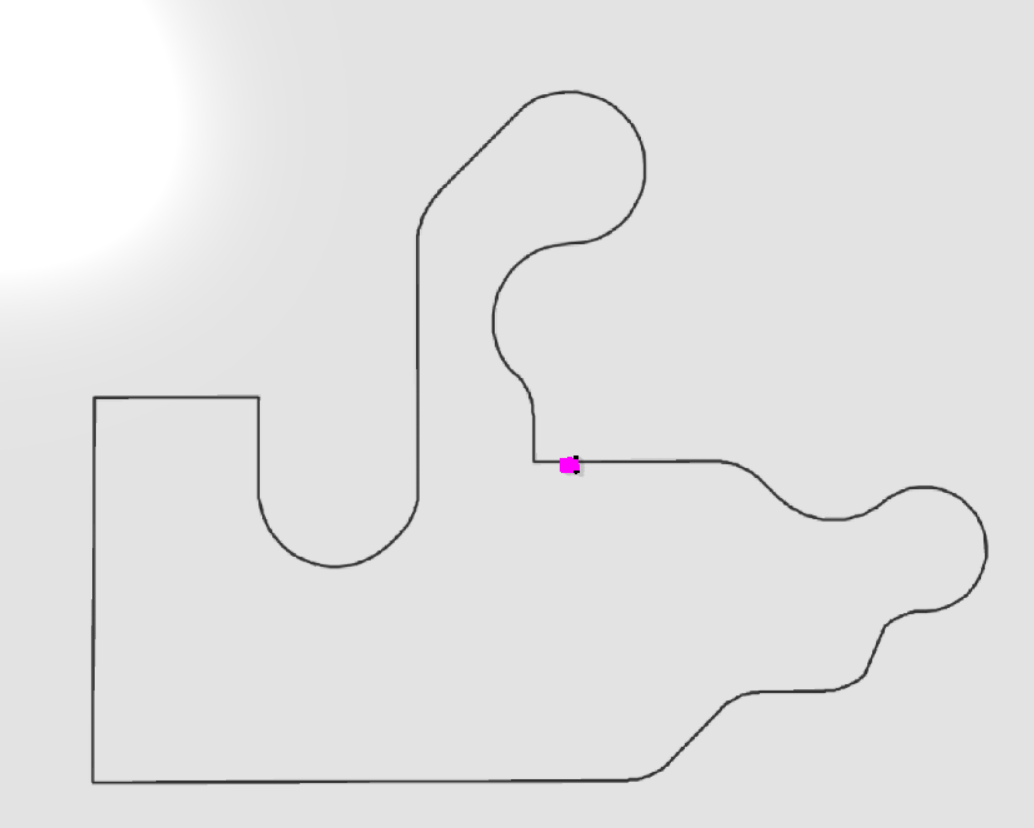
**OpenAI LineFollower SmartSystems 2019-2020**

Gilles Lenaerts, Jonas Aerts

This paper is focussed on the Linefollower in OpenAI Gym & Pybullet[1] used as starting point and is adapted to fullfill the first steps to create an OpenAI enviroment and model based on the Renesas MCU Rally track[2].

**Software install**

Visualstudio.microsoft.com/downloads, Community downloader

* MSVC v142 – VS 2019 C++ x64/x86 build tools (v14.23 or v14.24) OR MSVC v140 VS 2015 C++ build tools (v14.00)
* Windows 10SDK (10.0.16299.0)
* Windows universal CRT SDK

**Python packages**

To check library list “pip freeze”

* clone gym\_line\_follower from github -> “Pip install –e .”
* Tensorflow requires version 1.14.0, “pip install tensorflow==1.14.0”
* conda install -c powerai gym & pip install pybullet
* pip install git+https://github.com/benelot/pybullet-gym

**Running errors**

* 'cannot re-register id: Linefollower-v0', comment "\gym\_line\_follower" in \_init\_.py in directory. If running through an IDE, the kernel does not reset the ID's. When running through CLI there is no errors.

**LineFollower\_Env**

The enviroment has observations, actions, reset, render, … functions like most openAI-gym enviroments. This script is the source of training the linefollower bot and is working together with all the individual python scripts (enviroment parts) that will be explained throughout the paper.

Actions :

* yaw left
* yaw right
* Control speed (power limit)

Observations :

* Linefollower position on track
* Progress made on track
* bot yaw
* Checkpoints passed
* Reward

**LineFollower\_Bot**

* The initializer creates the bot with the pybullet client functions to the local physics server. Different parameters to set the starting XY coordinates as well the yaw of the bot on the track. The amount of cam(sensors) points for line detection and in what kind of observation type we would like to keep track of the progress.
* Reset, get position, apply actions, get pov image, ... and more functions to control the bots procedure.

