Training_analysis

April 12, 2021

1 Training analysis for DeepRacer

This notebook has been built based on the `DeepRacer Log Analysis.ipynb` provided by the AWS DeepRacer Team. It has been reorganised and expanded to provide new views on the training data without the helper code which was moved into utility `.py` files.

1.1 Usage

I have expanded this notebook from to present how I'm using this information. It contains descriptions that you may find not that needed after initial reading. Since this file can change in the future, I recommend that you make its copy and reorganize it to your liking. This way you will not lose your changes and you'll be able to add things as you please.

This notebook isn't complete. What I find interesting in the logs may not be what you will find interesting and useful. I recommend you get familiar with the tools and try hacking around to get the insights that suit your needs.

1.2 Contributions

As usual, your ideas are very welcome and encouraged so if you have any suggestions either bring them to the AWS DeepRacer Community or share as code contributions.

1.3 Training environments

Depending on whether you're running your training through the console or using the local setup, and on which setup for local training you're using, your experience will vary. As much as I would like everything to be taylored to your configuration, there may be some problems that you may face. If so, please get in touch through the AWS DeepRacer Community.

1.4 Requirements

Before you start using the notebook, you will need to install some dependencies. If you haven't yet done so, have a look at The README.md file to find what you need to install.

Apart from the install, you also have to configure your programmatic access to AWS. Have a look at the guides below, AWS resources will lead you by the hand:

AWS CLI: https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-configure.html

Boto Configuration: https://boto3.amazonaws.com/v1/documentation/api/latest/guide/configuration.html

1.5 Credits

I would like to thank the AWS DeepRacer Community for all the feedback about the notebooks. If you'd like, follow my blog where I tend to write about my experiences with AWS DeepRacer.

2 Log Analysis

Let's get to it.

2.1 Permissions

Depending on where you are downloading the data from, you will need some permissions: * Access to CloudWatch log streams * Access to S3 bucket to reach the log files

2.2 Installs and setups

If you are using an AWS SageMaker Notebook to run the log analysis, you will need to ensure you install required dependencies. To do that uncomment and run the following:

[1]:
```
# Make sure you have deepracer-utils >= 0.9

# import sys

# !{sys.executable} -m pip install --upgrade deepracer-utils
```

2.3 Imports

Run the imports block below:

[4]:
```
import pandas as pd
import matplotlib.pyplot as plt
from pprint import pprint

from deepracer.tracks import TrackIO, Track
from deepracer.tracks.track_utils import track_breakdown, track_meta
from deepracer.logs import \
    SimulationLogsIO as slio, \
    NewRewardUtils as nr, \
    AnalysisUtils as au, \
    PlottingUtils as pu, \
    ActionBreakdownUtils as abu, \
    DeepRacerLog

# Ignore deprecation warnings we have no power over
import warnings
warnings.filterwarnings('ignore')
```

2.4 Get the logs

Depending on which way you are training your model, you will need a slightly different way to load the data.

AWS DeepRacer Console

The logs can be downloaded from the training page. Once you download them, extract the archive into logs/[training-name] (just like logs/sample-logs)

DeepRacer for Cloud

If you're using local training, just point at your model's root folder in the minio bucket. If you're using any of the cloudy deployments, download the model folder to local and point at it.

Deepracer for dummies/Chris Rhodes' Deepracer/ARCC Deepracer or any training solution other than the ones above, read below

This notebook has been updated to support the most recent setups. Most of the mentioned projects above are no longer compatible with AWS DeepRacer Console anyway so do consider moving to the ones actively maintained.

[6]:
```
model_logs_root = 'logs/sample-logs'
log = DeepRacerLog(model_logs_root)

# load logs into a dataframe
log.load()

try:
    pprint(log.agent_and_network())
    print("-------------")
    pprint(log.hyperparameters())
    print("-------------")
    pprint(log.action_space())
except Exception:
    print("Robomaker logs not available")

df = log.dataframe()
```

```
Robomaker logs not available
```

If the code above worked, you will see a list of details printed above: a bit about the agent and the network, a bit about the hyperparameters and some information about the action space. Now let's see what got loaded into the dataframe - the data structure holding your simulation information. the `head()` method prints out a few first lines of the data:

[7]:
```
df.head()
```

[7]:

	episode	steps	x	y	heading	steering_angle	speed	\
0	0	1.0	0.322285	2.691374	-84.050685	-15.0	0.6	
1	0	2.0	0.322204	2.677266	-84.443996	30.0	0.6	
2	0	3.0	0.322527	2.663781	-84.696118	0.0	0.6	

```
3         0    4.0  0.322357  2.641572 -85.288707            30.0    0.6
4         0    5.0  0.325243  2.608099 -85.292034            15.0    0.6

   action  reward   done  all_wheels_on_track  progress  closest_waypoint  \
0       1     1.0  False                 True  0.605483                 1
1       4     0.8  False                 True  0.665849                 1
2       2     1.0  False                 True  0.723791                 1
3       4     0.8  False                 True  0.818796                 1
4       3     1.0  False                 True  0.963899                 1

   track_len        tstamp episode_status  iteration  worker  unique_episode
0  23.118222  1.613889e+09    in_progress          0       0               0
1  23.118222  1.613889e+09    in_progress          0       0               0
2  23.118222  1.613889e+09    in_progress          0       0               0
3  23.118222  1.613889e+09    in_progress          0       0               0
4  23.118222  1.613889e+09    in_progress          0       0               0
```

2.5 Load waypoints for the track you want to run analysis on

The track waypoint files represent the coordinates of characteristic points of the track - the center line, inside border and outside border. Their main purpose is to visualise the track in images below.

The naming of the tracks is not super consistent. The ones that we already know have been mapped to their official names in the track_meta dictionary.

Some npy files have an 'Eval' suffix. One of the challenges in the past was that the evaluation tracks were different to physical tracks and we have recreated them to enable evaluation. Remeber that evaluation npy files are a community effort to visualise the tracks in the trainings, they aren't 100% accurate.

Tracks Available:

[8]:
```
tu = TrackIO()

for track in tu.get_tracks():
    print("{} - {}".format(track, track_meta.get(track[:-4], "I don't know")))
```

```
AWS_track.npy - I don't know
Albert.npy - Yun Speedway
AmericasGeneratedInclStart.npy - Badaal Track
Aragon.npy - Stratus Loop
Austin.npy - American Hills Speedway
Belille.npy - Cumulo Turnpike
Bowtie_track.npy - Bowtie Track
Canada_Eval.npy - Toronto Turnpike Eval
Canada_Training.npy - Toronto Turnpike Training
China_eval_track.npy - Shanghai Sudu Eval
China_track.npy - Shanghai Sudu Training
FS_June2020.npy - Fumiaki Loop
```

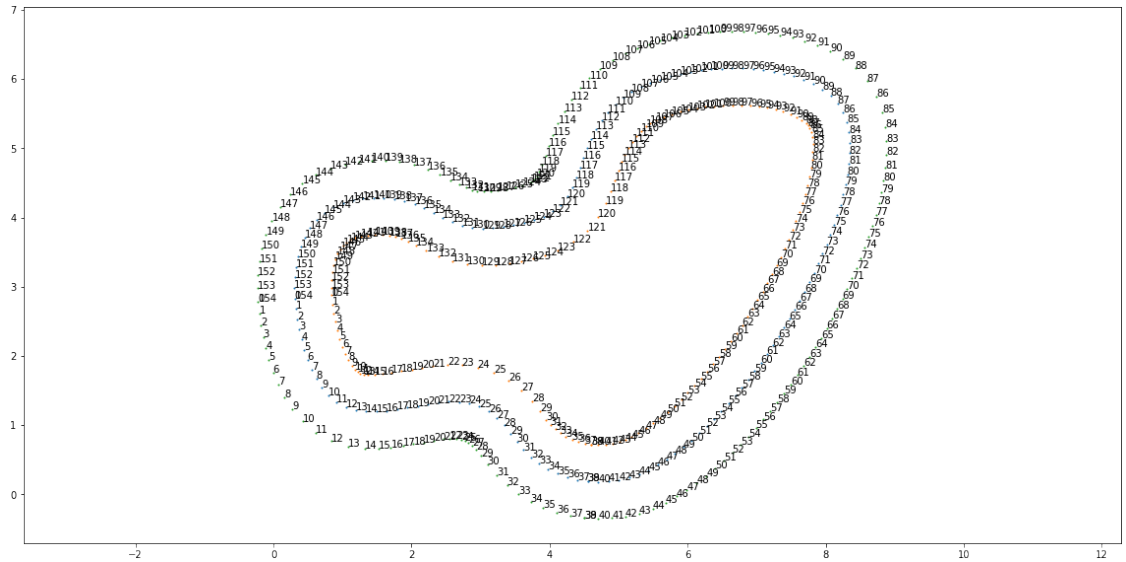
```
H_track.npy - H track
July_2020.npy - Roger Raceway
LGSWide.npy - SOLA Speedway
London_Loop_Train.npy - I don't know
Mexico_track.npy - Cumulo Carrera Training
Mexico_track_eval.npy - Cumulo Carrera Eval
Monaco.npy - European Seaside Circuit
New_York_Eval_Track.npy - Empire City Eval
New_York_Track.npy - Empire City Training
Oval_track.npy - Oval Track
Singapore.npy - Asia Pacific Bay Loop
Spain_track.npy - Circuit de Barcelona-Catalunya
Straight_track.npy - Straight track
Tokyo_Training_track.npy - Kumo Torakku Training
Vegas_track.npy - AWS Summit Raceway
Virtual_May19_Train_track.npy - London Loop Training
reInvent2019_track.npy - The 2019 DeepRacer Championship Cup
reInvent2019_wide.npy - re:Invent 2018 Wide
reInvent2019_wide_mirrored.npy - re:Invent 2018 Wide Mirrored
reinvent_base.npy - re:Invent 2018
```

Now let's load the track:

[10]:
```
# We will try to guess the track name first, if it
# fails, we'll use the constant in quotes

try:
    track_name = log.agent_and_network()["world"]
except Exception as e:
    track_name = "reInvent2019_track"


track: Track = tu.load_track(track_name)

pu.plot_trackpoints(track)
```

```
Loaded 155 waypoints
```

[10]: <AxesSubplot:>



2.6 Graphs

The original notebook has provided some great ideas on what could be visualised in the graphs. Below examples are a slightly extended version. Let's have a look at what they are presenting and what this may mean to your training.

