Using Reinforcement Learning and 2D LiDAR to Control a Vehicle and Avoid Obstacles in Carla.

A. Bloom, Dr. Y. Vorobeychik

Washington University in Saint Louis

To train a reinforcement learning algorithm to drive by itself and avoid obstacles I built a framework to handle and log everything efficiently. It contains a control panel to design experiments, a trainer to carry out and log those experiments and an environment in which to carry out those experiments. The code is open source and can be found [here](https://github.com/A-Bloom/CARLA_LiDAR_RL.py).

The Environment:

To begin, let’s start from the ground up with the environment. [Carla](https://carla.readthedocs.io/en/0.9.15/) is an open-source realistic driving simulator to help program or train autonomous vehicles. It is made up of two parts; a server which is the simulator and a client which is a python API. Everything was run on Carla 0.9.15. The server is built on the Unreal Engine and you might need to install legacy DirectX drivers [here](https://www.microsoft.com/en-gb/download/details.aspx?id=35&irgwc=1&OCID=AIDcmm549zy227_aff_7815_119570&tduid=%28ir__wf1t6jfdiwkfdzevufswalhllm2xdqxerm0icgxq00%29%287815%29%28119570%29%285728363%29%28lwqs5u6ave03es170tipy%29&irclickid=_wf1t6jfdiwkfdzevufswalhllm2xdqxerm0icgxq00). To access the python API, pip and import ‘carla’. The server can run on the same computer as the client or they can communicate over TCP. The default port for accessing the server is 2000 and the traffic manager communicates through port 2001.

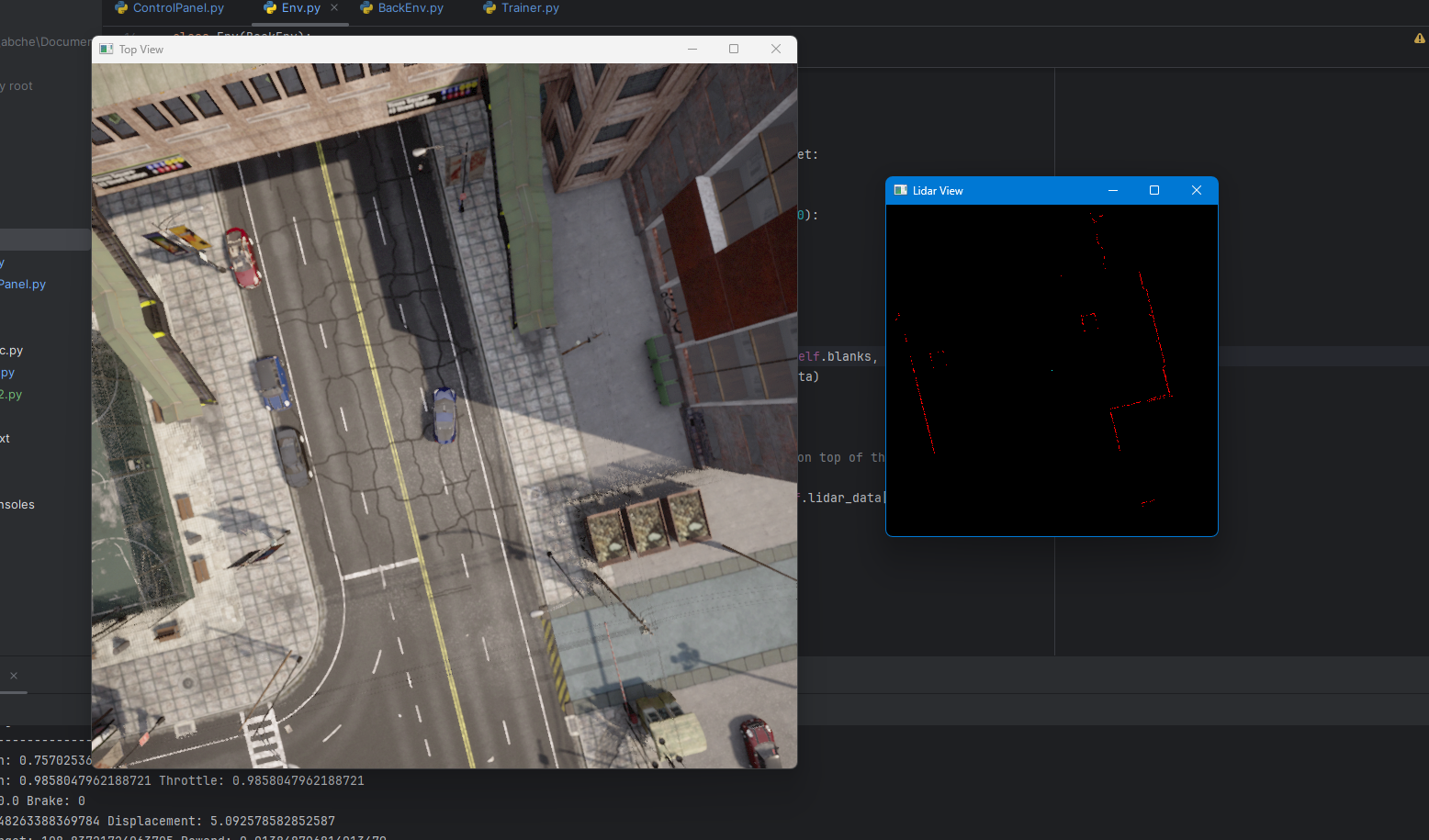
Everything related to connecting and setting up Carla is handled by BackEnv.py. It inherits the base [gymnasium](https://gymnasium.farama.org/) class and follows the standard gymnasium structure. By default, the server runs in asynchronous mode where it renders frames as quickly as possible irrespective of the state of the client. For our purposes we need to run the server in synchronous mode. In this mode the client controls the server and updates it by calling the tick() function. This is done so that the algorithm can update or save without the server continuing to simulate. This also allows you to run the simulation faster than real time at the expense of less precise physics. This is done by changing settings.delta\_seconds which determines how many seconds the server simulates per frame. Because the server is running in synchronous mode it needs to wait for a signal from the client to continue and can’t preprocess images which leaves us with a frame rate of about 30 FPS. Currently delta\_seconds is set at 0.05 seconds/frame, so the simulation is running at 1.5x normal speed. The visuals are a little choppy, but the physics seems to be fine. To place vehicles, sensors or objects in the world, first find a blueprint in the blueprint library. You can then change the blueprint, like changing the color, to meet your specifications. You then specify a location and rotation in the world to place instances of the blueprint in the world. To start listeners for sensors, use a lambda function and specify where to send the data. For some sensors, like LiDAR, there is a constant stream of data going to the function, but for other sensors, like the collision sensor, the function is only triggered when an event occurs.