**Track generation**

*“Generate(cls, approx\_width, hw\_ratio, seed , irregularity, spikeyness, num\_verts ,\*args, \*\*kwargs)”*:

This function generates all points that are used to draw the track. In the sample code the track is randomly generated. To make a custom track following steps have to be done.

* Clsis defined as 1 when calling the function.
* Approx\_width is the width of the track. This is defined as 1 and will later be upscaled.
* Hw\_ratio is the height to width ratio (y – x). this is defined as 1 so the x and y axis are the same.
* Seed, irregularity en spikeyness are used to randomly generate a track. To make sure there are no irregulatiries these variables are defined as 0 when they are used and seed will not be used at all.
* num\_verts defines the resolution of the curves. This variable defines the amount of points the curve is made off

“(*get\_bezier\_curve(a, rad=0.2, edgy=0)*”:

The bezier function returns 2 arrays, one with x coordinates and one with y coordinates which are stored accordingly “x, y, \_= get\_bezier\_curve(pts, rad=0, edgy=0)”. Straight lines and corners are then made with the steps below.

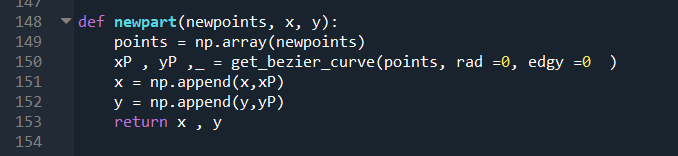
* corners and lines are generated with the bezier curve function. This function needs an array of coordinates (variable a).
* A line just needs 2 coordinates a begin and end.
* A corner needs 3 (2 end points and a corner points). This array is converted to a numpy array (pts) and is used in the function as “get\_bezier\_curve(pts, rad = 0 , edgy = 0)”. In our code we append the next ‘trackpart’ to the previous with the function ‘newpart(newpoints, x, y)’. This so the last X and Y are taken automatically.

“generate\_polygon(ctrx , ctry , aveRadius, irregularity = 0,spikeyness = 0, numverts , kwad, Reverse)”:

To build up the points for turns & curves a different function is used.

* ctrx en ctr y are the center coordinates of the circle that is generated.
* Radius is half diameter of the circle.
* Irregularity and spikeyness are defined as 1 so there are no irregularities.
* Numverts is the resolution of the curve (example : 180).
* kwad is used to define which part of the circle is used / what the starting point is (0-7 \* 45° offset).
* The resulting array is then converted to a numpy array.
* Invert is set to 1 if u want to draw a right turn (to flip the array pts = pts[::-1]) and 0 when u want to draw a left turn
* Amount is the number of 45° slices drawn from a circle’s starting from offset (kwad).

“‘newpart(newpoints, x, y)’”:



The next array of given points is converted and appended to the final X and Y coordinates of the complete track.

* newpoints is the given array of defined by the user, converted to a numpy array (points) and passed as “xP,yP, \_ = get\_bezier\_curve(points, rad=0, edgy=0)”.
* The result of the Bezier function are the x coordinates and y coordinates stored in xP, yP.
* xP and yP are appended to the last X and Y coordinates of the track and returned.

After track pieces duplicates get removed.

* x = x[:-1]
* y = y[:-1]

x and y array get scaled and stacked.

* unit\_scale = 180 (the lower the larger the track)
* x, y = x / unit\_scale, y / unit\_scale
* pts = np.stack((x, y), axis=-1)
* check = if the track is over its boundaries in the enviroment.

“Render()” :

This function is used to connect the points resulting from the generate function.

**Running the program**

Because the paper is focused on the Renesas MCU rally track and only generated and training on the same track, some changes have been made in the sample code. This so the development of the AI model is faster.