2.6.1 Training progress

As you have possibly noticed by now, training episodes are grouped into iterations and this notebook also reflects it. What also marks it are checkpoints in the training. After each iteration a set of ckpt files is generated - they contain outcomes of the training, then a model.pb file is built based on that and the car begins a new iteration. Looking at the data grouped by iterations may lead you to a conclusion, that some earlier checkpoint would be a better start for a new training. While this is limited in the AWS DeepRacer Console, with enough disk space you can keep all the checkpoints along the way and use one of them as a start for new training (or even as a submission to a race).

While the episodes in a given iteration are a mixture of decision process and random guesses, mean results per iteration may show a specific trend. Mean values are accompanied by standard deviation to show the concentration of values around the mean.

Rewards per Iteration You can see these values as lines or dots per episode in the AWS DeepRacer console. When the reward goes up, this suggests that a car is learning and improving with regards to a given reward function. **This does not have to be a good thing.** If your reward function rewards something that harms performance, your car will learn to drive in a way that will make results worse.

At first the rewards just grow if the progress achieved grows. Interesting things may happen slightly later in the training:

- The reward may go flat at some level - it might mean that the car can't get any better. If

you think you could still squeeze something better out of it, review the car's progress and consider updating the reward function, the action space, maybe hyperparameters, or perhaps starting over (either from scratch or from some previous checkpoint)
- The reward may become wobbly - here you will see it as a mesh of dots zig-zagging. It can be a gradually growing zig-zag or a roughly stagnated one. This usually means the learning rate hyperparameter is too high and the car started doing actions that oscilate around some local extreme. You can lower the learning rate and hope to step closer to the extreme. Or run away from it if you don't like it
- The reward plunges to near zero and stays roughly flat - I only had that when I messed up the hyperparameters or the reward function. Review recent changes and start training over or consider starting from scratch

The Standard deviation says how close from each other the reward values per episode in a given iteration are. If your model becomes reasonably stable and worst performances become better, at some point the standard deviation may flat out or even decrease. That said, higher speeds usually mean there will be areas on track with higher risk of failure. This may bring the value of standard deviation to a higher value and regardless of whether you like it or not, you need to accept it as a part of fighting for significantly better times.

Time per iteration I'm not sure how useful this graph is. I would worry if it looked very similar to the reward graph - this could suggest that slower laps will be getting higher rewards. But there is a better graph for spotting that below.

Progress per Iteration This graph usually starts low and grows and at some point it will get flatter. The maximum value for progress is 100% so it cannot grow without limits. It usually shows similar initial behaviours to reward and time graphs. I usually look at it when I alter an action in training. In such cases this graph usually dips a bit and then returns or goes higher.

Total reward per episode This graph has been taken from the orignal notebook and can show progress on certain groups of behaviours. It usually forms something like a triangle, sometimes you can see a clear line of progress that shows some new way has been first taught and then perfected.

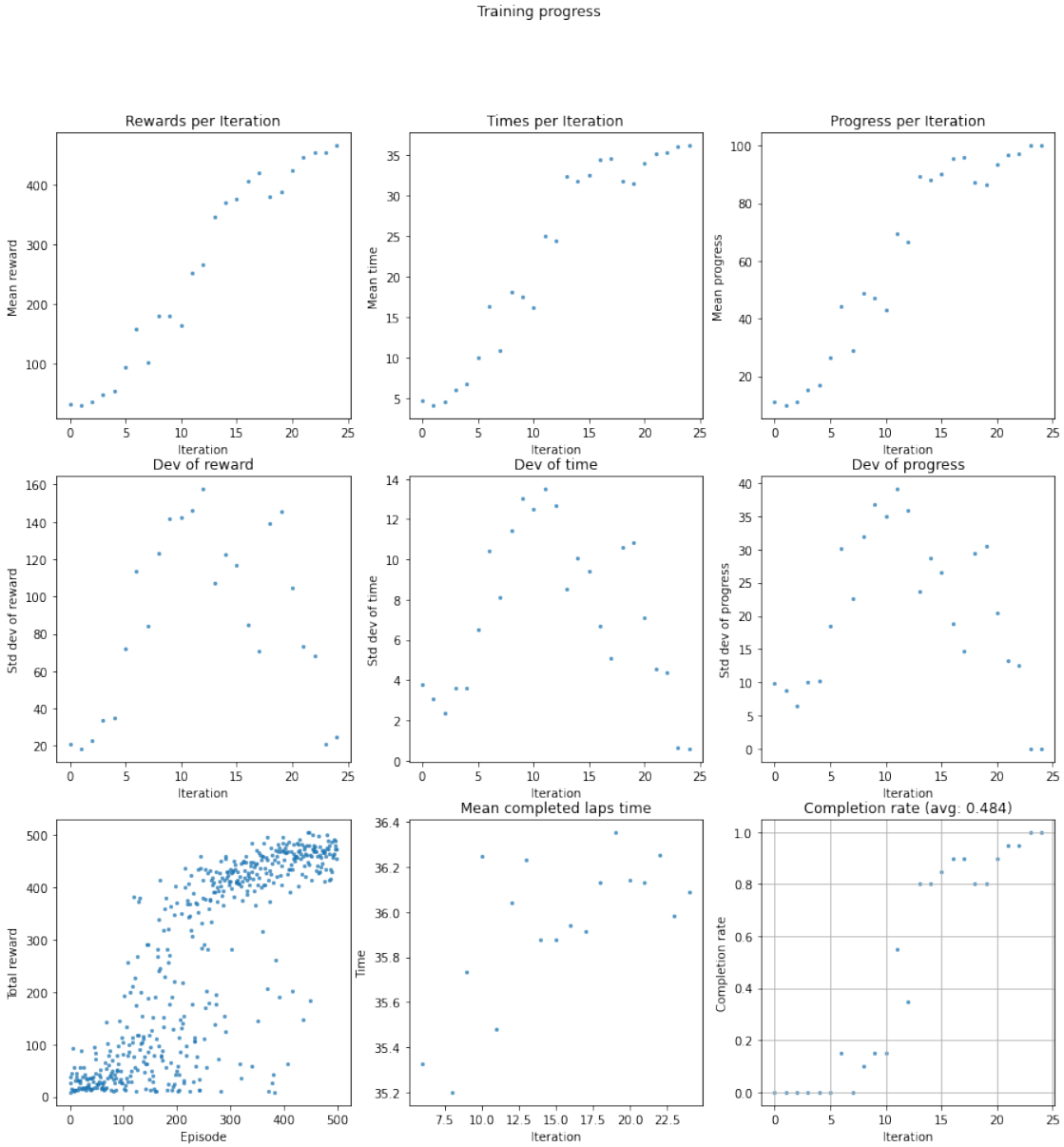
Mean completed lap times per iteration Once we have a model that completes laps reasonably often, we might want to know how fast the car gets around the track. This graph will show you that. I use it quite often when looking for a model to shave a couple more miliseconds. That said it has to go in pair with the last one:

Completion rate per iteration It represents how big part of all episodes in an iteration is full laps. The value is from range [0, 1] and is a result of deviding amount of full laps in iteration by amount of all episodes in iteration. I say it has to go in pair with the previous one because you not only need a fast lapper, you also want a race completer.

The higher the value, the more stable the model is on a given track.

