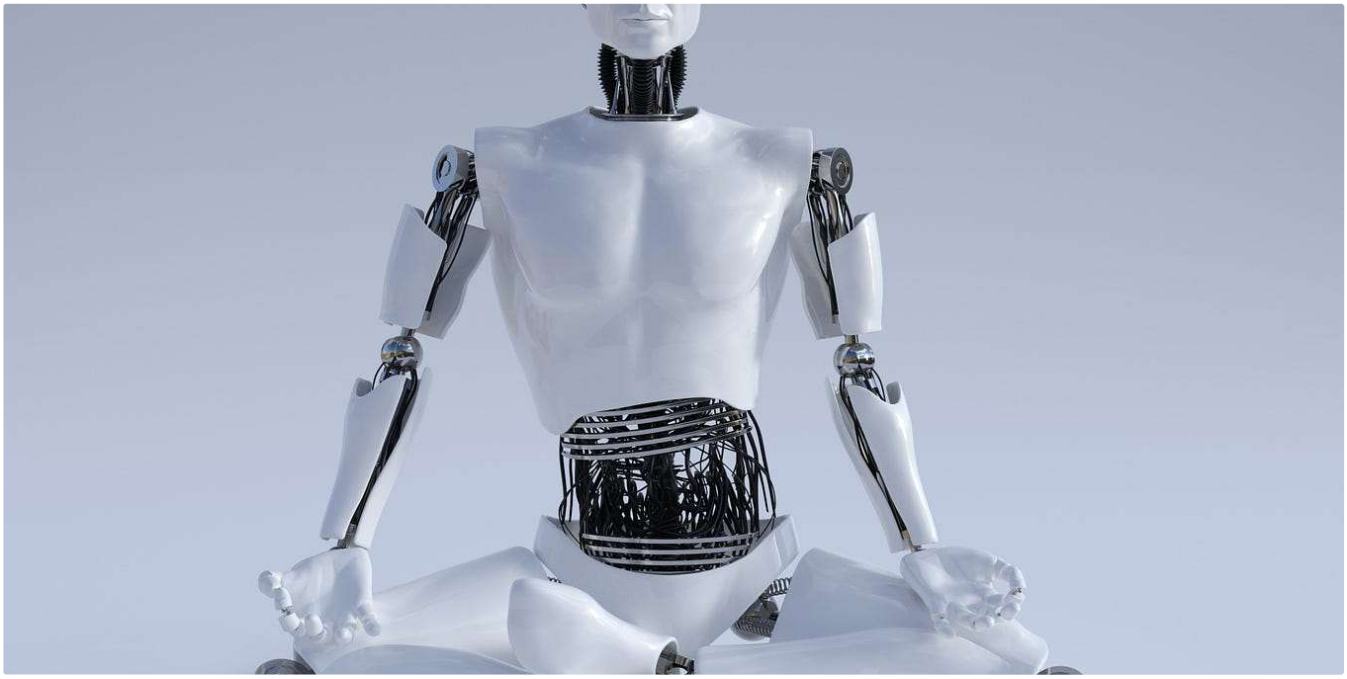
Aleksander Berezowski

Parameters In Depth: DeepRacer Student League Guide

Images and foundational content are based on this official AWS DeepRacer Guide, but adapted and expanded for DeepRacer's Student League...

8 min read · Mar 28

21



The PyCoach in Artificial Corner

You're Using ChatGPT Wrong! Here's How to Be Ahead of 99% of ChatGPT Users

Master ChatGPT by learning prompt engineering.