The most complex piece of this project is designing observation, action and reward functions so that the agent learns the correct thing to do. This is all handled in Env.py. I created each function with multiple possibilities and if I thought of another possibility, I just added it to the code and set it as a possible hyperparameter in the control panel. This idea will become clearer later. In the initialization function the size of the observation and action spaces are set up. This must be done because they are fixed throughout training and the agent needs to know their sizes to set up the neural network properly.

The step function is called every step of the training loop. The first piece is a bit to try to restart the server if it crashes. Then it moves on to the action processing. Different algorithms can provide different types of action possibilities. Some algorithms can only provide continuous actions, others can only provide discrete actions and certain algorithms can do both. I added logic to handle any situation. The Carla car takes throttle from 0 to 1 (with a shifter for reverse), steering from -1 to 1 and brake from 0 to 1. For continuous actions this was pretty simple to translate (with the exception of reverse) because the actions coming in are normalized from -1 to 1. For discrete actions I remapped each action to a position on the throttle, steering or brake. The other environmental hyperparameter I added here was “action\_possibilities” which controls which actions the car can access. It has values 0 for only steering (constant throttle), 1 for throttle forward and steering, 2 for throttle (forward and backward) and steering, and 3 for throttle, steer and break. The reason I did this is because sometimes if the agent needs to figure out too many things at the same time it can’t get anything correct. This way you can first train the agent to steer correctly before giving it control over the throttle.

The reward structure comes next and is probably the most crucial for getting the agent to behave properly. So far, I have added four things you can punish the agent for and three things you can reward it for. When the agent collides with something, it automatically gets -1 and the episode ends. If it is not going fast enough you can also punish it. This is necessary because I found that sometimes the agent would find a spot where it could get some reward and just stop there. The negative reward for low speed keeps it constantly moving. There is a negative reward for being too close to an object which is basically a non-terminal extension of the collision punishment. You can also punish it for turning, which I will discuss in a moment. For positive reward it can get a reward for speed, displacement (being further from its original position) and for getting closer to a destination. The amount of reward it gets for each of these can be set by the experimenter before the run begins. When I just had speed, I found that the agent just learned to do really fast donuts. This is an example where the agent maximizes reward, by going fast and not crashing, but doesn’t learn what you want it to learn. To help with that I added displacement which sometimes helped but other times just taught the agent to make bigger donuts. To help I added location, which should guide it in a specific direction, and the punishment for turning. Other options I am considering are adding a reward for heading toward the destination. There are other variables that the experimenter can fine-tune to help the training. One variable of particular note is exponentialize\_reward. It allows you to put the reward to some exponent. This can be helpful in incentivizing the agent to go for the really big reward instead of just collecting a bunch of small rewards.