[11]:
```
simulation_agg = au.simulation_agg(df)

au.analyze_training_progress(simulation_agg, title='Training progress')
```

```
new reward not found, using reward as its values
Number of episodes =  499
Number of iterations =  24
```




```
<Figure size 432x288 with 0 Axes>
```

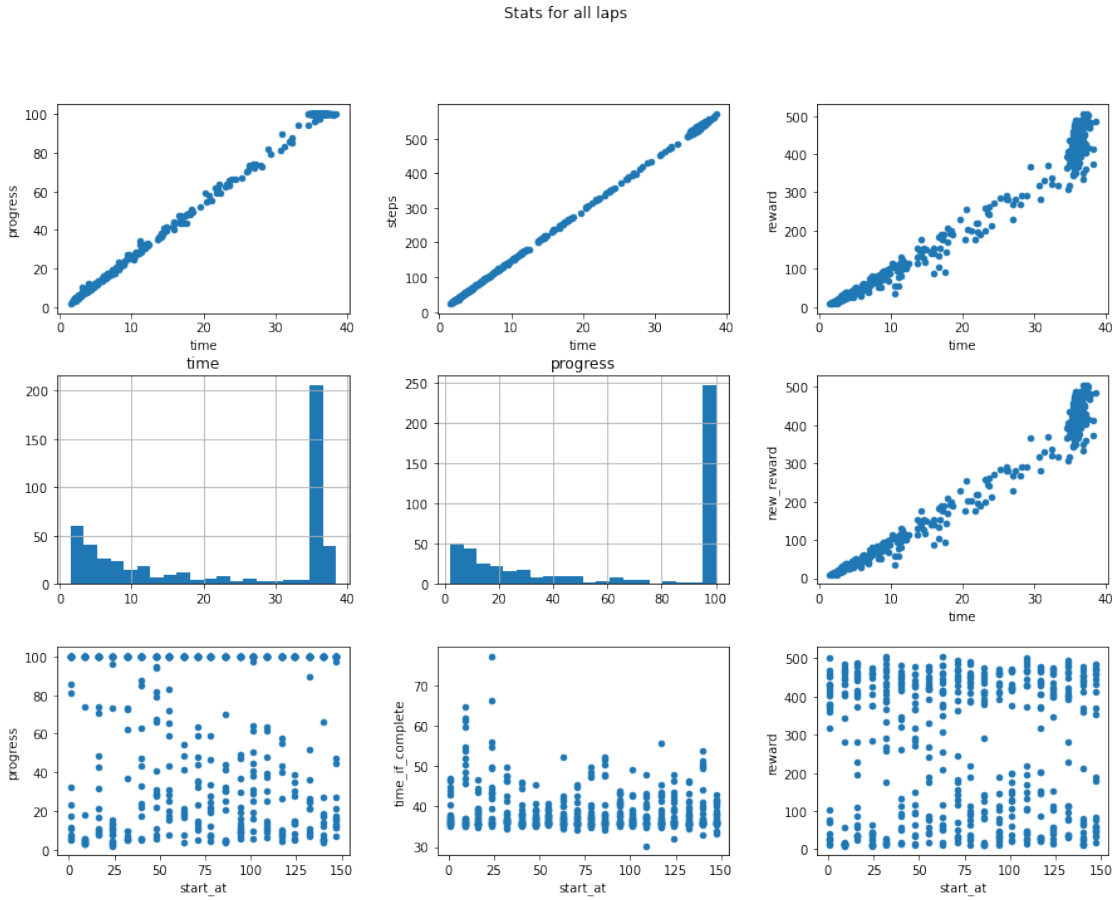
2.6.2 Stats for all laps

Previous graphs were mainly focused on the state of training with regards to training progress. This however will not give you a lot of information about how well your reward function is doing overall.

In such case `scatter_aggregates` may come handy. It comes with three types of graphs: * progress/steps/reward depending on the time of an episode - of this I find reward/time and new_reward/time especially useful to see that I am rewarding good behaviours - I expect the reward to time scatter to look roughly triangular * histograms of time and progress - for all episodes the progress one is usually quite handy to get an idea of model's stability * progress/time_if_complete/reward to closest waypoint at start - these are really useful during training as they show potentially problematic spots on track. It can turn out that a car gets best reward (and performance) starting at a point that just cannot be reached if the car starts elsewhere, or that there is a section of a track that the car struggles to get past and perhaps it's caused by an aggressive action space or undesirable behaviour prior to that place

Side note: `time_if_complete` is not very accurate and will almost always look better for episodes closer to 100% progress than in case of those 50% and below.

[12]:
```
au.scatter_aggregates(simulation_agg, 'Stats for all laps')
```

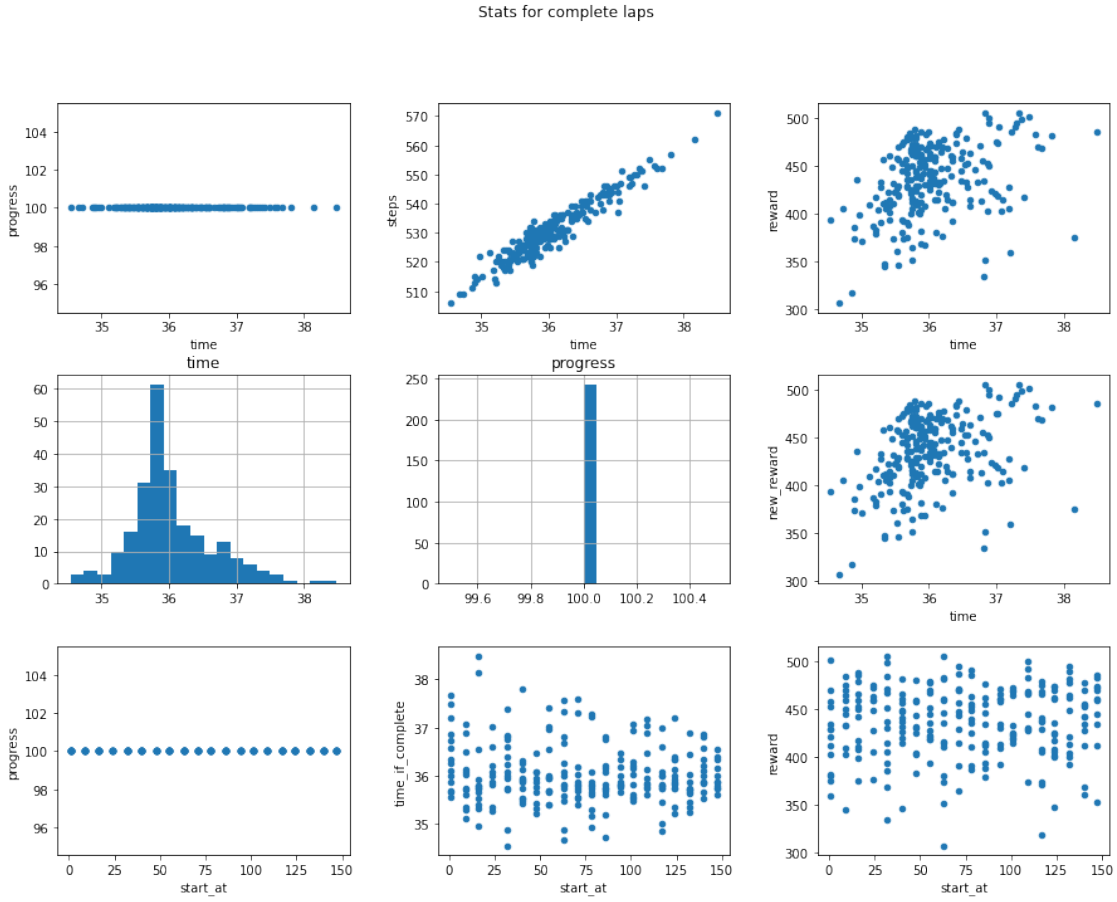



```
<Figure size 432x288 with 0 Axes>
```

2.6.3 Stats for complete laps

The graphs here are same as above, but now I am interested in other type of information: * does the reward scatter show higher rewards for lower completion times? If I give higher reward for a slower lap it might suggest that I am training the car to go slow * what does the time histogram look like? With enough samples available the histogram takes a normal distribution graph shape. The lower the mean value, the better the chance to complete a fast lap consistently. The longer the tails, the greater the chance of getting lucky in submissions * is the car completing laps around the place where the race lap starts? Or does it only succeed if it starts in a place different to the racing one?

[13]:
```
complete_ones = simulation_agg[simulation_agg['progress']==100]

if complete_ones.shape[0] > 0:
    au.scatter_aggregates(complete_ones, 'Stats for complete laps')
else:
    print('No complete laps yet.')
```




```
<Figure size 432x288 with 0 Axes>
```

2.6.4 Categories analysis

We're going back to comparing training results based on the training time, but in a different way. Instead of just scattering things in relation to iteration or episode number, this time we're grouping episodes based on a certaing information. For this we use function:

`analyze_categories(panda, category='quintile', groupcount=5, title=None)`

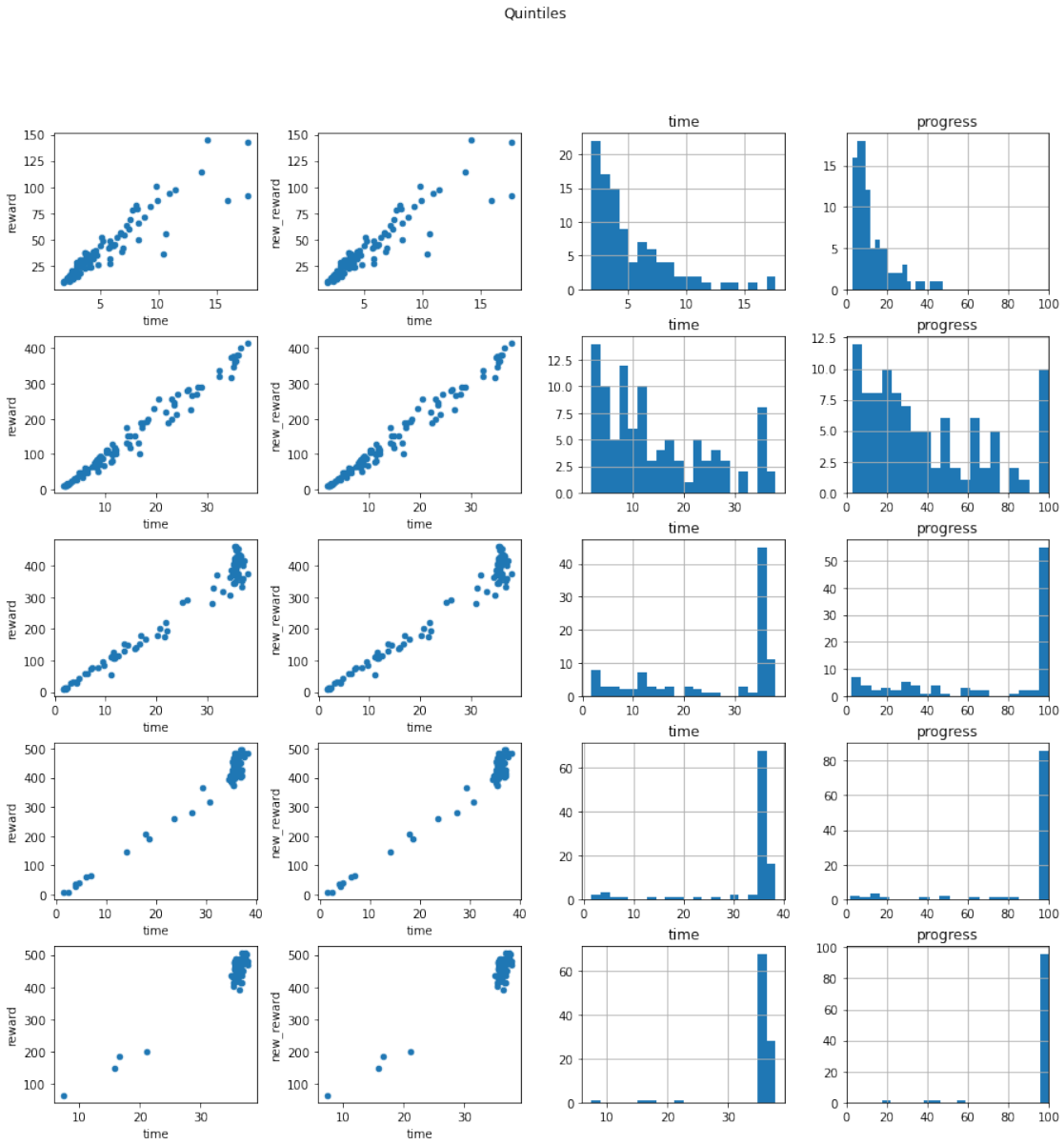
The idea is pretty simple - determine a way to cluster the data and provide that as the `category` parameter (alongside the count of groups available). In the default case we take advantage of the aggregated information to which quintile an episode belongs and thus build buckets each containing 20% of episodes which happened around the same time during the training. If your training lasted for five hours, this would show results grouped per each hour.

A side note: if you run the function with `category='start_at'` and `groupcount=20` you will get results based on the waypoint closest to the starting point of an episode. If you need to, you can introduce other types of categories and reuse the function.

The graphs are similar to what we've seen above. I especially like the progress one which shows where the model tends to struggle and whether it's successful laps rate is improving or beginning to decrease. Interestingly, I also had cases where I saw the completion drop on the progress rate only to improve in a later quintile, but with a better time graph.

A second side note: if you run this function for `complete_ones` instead of `simulation_agg`, suddenly the time histogram becomes more interesting as you can see whether completion times improve.

[14]:
```
au.scatter_by_groups(simulation_agg, title='Quintiles')
```



<Figure size 432x288 with 0 Axes>

2.7 Data in tables

While a lot can be seen in graphs that cannot be seen in the raw numbers, the numbers let us get into more detail. Below you will find a couple examples. If your model is behaving the way you would like it to, below tables may provide little added value, but if you struggle to improve your car's performance, they may come handy. In such cases I look for examples where high reward is giving to below-expected episode and when good episodes are given low reward.

You can then take the episode number and scatter it below, and also look at reward given per step - this can in turn draw your attention to some rewarding anomalies and help you detect some

unexpected outcomes in your reward function.

There is a number of ways to select the data for display: * `nlargest/nsmallest` lets you display information based on a specific value being highest or lowest * filtering based on a field value, for instance `df[df['episode']==10]` will display only those steps in `df` which belong to episode 10 * `head()` lets you peek into a dataframe

There isn't a right set of tables to display here and the ones below may not suit your needs. Get to know Pandas more and have fun with them. It's almost as addictive as DeepRacer itself.

The examples have a short comment next to them explaining what they are showing.

[15]:
```
# View ten best rewarded episodes in the training
simulation_agg.nlargest(10, 'new_reward')
```

[15]:

	iteration	episode	steps	start_at	progress	time	new_reward	\
444	22	444	543.0	32	100.0	36.828290	505.2	
448	22	448	552.0	63	100.0	37.342633	504.9	
480	24	480	555.0	1	100.0	37.490687	500.7	
454	22	454	541.0	109	100.0	36.890030	500.1	
464	23	464	551.0	32	100.0	37.374708	498.8	
397	19	397	543.0	132	100.0	36.886872	494.9	
369	18	369	550.0	71	100.0	37.301218	494.8	
494	24	494	541.0	109	100.0	37.038880	491.3	
430	21	430	550.0	78	100.0	37.274288	490.4	
497	24	497	536.0	132	100.0	36.440738	488.6	

	speed	reward	time_if_complete	reward_if_complete	quintile	complete
444	0.6	505.2	36.828290	505.2	5th	1
448	0.6	504.9	37.342633	504.9	5th	1
480	0.6	500.7	37.490687	500.7	5th	1
454	0.6	500.1	36.890030	500.1	5th	1
464	0.6	498.8	37.374708	498.8	5th	1
397	0.6	494.9	36.886872	494.9	4th	1
369	0.6	494.8	37.301218	494.8	4th	1
494	0.6	491.3	37.038880	491.3	5th	1
430	0.6	490.4	37.274288	490.4	5th	1
497	0.6	488.6	36.440738	488.6	5th	1

[16]:
```
# View five fastest complete laps
complete_ones.nsmallest(5, 'time')
```

[16]:

	iteration	episode	steps	start_at	progress	time	new_reward	\
344	17	344	506.0	32	100.0	34.546920	393.02	
228	11	228	509.0	63	100.0	34.667655	306.56	
351	17	351	509.0	86	100.0	34.722943	405.08	
175	8	175	511.0	117	100.0	34.864801	317.62	
128	6	128	513.0	63	100.0	34.889527	373.52	

```
     speed  reward  time_if_complete  reward_if_complete quintile  complete
344    0.6  393.02         34.546920              393.02      4th         1
228    0.6  306.56         34.667655              306.56      3rd         1
351    0.6  405.08         34.722943              405.08      4th         1
175    0.6  317.62         34.864801              317.62      2nd         1
128    0.6  373.52         34.889527              373.52      2nd         1
```

[17]:
```
# View five best rewarded completed laps
complete_ones.nlargest(5, 'reward')
```

[17]:
```
     iteration  episode  steps  start_at  progress       time  new_reward  \
444         22      444  543.0        32     100.0  36.828290       505.2
448         22      448  552.0        63     100.0  37.342633       504.9
480         24      480  555.0         1     100.0  37.490687       500.7
454         22      454  541.0       109     100.0  36.890030       500.1
464         23      464  551.0        32     100.0  37.374708       498.8

     speed  reward  time_if_complete  reward_if_complete quintile  complete
444    0.6   505.2         36.828290               505.2      5th         1
448    0.6   504.9         37.342633               504.9      5th         1
480    0.6   500.7         37.490687               500.7      5th         1
454    0.6   500.1         36.890030               500.1      5th         1
464    0.6   498.8         37.374708               498.8      5th         1
```

[18]:
```
# View five best rewarded in completed laps (according to new_reward if you are␣
↪using it)
complete_ones.nlargest(5, 'new_reward')
```

[18]:
```
     iteration  episode  steps  start_at  progress       time  new_reward  \
444         22      444  543.0        32     100.0  36.828290       505.2
448         22      448  552.0        63     100.0  37.342633       504.9
480         24      480  555.0         1     100.0  37.490687       500.7
454         22      454  541.0       109     100.0  36.890030       500.1
464         23      464  551.0        32     100.0  37.374708       498.8

     speed  reward  time_if_complete  reward_if_complete quintile  complete
444    0.6   505.2         36.828290               505.2      5th         1
448    0.6   504.9         37.342633               504.9      5th         1
480    0.6   500.7         37.490687               500.7      5th         1
454    0.6   500.1         36.890030               500.1      5th         1
464    0.6   498.8         37.374708               498.8      5th         1
```

[19]:
```
# View five most progressed episodes
simulation_agg.nlargest(5, 'progress')
```

[19]:
```
     iteration  episode  steps  start_at  progress       time  new_reward  \
120          6      120  529.0         1     100.0  35.883607    380.9426
```

```
128          6      128  513.0        63     100.0  34.889527    373.5200
131          6      131  514.0        86     100.0  35.204030    379.0000
175          8      175  511.0       117     100.0  34.864801    317.6200
178          8      178  525.0       140     100.0  35.534641    359.9000

     speed    reward  time_if_complete  reward_if_complete quintile  complete
120    0.6  380.9426         35.883607            380.9426      2nd         1
128    0.6  373.5200         34.889527            373.5200      2nd         1
131    0.6  379.0000         35.204030            379.0000      2nd         1
175    0.6  317.6200         34.864801            317.6200      2nd         1
178    0.6  359.9000         35.534641            359.9000      2nd         1
```