· 7 min read · Mar 18

31K 568

Lists

Predictive Modeling w/ Python

20 stories · 336 saves

Practical Guides to Machine Learning

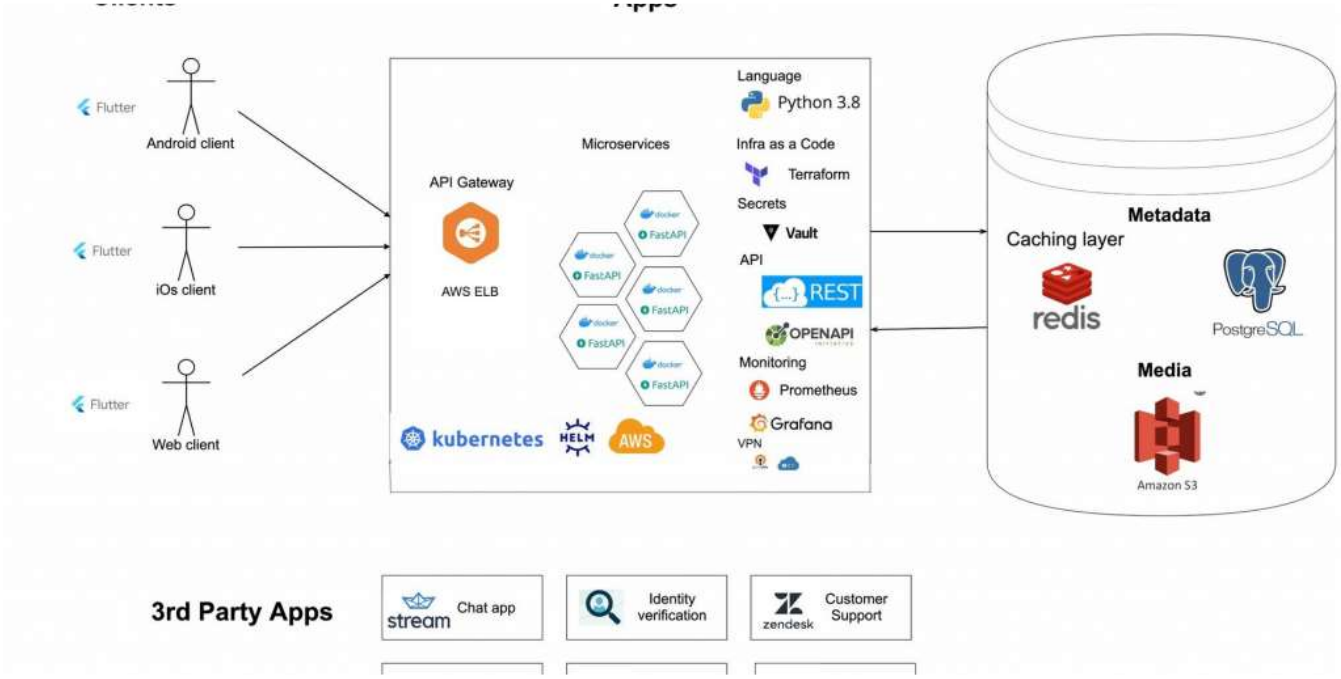
10 stories · 378 saves

Natural Language Processing

568 stories · 188 saves

The New Chatbots: ChatGPT, Bard, and Beyond

13 stories · 104 saves



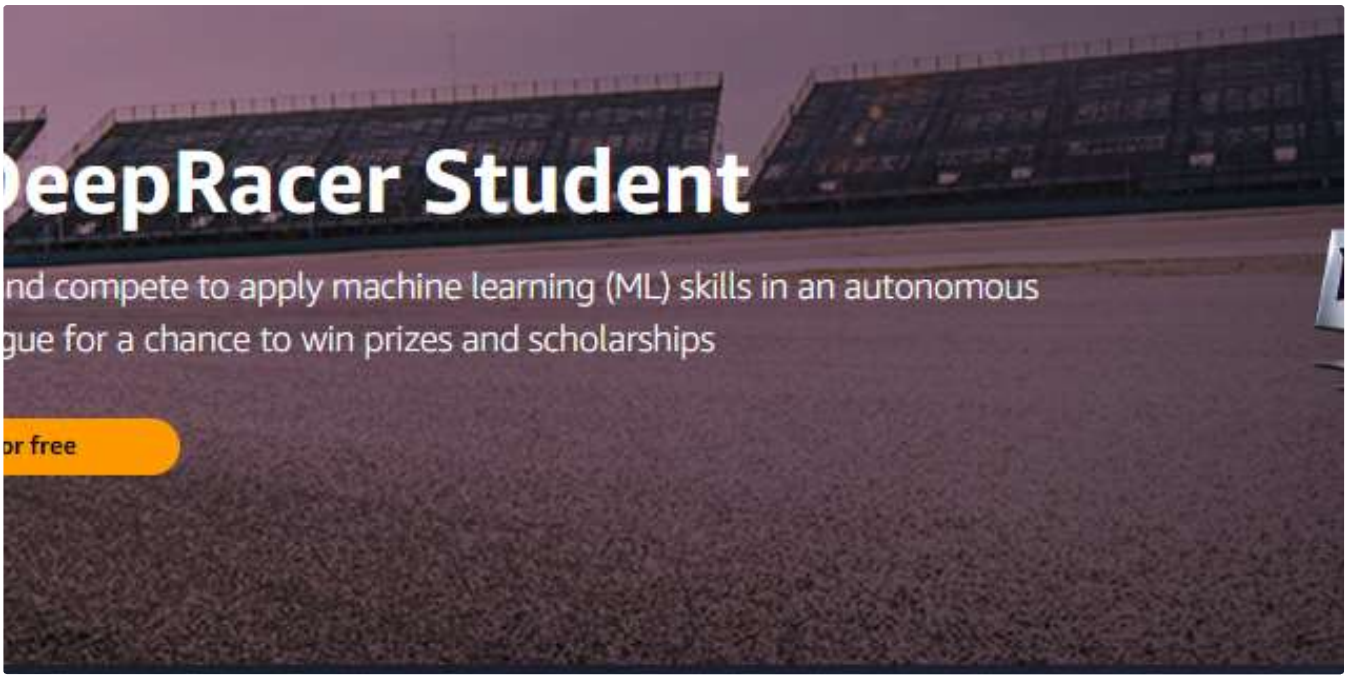
Dmitry Kruglov in Better Programming

The Architecture of a Modern Startup

Hype wave, pragmatic evidence vs the need to move fast

16 min read · Nov 8, 2022

5.3K 48



Ansh Tanwar

Dominate the Student League of AWS DeepRacer

PART-I

4 min read · Mar 24

159 3

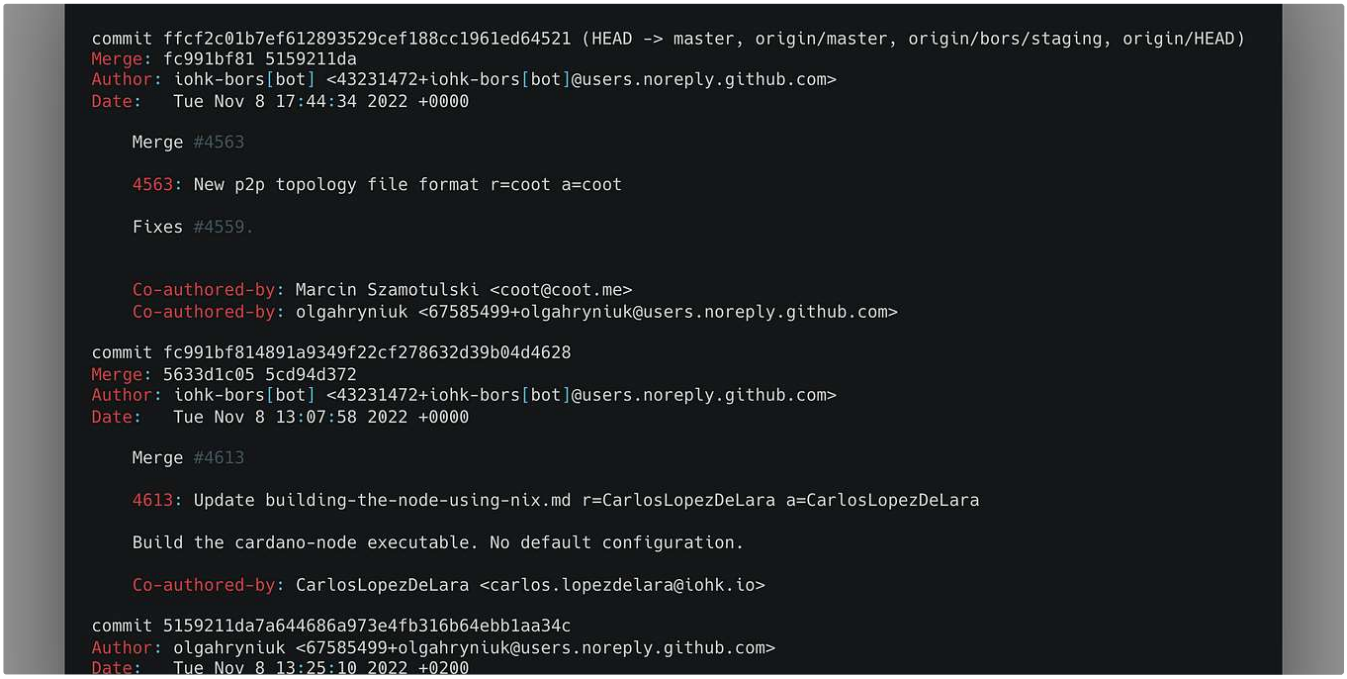
Matt Chapman in Towards Data Science

The Portfolio that Got Me a Data Scientist Job

Spoiler alert: It was surprisingly easy (and free) to make

✨ · 10 min read · Mar 25

4.3K 74



```
commit ffcf2c01b7ef612893529cef188cc1961ed64521 (HEAD -> master, origin/master, origin/bors/staging, origin/HEAD)
Merge: fc991bf81 5159211da
Author: iohk-bors[bot] <43231472+iohk-bors[bot]@users.noreply.github.com>