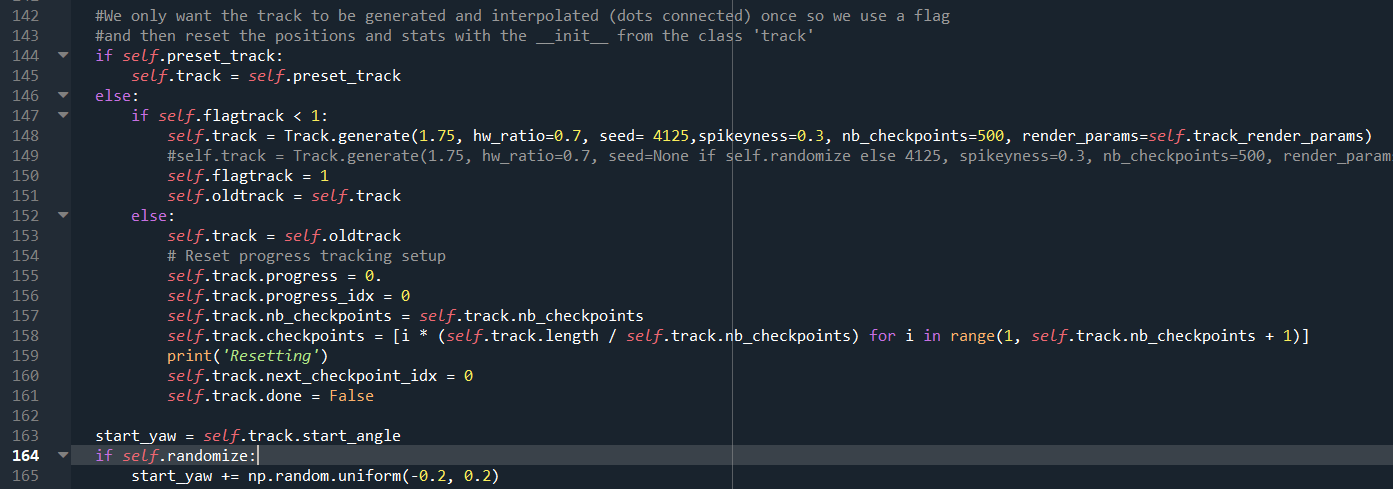
**track.py**

* function get\_bezier\_curve() comment a = ccw\_sort(a).
* function get\_curve make sure the for loop is “for i in range(0,(len(points)-2))”.this makes it so that the first and last points in the array don’t get connected by a line so we can draw a corner instead of a triangle.
* unit\_scale has been set to 180, but can be changed accordingly depening on the size of the track parts.
* In the check for boundaries change ‘<1,5’ to ‘<3.5’. Keep track of how big the enviroment is.

**line\_follower\_env.py**

This script is the main source of controls in the pybullet client (physics enviroment).

* in \_\_init\_\_() add:
  + self.flagtrack =0
  + self.oldtrack = track
* Line 144 defines random positions and track generations. Change this to the code shown below for using the track (mcu) made in ‘track.py’.

****

The flagtrack is used so the generation is only done once as this takes the longest because of connecting the dots. When the Linefollower crashes or has too many error it will reset its position and progress, but not the track.

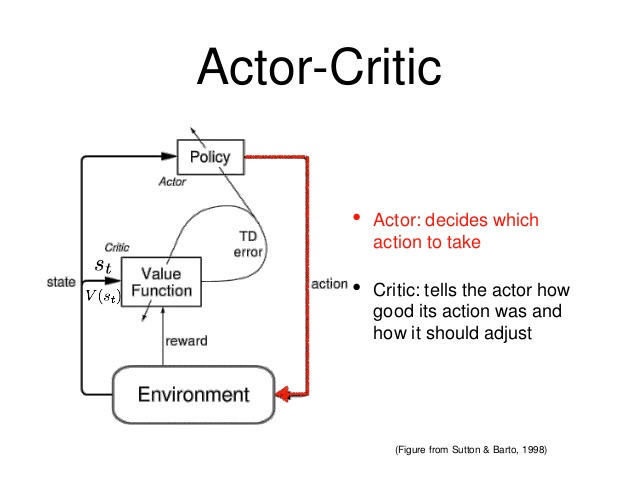
Changing how strict the error for linefollower is and update the model is done within the parameters of the \_\_init\_\_() from the linefollower\_env. This is put on 0.03 since you can’t really go off-track in the Renesas MCU Rally track and so taking clean corners is important (train a model on high error rate first and go lower in new models).

**Training a model**

Changing the track to the MCU is done with previous chapter. When the track is changed we can train a model with the ‘ddpg.py’ script:

* The linefollower gets imported as gym enviroment, a network is created with the build\_agent().
* The keras-rl library has different agents to choose from. the ddpg agent is chosen and imported as DPPGAgent.
* With the DPPGAgent we can send the agent an enviroment and it will look the at the enviroment actions available. The network can execute these actions and train with the rewards gotten from the observation of the Linefollower\_env.