[20]:
```
# View information for a couple first episodes
simulation_agg.head()
```

[20]:
```
   iteration  episode  steps  start_at   progress       time  new_reward  \
0          0        0   56.0         1   7.840225   3.651709     27.6884
1          0        1   29.0         9   3.134118   1.876096      9.4278
2          0        2   56.0        16   8.292591   3.678375     37.5458
3          0        3   77.0        24  11.581545   5.022693     44.2464
4          0        4   36.0        32   4.826853   2.410421     15.5266

   speed   reward  time_if_complete  reward_if_complete quintile  complete
0    0.6  27.6884         46.576590          353.158243      1st         0
1    0.6   9.4278         59.860423          300.811871      1st         0
2    0.6  37.5458         44.357364          452.763194      1st         0
3    0.6  44.2464         43.368076          382.042301      1st         0
4    0.6  15.5266         49.937738          321.671262      1st         0
```

[21]:
```
# Set maximum quantity of rows to view for a dataframe display - without that
# the view below will just hide some of the steps
pd.set_option('display.max_rows', 500)

# View all steps data for episode 10
df[df['episode']==10]
```

[21]:
```
     episode  steps         x         y    heading  steering_angle  speed  \
826       10    1.0  8.268747  4.396783  75.034348           -15.0    0.6
827       10    2.0  8.303151  4.454062  70.897635            15.0    0.6
828       10    3.0  8.303151  4.454062  70.897635             0.0    0.6
829       10    4.0  8.324448  4.490759  69.031535           -30.0    0.6
830       10    5.0  8.330093  4.500213  68.555151           -30.0    0.6
831       10    6.0  8.360516  4.549855  66.280756           -15.0    0.6
832       10    7.0  8.379801  4.577778  65.047383            30.0    0.6
833       10    8.0  8.405204  4.612198  63.265690            15.0    0.6
834       10    9.0  8.431871  4.654362  62.138433             0.0    0.6
835       10   10.0  8.451370  4.681748  61.238293            30.0    0.6
```

836	10	11.0	8.477036	4.722731	60.627401	0.0	0.6
837	10	12.0	8.496280	4.758167	60.765811	-30.0	0.6
838	10	13.0	8.520272	4.793629	59.992667	-30.0	0.6
839	10	14.0	8.541651	4.823166	59.123177	0.0	0.6
840	10	15.0	8.566147	4.855193	58.053408	15.0	0.6
841	10	16.0	8.594283	4.893471	57.230923	30.0	0.6
842	10	17.0	8.613539	4.924073	57.288346	0.0	0.6
843	10	18.0	8.634466	4.960203	57.734062	0.0	0.6
844	10	19.0	8.659073	5.001890	58.110875	0.0	0.6
845	10	20.0	8.678570	5.036555	58.534986	-30.0	0.6
846	10	21.0	8.700031	5.069709	58.301596	-30.0	0.6
847	10	22.0	8.727246	5.108039	57.631145	0.0	0.6
848	10	23.0	8.753735	5.142268	56.692578	-30.0	0.6
849	10	24.0	8.775768	5.167796	55.655882	-30.0	0.6
850	10	25.0	8.812356	5.204719	53.481735	-30.0	0.6
851	10	26.0	8.840839	5.230651	51.752883	-30.0	0.6
852	10	27.0	8.875911	5.256237	49.016198	-15.0	0.6
853	10	28.0	8.909186	5.280430	46.817050	-15.0	0.6
854	10	29.0	8.946724	5.304774	44.378264	0.0	0.6
855	10	30.0	8.985564	5.329146	42.135086	-30.0	0.6
856	10	31.0	9.025292	5.352949	40.046005	0.0	0.6
857	10	32.0	9.064571	5.373393	37.922779	15.0	0.6
858	10	33.0	9.096695	5.391239	36.575861	-15.0	0.6

	action	reward	done	all_wheels_on_track	progress	closest_waypoint	\
826	1	1.0000	False	True	0.611198	78	
827	3	1.0000	False	True	0.885757	79	
828	2	1.0000	False	True	0.885757	79	
829	0	0.8000	False	True	1.054688	79	
830	0	0.8000	False	True	1.099355	79	
831	1	1.0000	False	True	1.334505	80	
832	4	0.8000	False	True	1.468494	80	
833	3	1.0000	False	True	1.624263	80	
834	2	0.5000	False	True	1.813888	80	
835	4	0.4000	False	True	1.942968	80	
836	2	0.5000	False	True	2.134007	81	
837	0	0.4000	False	True	2.277847	81	
838	0	0.4000	False	True	2.417180	81	
839	2	0.5000	False	True	2.552053	81	
840	3	0.5000	False	True	2.698755	82	
841	4	0.4000	False	True	2.873691	82	
842	2	0.1000	False	True	2.931379	82	
843	2	0.1000	False	True	3.081850	82	
844	2	0.1000	False	True	3.262812	83	
845	0	0.0800	False	True	3.413269	83	
846	0	0.0800	False	True	3.557241	83	
847	2	0.1000	False	True	3.584855	83	

848	0	0.0800	False	True	3.713432	83
849	0	0.0800	False	True	3.815494	83
850	0	0.0800	False	True	3.961406	84
851	0	0.0800	False	True	4.062874	84
852	1	0.0010	False	False	4.160471	84
853	1	0.0010	False	False	4.238288	84
854	2	0.0010	False	False	4.238288	84
855	0	0.0008	False	False	4.238288	84
856	2	0.0010	False	False	4.238288	84
857	3	0.0010	False	False	4.238288	84
858	1	0.0010	True	False	4.267462	84

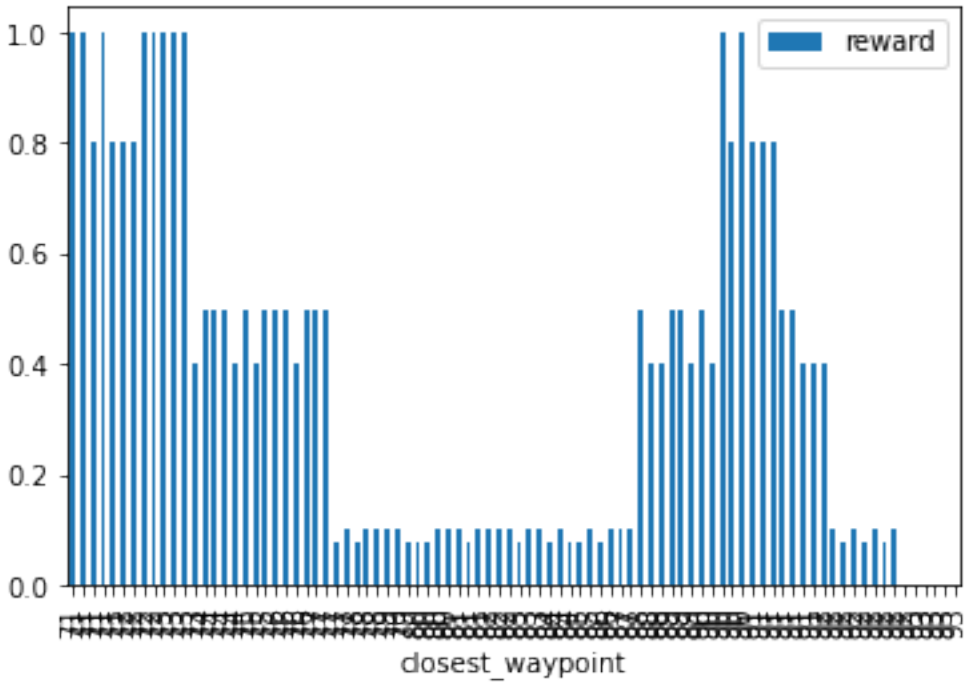
	track_len	tstamp	episode_status	iteration	worker	\
826	23.118222	1.613889e+09	in_progress	0	0	
827	23.118222	1.613889e+09	in_progress	0	0	
828	23.118222	1.613889e+09	in_progress	0	0	
829	23.118222	1.613889e+09	in_progress	0	0	
830	23.118222	1.613889e+09	in_progress	0	0	
831	23.118222	1.613889e+09	in_progress	0	0	
832	23.118222	1.613889e+09	in_progress	0	0	
833	23.118222	1.613889e+09	in_progress	0	0	
834	23.118222	1.613889e+09	in_progress	0	0	
835	23.118222	1.613889e+09	in_progress	0	0	
836	23.118222	1.613889e+09	in_progress	0	0	
837	23.118222	1.613889e+09	in_progress	0	0	
838	23.118222	1.613889e+09	in_progress	0	0	
839	23.118222	1.613889e+09	in_progress	0	0	
840	23.118222	1.613889e+09	in_progress	0	0	
841	23.118222	1.613889e+09	in_progress	0	0	
842	23.118222	1.613889e+09	in_progress	0	0	
843	23.118222	1.613889e+09	in_progress	0	0	
844	23.118222	1.613889e+09	in_progress	0	0	
845	23.118222	1.613889e+09	in_progress	0	0	
846	23.118222	1.613889e+09	in_progress	0	0	
847	23.118222	1.613889e+09	in_progress	0	0	
848	23.118222	1.613889e+09	in_progress	0	0	
849	23.118222	1.613889e+09	in_progress	0	0	
850	23.118222	1.613889e+09	in_progress	0	0	
851	23.118222	1.613889e+09	in_progress	0	0	
852	23.118222	1.613889e+09	in_progress	0	0	
853	23.118222	1.613889e+09	in_progress	0	0	
854	23.118222	1.613889e+09	in_progress	0	0	
855	23.118222	1.613889e+09	in_progress	0	0	
856	23.118222	1.613889e+09	in_progress	0	0	
857	23.118222	1.613889e+09	in_progress	0	0	
858	23.118222	1.613889e+09	off_track	0	0	

	unique_episode	new_reward
826	10	1.0000
827	10	1.0000
828	10	1.0000
829	10	0.8000
830	10	0.8000
831	10	1.0000
832	10	0.8000
833	10	1.0000
834	10	0.5000
835	10	0.4000
836	10	0.5000
837	10	0.4000
838	10	0.4000
839	10	0.5000
840	10	0.5000
841	10	0.4000
842	10	0.1000
843	10	0.1000
844	10	0.1000
845	10	0.0800
846	10	0.0800
847	10	0.1000
848	10	0.0800
849	10	0.0800
850	10	0.0800
851	10	0.0800
852	10	0.0010
853	10	0.0010
854	10	0.0010
855	10	0.0008
856	10	0.0010
857	10	0.0010
858	10	0.0010