Date:   Tue Nov 8 17:44:34 2022 +0000

    Merge #4563

    4563: New p2p topology file format r=coot a=coot

    Fixes #4559.


    Co-authored-by: Marcin Szamotulski <coot@coot.me>
    Co-authored-by: olgahryniuk <67585499+olgahryniuk@users.noreply.github.com>

commit fc991bf814891a9349f22cf278632d39b04d4628
Merge: 5633d1c05 5cd94d372
Author: iohk-bors[bot] <43231472+iohk-bors[bot]@users.noreply.github.com>
Date:   Tue Nov 8 13:07:58 2022 +0000

    Merge #4613

    4613: Update building-the-node-using-nix.md r=CarlosLopezDeLara a=CarlosLopezDeLara

    Build the cardano-node executable. No default configuration.

    Co-authored-by: CarlosLopezDeLara <carlos.lopezdelara@iohk.io>

commit 5159211da7a644686a973e4fb316b64ebb1aa34c
Author: olgahryniuk <67585499+olgahryniuk@users.noreply.github.com>
Date:   Tue Nov 8 13:25:10 2022 +0200
```

Jacob Bennett in Level Up Coding

Use Git like a senior engineer

Git is a powerful tool that feels great to use when you know how to use it.

4 min read · Nov 15, 2022

9.1K 100

See more recommendations