The observation is a compilation of LiDAR measurements and is handled by the function create\_lidar\_plane. LiDAR comes in as (x, y) distance points from the agent. These can then be processed to create a top-down LiDAR image like the one shown to the right. A complication in doing so is that each point is represented by a continuous number representing the distance in the x and y directions while a picture is made of individual pixels from a 2D array and can only be represented by a finite resolution. To deal with this I made Lidar\_Resolution a hyperparameter that can be tested to see if the resolution has any effect on performance. To manipulate the LiDAR from (x, y) points to an image, I first multiplied all the points by the resolution and found the closest integer to each point. Then I shifted all the points so that they were all positive. This is because the points started in all four quadrants, but an image is only represented in one quadrant with the top left corner being (0, 0). Then I used the rounded points as indexes and set all the pixels at those indexes to 1. One problem I faced is that the LiDAR packets each come in with a different number of points, but the observation needs to stay the same size, and I wanted each observation to contain enough points for a 360° view. To deal with this I created a buffer that fills with points until a specified number. This number can also be specified as a hyperparameter, but I found the best number to be about 250. Until the buffer fills the environment produces the old observation. When it is done filling the buffer becomes the observation and the process starts again. When setting up the experiment the experimenter has three options for observations to give the agent. The unchanged points, a single layer 2D grid and a three-layer 2D grid, otherwise understood as a normalized image. The reason I did it like that is because each has pros and cons. The more complex the observation gets the more preprocessing that needs to be done and the more features the algorithm needs to deal with. This can increase training time significantly. Another drawback is that the policy is bigger. For images I found that a 30x30 image policy saved 100 times is a GB. Raising the resolution or running the algorithm multiple times can increase your data consumption exponentially. On the other hand, just (x, y) points don’t give any indication of spatial relationship between the points so the algorithm might not be able to solve our problem at all with just points. Image processing algorithms get around this because they first send the picture through convolutional layers. This allows the algorithm to make decisions based on segmented areas of the image. For my specific problem a single layered 2D grid should be enough because each point is either 0 or 1 and I don’t need to deal with multiple color channels. However, many algorithms are designed specifically for images so creating a three channeled image format might help them learn. To do that I just stacked blank arrays on top of the first one. As an extra bonus, in the blank arrays I added ones in the very center to represent the agent. I don’t know if it makes any difference, but it definitely makes it easier for the experimenter to understand where the vehicle is. I speculate that any one of these observation types could work so I am going to try all of them. To give it color for the final image that the experimenter sees, I multiplied the entire thing by 255. This gives the obstacle points in the top red channel full color and the center points in the blue and green channel full color while leaving everything else black.



Agent

Obstacles

ControlPanel.py is just a list of all possible hyperparameters. To test only one option, leave it as a single value. To test multiple options, put the values in an array. To run an experiment, run the ControlPanel directly and it will hand all the information to the trainer. Trainer.py handles all of the training and logging. First it does some housekeeping by creating folders and files with timestamps and saving the experiment to a log file. Then it launches the TensorBoard. TensorBoard is a useful tool built specifically for tracking machine learning experiments. It displays everything you log from your experiment including the length and reward of each episode against time steps. It runs as a server so that you can continuously update it with the newest information coming from the agent. To launch it use python -m tensorboard.main --logdir=/path/to/logdir. I just have my code launch it automtically. Next I get every possible experiment with the VariableUnion function which basically loops through the input variables and creates a new array every time there is a variable with multiple possibilities. The code then loops through every run, algorithm, experiment, algorithm configuration and epoch. The whole time it is timestamping, logging and saving everything. Optionally if you have an experimet that was aborted you can pick up from where it stopped by running Trainer.py directly from command line with the –resume flag and the path to the aborted experiment as the only parameter. You can also upgrade a fully completed policy .zip file with the –upgrade flag, a parameter to the .zip file, and positional arguments for epochs and steps per epoch that you want to continue training. Note that this will create a new experiment and TensorBoard.