2.8 Analyze the reward distribution for your reward function

[22]:
```
# This shows a histogram of actions per closest waypoint for episode 889.
# Will let you spot potentially problematic places in reward granting.
# In this example reward function is clearly `return 1`. It may be worrying
# if your reward function has some logic in it.
# If you have a final step reward that makes the rest of this histogram
# unreadable, you can filter the last step out by using
# `episode[:-1].plot.bar` instead of `episode.plot.bar`
episode = df[df['episode']==9]

if episode.empty:
```

```
    print("You probably don't have episode with this number, try a lower one.")
else:
    episode.plot.bar(x='closest_waypoint', y='reward')
```



2.8.1 Path taken for top reward iterations

NOTE: at some point in the past in a single episode the car could go around multiple laps, the episode was terminated when car completed 1000 steps. Currently one episode has at most one lap. This explains why you can see multiple laps in an episode plotted below.

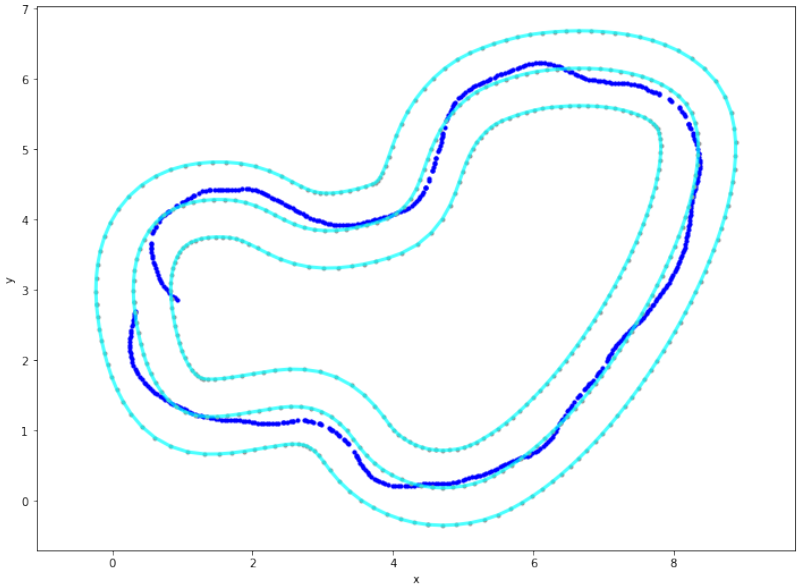
Being able to plot the car's route in an episode can help you detect certain patterns in its behaviours and either promote them more or train away from them. While being able to watch the car go in the training gives some information, being able to reproduce it after the training is much more practical.

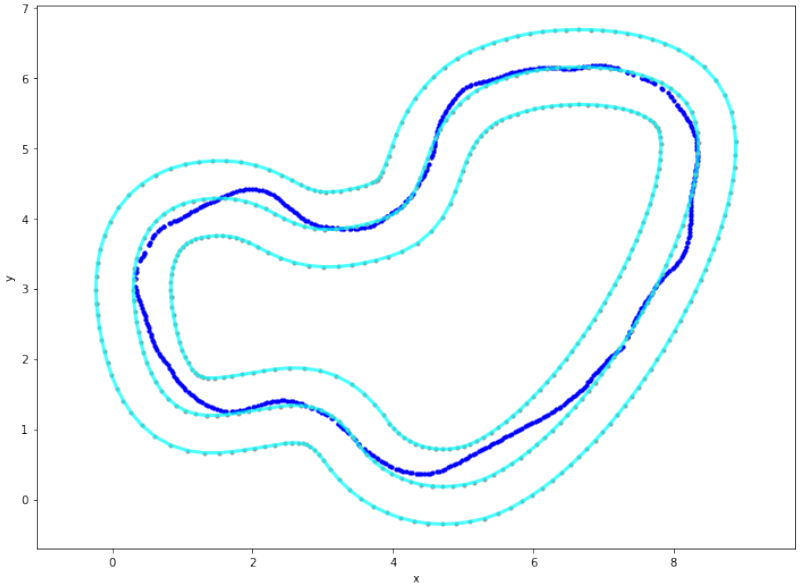
Graphs below give you a chance to look deeper into your car's behaviour on track.

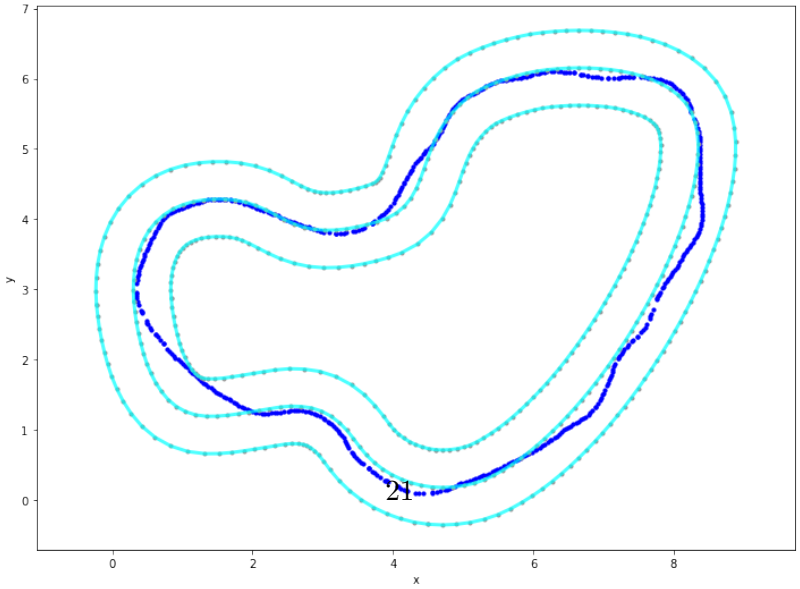
We start with plot_selected_laps. The general idea of this block is as follows: * Select laps(episodes) that have the properties that you care about, for instance, fastest, most progressed, failing in a certain section of the track or not failing in there, * Provide the list of them in a dataframe into the plot_selected_laps, together with the whole training dataframe and the track info, * You've got the laps to analyse.

[23]:
```
# Some examples:
# highest reward for complete laps:
# episodes_to_plot = complete_ones.nlargest(3,'reward')
```

```
# highest progress from all episodes:
episodes_to_plot = simulation_agg.nlargest(3,'progress')

pu.plot_selected_laps(episodes_to_plot, df, track)
```