The networks used exists of a model like this :



**Critic network**

* Analyzes the action the enviroment took together with the observations and rewards returned. the output is a value to the actor which tells if it did well or not.

**Actor network**

* This network takes the input from the critic network and chooses an action based on the feedback it got.

the actual DDPGAgent is created using the actor, critic, random process, and other parameters. It is compiled with an Adam optimizer with a learning rate of 0.003 and the mean absolute error metric.

In theory, this agent should quickly learn that returning the values observed will provide the highest reward and how to react on a result.

The train func():

* calls the build\_agent()
* Loads old weights if defined
* Saves weights on intervals as checkpoints.

After training the weights the model is tested with 20 episodes (predefined).

**Loading a saved pretrained model**

Change the pretrained\_path variable to the dir folder of the models trained. In train function add "agent.load\_weights(os.path.join(pretrained\_path,"weights"))". 'Weights' is the name of your file with weights to be loaded.

**Model Graphs**

<https://www.youtube.com/watch?v=YGQqh7mmWb4>

In CLI go to ‘LineFollower\_SmartSystems\Opdracht’ and type ‘tensorboard --logdir=logs --port 6006’. This will return an adress to open the tensboard in a browser and show steps and rewards from the different model logs. Here the logs of the the model training are graphically displayed so we can see the progress in steps taken and rewards gotten over time. Some optimizing can be done to also display the loss & mae curves. This is a very usefull tool !

**Results**

**Github to project:** [**https://github.com/Gilleslenaerts007/LineFollower\_SmartSystems**](https://github.com/Gilleslenaerts007/LineFollower_SmartSystems)

The first weights were trained on a three-piece track with a 90° corner, straight lines and simple curves. This track is saved in code ‘RechthoekTrack.py’. Changing the track is explained in chapter Track generation.

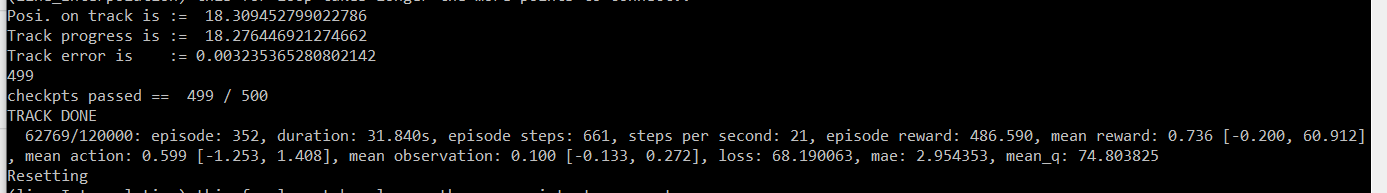
* Parameters for agent = (size=nb\_actions, theta=.4, mu=0.2, sigma=0.3).

After training 120000 steps, the model executed these obstacles ok, but not great. This model is saved in directory ‘firsttracker’. Then a first half of the renesas track is made, implemented into the enviroment and started training. Performance was good at start, but degraded after more steps.