The Agent:

Each algorithm that I tuned the most was PPO so the algorithm hyperparameters that I focused on where gamma, batch size, learning rate, clip range, entrophy coefficient and buffer size. Gamma is the reward discount. When an RL agent is determining if the next action should be taken, it also needs to consider the impact this action will have on the reward in future steps as well. The agent should take a metocre action now if long term it will be fine instead of taking a really good action now that will lead to a position of negative reward later. However, we want the agent to prioritize the next reward somewhat over future reward because future reward is less certain. This is where the reward discount (λ) comes in. The agent calculates its assumed total reward at step 0 as: . Essentially gamma determines how much the agent prioritizes the current steps reward over future reward. Batch size is the number of steps that the agent progresses through and analyzes before each update. Learning rate is how much the agent updates its policy based on a single step or batch. If the agent gives too much weight to any single step or batch you run the risk of swinging back and forth to much and never converging on the solution. However, the smaller the update the longer it will take to train. Clip rate sets a limit to the maximum change an update can have on the policy. This stops outliers from ruining an otherwise stable learning curve. The entropy coefficinet determines how the agent handles the explore/exploit trade off. Exploring is when the agent makes random choices to explore all its options and find what rewards are available. Exploiting is when the agent does what it thinks will maximize reward. You don’t want to explore too little because you run the risk of falling into a local reward maximum and not finding the global reward maximum. For example in a hypothetical environment, right gets the agent +1 reward and left gets the agent -1 reward, but going left four times produces a +100 reward. If the agent only exploits it will always choose right because it thinks that is the best option and it will never get the chance to find the +100 reward. However, you can’t explore too much or the agent will never do the correct thing. To deal with this the algorithm slowly changes from all exploration to all exploitation. The rate of this change is determined by the entropy coefficient. When an agent needs to determine its next move it looks at a replay buffer to determine what actions led to which rewards. PPO does not rely on a replay buffer but it is important for other algorithms. The buffer size hyperparameter controls how big that buffer is. If the buffer is too small you run the risk of significant action reward pairs (like finding a particularly big reward) getting dropped from the buffer and the agent no longer understands how to find that reward. On the other hand having to large of a buffer will cause the algorithm to slow down significantly because it needs to process all the information in the buffer.

Hyperparameter Tuning:

There are a couple of methods to go about hyperparameter tuning. The simplest ones are random search, grid search and manual search. Random search just chooses random hyperparameters and hopes that it works. This can be faster than grid search but is similar to playing the lottery. Grid search takes a range of possible hyperparameters and computes all the possible combinations. This can take an extremely long time but will ensure that you have covered all the options and can easily find the best option. My code is set up to easily do a grid search, but I usually only did it with a relatively small number of options. More complex automated options include a Bayesian search or hyperband tuning where the code automatically selects the next hyperparameters based on the results of the previous run. I used a combination of the first three. First, I ran a couple of random runs to find a starting point to train from. Then I used a grid search in that localized area to find trends. Finally, I read the graphs to determine whether to increase or decrease each hyperparameter and manually tweaked each one.

A graph on a screen

Description automatically generatedA graph with lines and numbers

Description automatically generated The main graph to examine to determine that the algorithm is learning is the reward graph. On the TensorBoard it is labeled as rollout/ep\_rew\_mean. The result that needs to be achieved is a steady increase in reward over time. Figure 1 is an example of a good learning curve even though it is a bit choppy. The x axis in every graph of the TensorBoard is time steps.

Figure 1 Figure 2

Figure 2, even though it occasionally does better than Figure 1, is not a healthy learning curve and most of the spikes are when randomness worked in the algorithms favor. As you can see, as quickly as it rises it falls.

A graph with lines on it

Description automatically generatedA graph of a graph

Description automatically generatedThe next piece that needed to be dealt with was the clip range. The default clip range for PPO is 0.2 which is way too low for this problem. To tell if the clip range is too low you can look at the number of times the gradient is clipped. This is found in the TensorBoard under train/clip\_fraction.

Figure 3 Figure 4

A graph with lines and numbers

Description automatically generatedA graph with different colored lines

Description automatically generatedFigure 3 is an example of a bunch of runs with various hyperparameters with a clip range of 0.2. As you can see in some cases most of the gradients are being clipped. Figure 4 contains more reasonable clip fractions at a clip range of 0.4. If there is no clipping at all then the clip range is probably too high. Another thing to look for to determine that the clip rate is too high is if there are extreme negative learning trends. Figure 5 depicts runs with a clip rate of 0.2.