y
x


y
x


y
x

```
<Figure size 432x288 with 0 Axes>
```

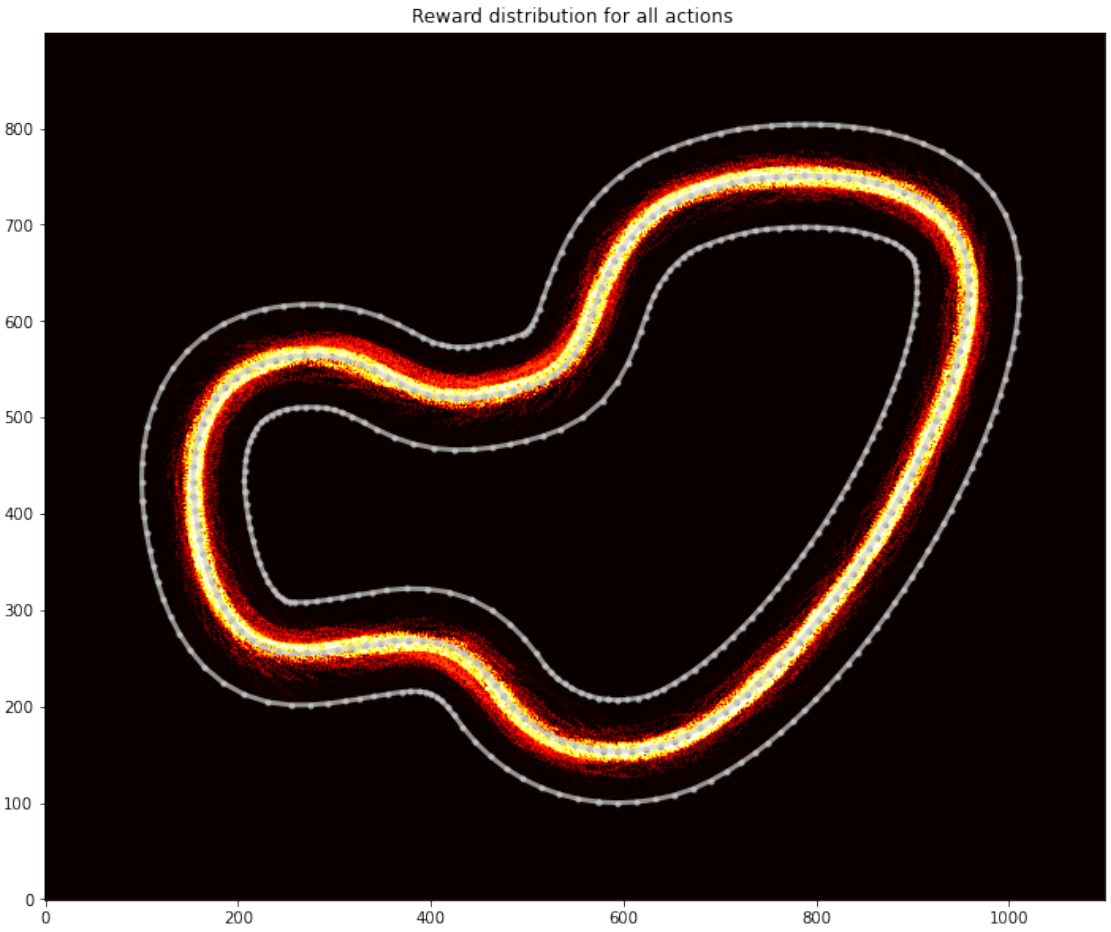
2.8.2 Plot a heatmap of rewards for current training.

The brighter the colour, the higher the reward granted in given coordinates. If instead of a similar view as in the example below you get a dark image with hardly any dots, it might be that your rewards are highly disproportionate and possibly sparse.

Disproportion means you may have one reward of 10.000 and the rest in range 0.01-1. In such cases the vast majority of dots will simply be very dark and the only bright dot might be in a place difficult to spot. I recommend you go back to the tables and show highest and average rewards per step to confirm if this is the case. Such disproportions may not affect your traning very negatively, but they will make the data less readable in this notebook.

Sparse data means that the car gets a high reward for the best behaviour and very low reward for anything else, and worse even, reward is pretty much discrete (return 10 for narrow perfect, else return 0.1). The car relies on reward varying between behaviours to find gradients that can lead to improvement. If that is missing, the model will struggle to improve.

[24]:
```
#If you'd like some other colour criterion, you can add
#a value_field parameter and specify a different column

pu.plot_track(df, track)
```

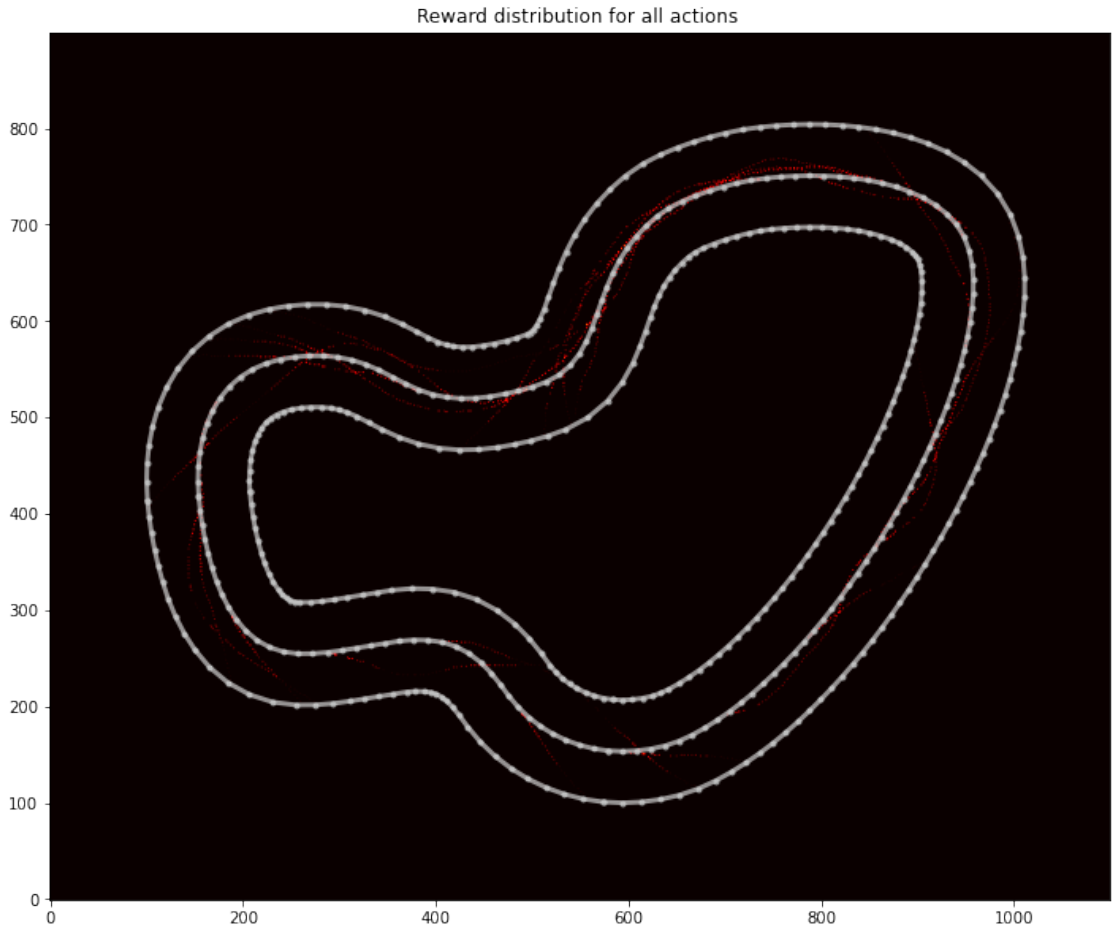



```
<Figure size 432x288 with 0 Axes>
```

2.8.3 Plot a particular iteration

This is same as the heatmap above, but just for a single iteration.

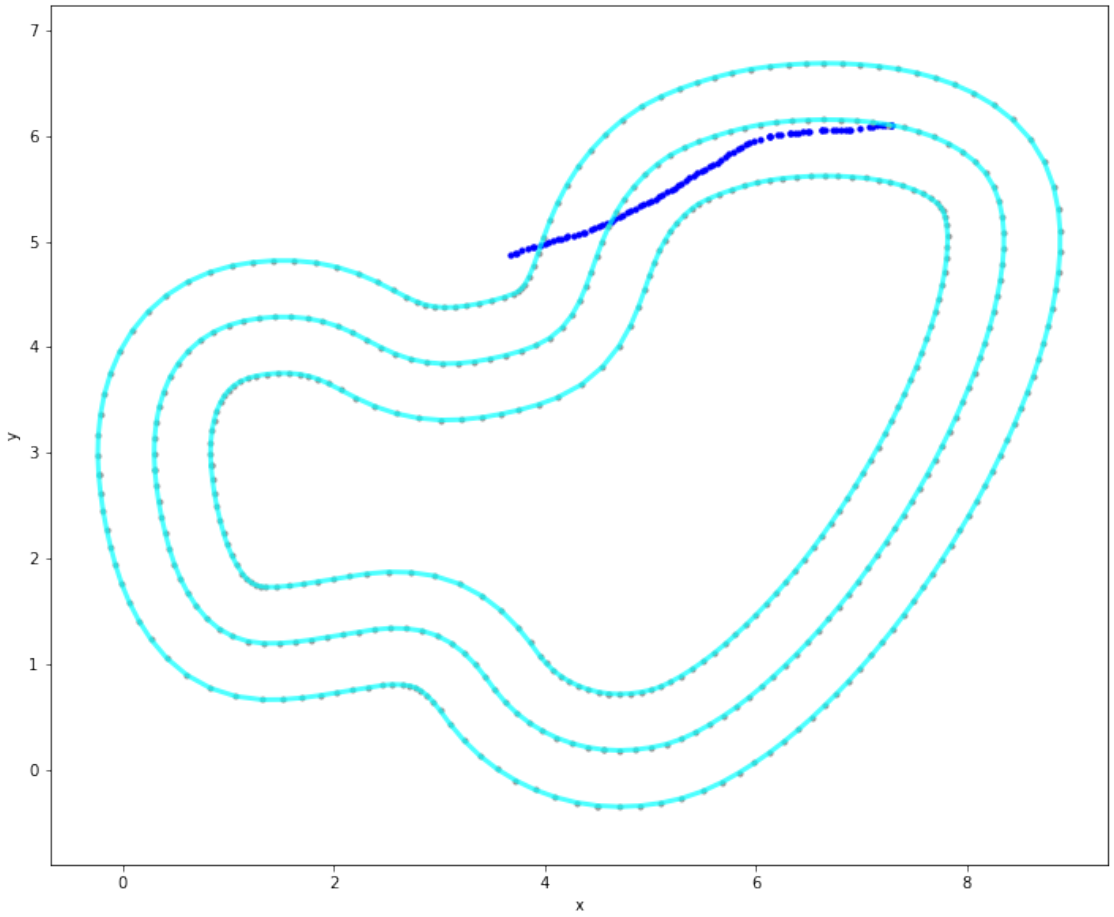
[25]:
```
#If you'd like some other colour criterion, you can add
#a value_field parameter and specify a different column
iteration_id = 3

pu.plot_track(df[df['iteration'] == iteration_id], track)
```




```
<Figure size 432x288 with 0 Axes>
```