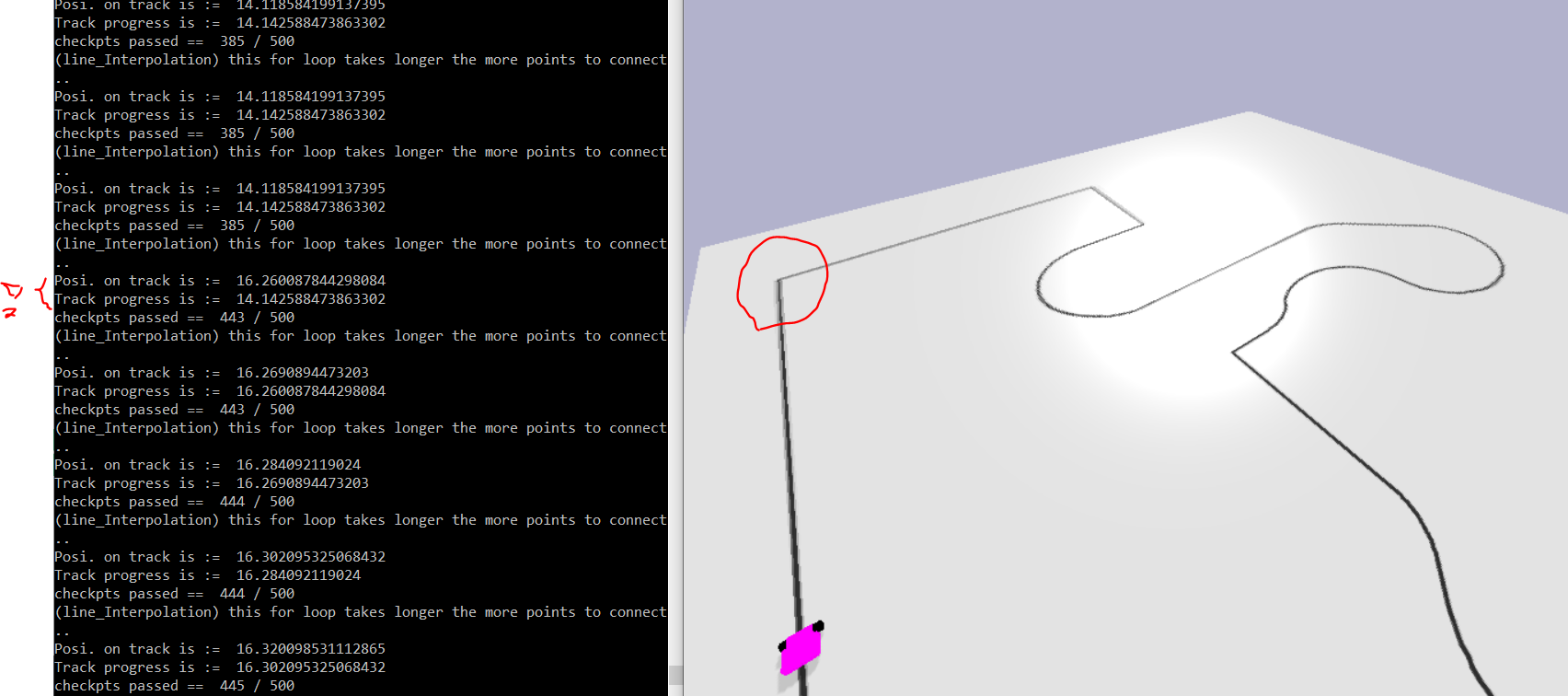
Changing the parameters of the agent works better and completes the track in ~45seconds.

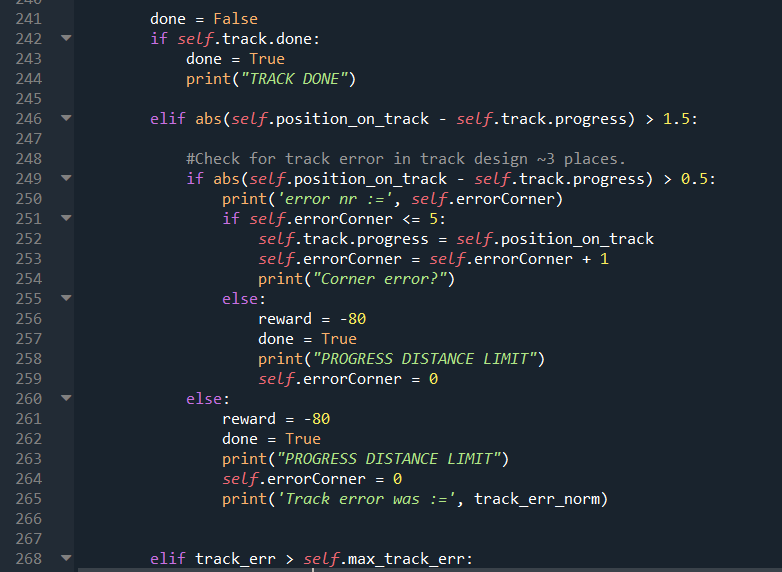
* Parameters for agent = (size=nb\_actions, theta=.2, mu=0.2, sigma=0.1).

By lowering the ‘max\_track\_err’ (going offtrack etc.) from 0.3 to 0.03 and the power limit from 0.4 to 0.7 in the \_\_init\_\_() from the linefollower. We got as expected faster laps and completes the track in ~33seconds. Training a good model takes a lot of time and correct agent parameters. Learning these factors is an important rule !



Sometimes the position on track vs the progress of the track is glitched and increases with +- 1.5 for unkown reason.



For this reason the code in linefollower\_env.py on line 226 is changed so the it can complete a full cycle. 

* from ‘if self.position\_on\_track - self.track.progress < 0.4:’
* into ‘if self.position\_on\_track - self.track.progress < 1.5:’

A dubbel check is implemented, shown in the picture on the side, so the model learns to stay on track with an error of <0.5.

**References**

[1] <https://github.com/nplan/gym-line-follower>

[2] <https://renesasrulz.com/university/mcurally/>