Figure 5 Figure 6

A graph with lines and numbers

Description automatically generatedA graph with lines and numbers

Description automatically generated with medium confidenceProbably the most important and volatile hyperparameter to tune is the learning rate. The easiest way to tell that your learning rate is too high is if there is a slight negative learning trend. In Figure 6 the yellow and pink lines are runs with learning rates of 0.0001 while the blue and black lines are learning well with learning rates of 0.00001. Another thing to look at to help tune the learning rate are the policy gradient loss and the approximate KL divergence. Figure 7 depicts learning rates of 0.0001 (yellow), 0.00001 (blue) and 0.000005 (green). The green run is hardly learning anything because it is learning so slowly. The yellow run is learning too quickly and diverging, and the blue run is learning at a steady rate.

Figure 7 Figure 8

A graph with lines and numbers

Description automatically generatedA graph of a wave

Description automatically generatedSimilarly for KL divergence. KL divergence measures the amount that a policy changes from one step to the next. If the run starts off by bouncing up and down the learning rate is too high and it changes too rapidly to ever converge. No change signifies that the learning rate is too low and that no learning will occur. Figure 9 depicts a healthy mostly linear KL divergence trend. Even though it becomes more unstable toward the end the changes are tiny fractions compared to the changes in Figure 8.

Figure 9 Figure 10

Figure 10 depicts the policy gradient loss. This is probably the most significant loss because it can help you determine how stable your training is. Figure 10 is a normal trend because initially making any progress is easy but the further the algorithm gets into training the more difficult it is to learn the finer details of the environment. To help with training stability you can tweak the vf\_coefficient.

Results:

Figure 11 is the best PPO model I have trained so far and the parameters for it and the range around it that I checked are detailed in Table 1. All other hyperparameters were left at their defaults. I trained it for 1.8M steps but it peaked at 880,000 steps.

|  |  |  |  |
| --- | --- | --- | --- |
| PPO Algorithm Hyperparameters | More Than | Tested At | Less Than |
| Learning Rate | 0.000005 | 0.00001 | 0.0001 |
| Clip Range | 0.4 | 0.45 | 0.5 |
| Entropy Coefficient | 0.000001 | 0.00001 |  |
| Vf Coefficient |  | 0.5 | 0.7 |
| Gamma |  | 0.99 |  |
| Action Format |  | discrete |  |
| Environmental Hyperparameters |  |  |  |
| Exponentialize Reward |  | 1 | 2 |
| Reward for Speed |  | 0 |  |
| Reward for Displacement |  | 1 |  |
| Reward for Destination |  | 0 |  |
| Turn Punishment |  | 0.5 |  |
| Collision Course Punishment |  | 0.25 |  |
| LiDAR Resolution |  | 1 |  |
| Extra Observations |  | None |  |

A graph with a line

Description automatically generatedTable 1

Figure 11

Next Steps:

PPO has shown some good results but has its limitations. To continue working on PPO I would recommend working with the entropy coefficient. All my runs have peaked below 1M steps and that might indicate that the entropy coefficient is too low. I tested it at 0.00005 and it did not perform as well as 0.00001 but I omitted it from the table because it had an upward trend and training it longer might produce a better result.

I think that the best option would be to switch to another algorithm. There are a couple options, and I will start with the easiest to implement and proceed to the one I think will give the best results. SAC can be very easily implemented with my code. SAC is arguably better because it rewards for picking the best most random option. This automatically incentivizes exploration and there is no need to tune the entropy coefficient. Possibly an even better option is to use an algorithm that is not found in Stable-baselines3. An easy upgrade would be to [SBX](https://github.com/araffin/sbx) with an algorithm like DroQ. Pip ‘sbx-rl’ and add those algorithms to the code. There are a bunch of dependency issues that would need to be worked out but the general structure of SBX is the same as Stable-Baselines3. A harder option would be trying some algorithms that performed well on Atari games. Our problem seems pretty similar to some Atari games in that the algorithm takes in an image and controls the car like a video game. There are leaderboards for algorithms [here](https://paperswithcode.com/task/atari-games) and the best agents for games similar to ours are Agent57, GDI-H3, MuZero, R2D2, GDI-I3 and Ape-X. This would require an entire rewrite of the agent part of the code.

When a sufficiently accurate policy is found, I would recommend increasing the LiDAR resolution and maybe the LiDAR range as well and training another policy. Training this policy will take more computational time and memory but the policy will be able to get closer to objects without hitting them and will also be able to see further. I doubt that this policy will require any hyperparameter tuning over the original policy.

The final steps are getting this policy to run on a car in the Mini-City and then integrate it with other policies for lane and traffic pattern following.