2.8.4 Path taken in a particular episode

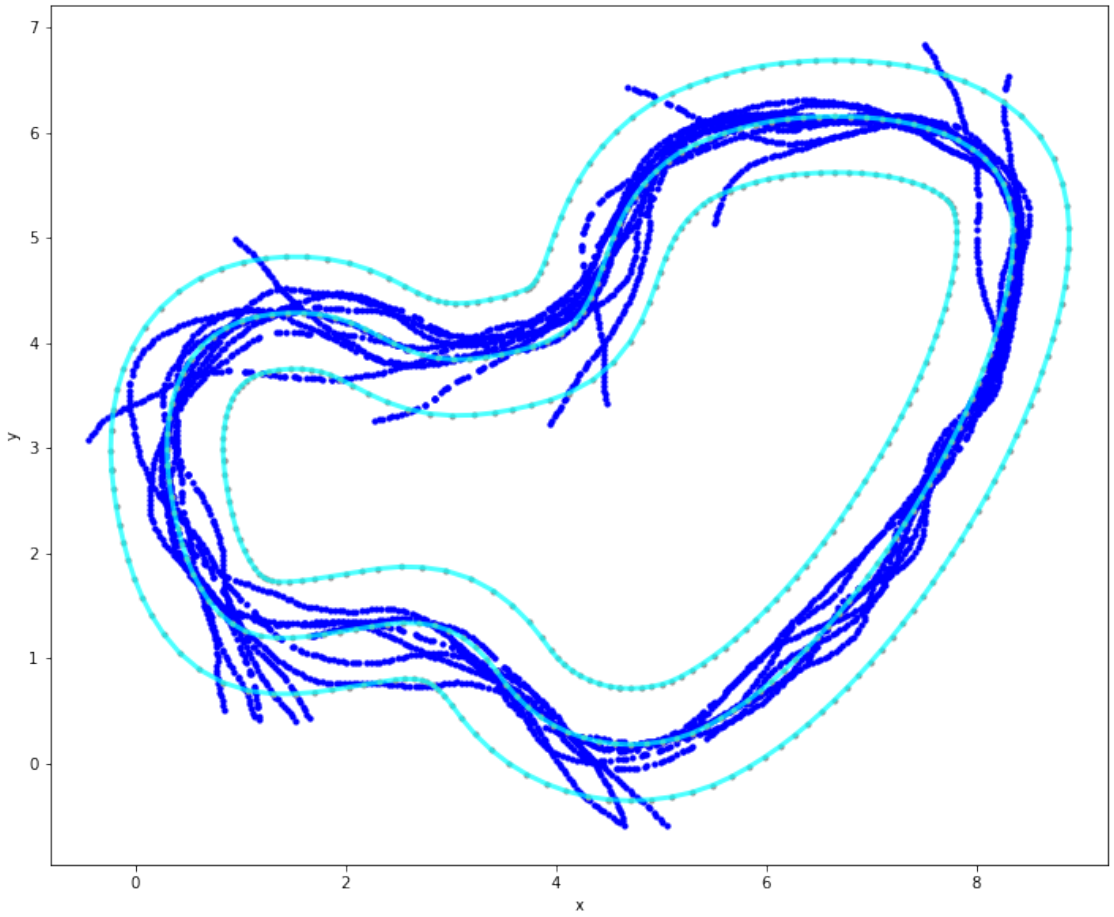
[26]:
```
episode_id = 12

pu.plot_selected_laps([episode_id], df, track)
```



<Figure size 432x288 with 0 Axes>

2.8.5 Path taken in a particular iteration

[27]:
```
iteration_id = 10

pu.plot_selected_laps([iteration_id], df, track, section_to_plot = 'iteration')
```




```
<Figure size 432x288 with 0 Axes>
```

3 Action breakdown per iteration and historgram for action distribution for each of the turns - reinvent track

This plot is useful to understand the actions that the model takes for any given iteration. Unfortunately at this time it is not fit for purpose as it assumes six actions in the action space and has other issues. It will require some work to get it to done but the information it returns will be very valuable.

This is a bit of an attempt to abstract away from the brilliant function in the original notebook towards a more general graph that we could use. It should be treated as a work in progress. The track_breakdown could be used as a starting point for a general track information object to handle all the customisations needed in methods of this notebook.

A breakdown track data needs to be available for it. If you cannot find it for the desired track, MAKEIT.

Currently supported tracks:

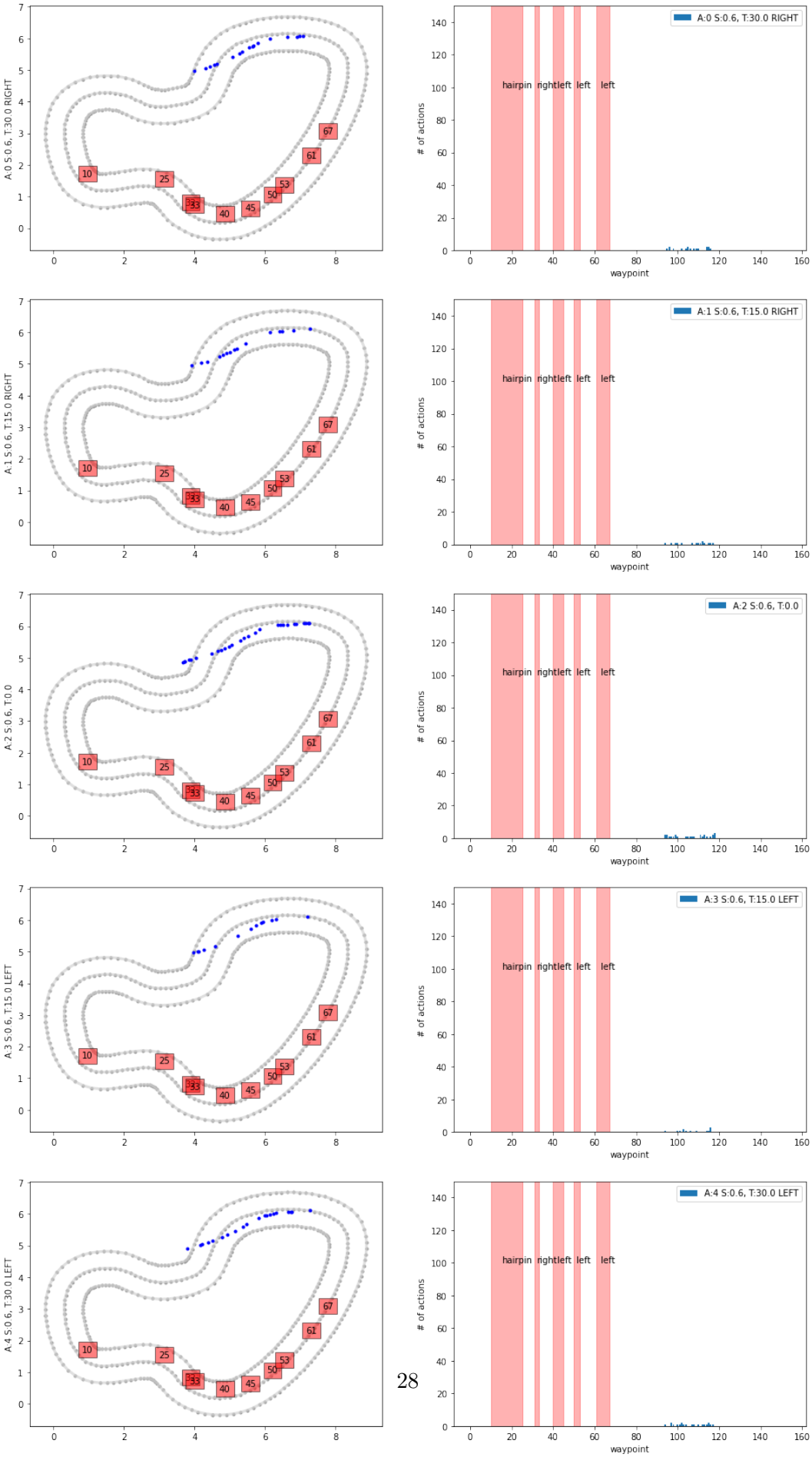
[29]:
```
track_breakdown.keys()
```

[29]:
```
dict_keys(['reinvent2018', 'london_loop'])
```

You can replace episode_ids with iteration_ids and make a breakdown for a whole iteration.

Note: does not work for continuous action space (yet).

[30]:
```
abu.action_breakdown(df, track, track_breakdown=track_breakdown.
↪get('reinvent2018'), episode_ids=[12])
```


A:0 S:0.6, T:30.0 RIGHT
A:1 S:0.6, T:15.0 RIGHT
A:2 S:0.6, T:0.0
A:3 S:0.6, T:15.0 LEFT
A:4 S:0.6, T:30.0 LEFT
hairpin
rightleft
left
left
# of actions
waypoint

```
<Figure size 432x288 with 0 Axes>
```

[ ]: