ITI110 Deep Learning Project
Final Report

Team Name: Team 4

21A466A - Ong Joo Kiat Kenneth
21A471K - Shee Jing Le
21A530M - Pamela Sin Hui

Section 1: Vanilla Actor Critic

Introduction

Actor Critic is similar to a pair of adversarial networks in the sense that they both learn from each other. The Actor network, which is the policy network predicting the action taken given a state S, learns from the Critic network, which predicts (in this model) the total expected/potential returns given a state S. The Actor learns actions; improves probabilities of actions which gives a better reward than the one predicted by Critic and decreases probabilities of actions that don't. In turns, the Critic learns to predict a more accurate value from the actual returns given by the Actor.

There are various variations to the Actor Critic model, one of which is PPO, which will be explored by my teammate after this section.

Environment

The environment will be done in the simulator by Carla, with an edited gym-wrapper custom for Carla[2] for simplification and slight modifications purposes. The rewards function/table is shown in Table (1), and the inputs from Carla used, as well as the model used in the project is shown in Fig (1). With this set of rewards, the main objective of the reinforcement learning is to teach the agent to 1) keep to own lane. 2) obey traffic lights. 3) Avoid collision

Reward	Penalize	Termination
Simply moving	Travelling at speed below desired speed	Collision
Steering	Travelling at speed above desired speed	Out of lane
		Failure to stop at red traffic lights

Table (1). A general overview of the actions being rewarded or penalized. Actual factors are changed throughout the experiments

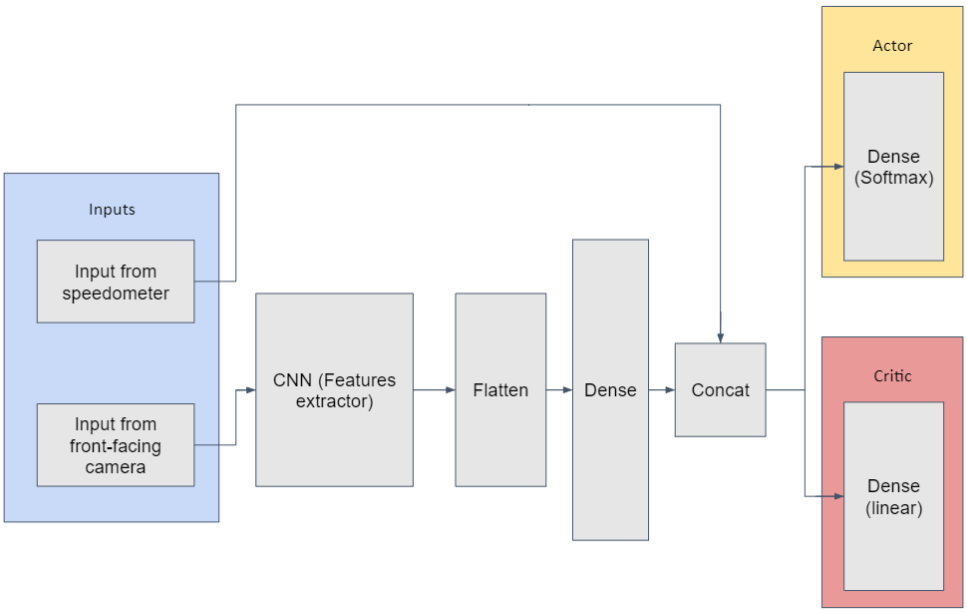


Figure (1). Overview of inputs, architecture and outputs used in the experiments

Experiments

A total of 25462 episodes, 57965 sub-episodes, roughly 6 million timesteps/frames were run for experiment 3, run2

	CNN	TD function	Exploration vs exploitation	Reward function	Optimizer	Other parameters	Observations
Experiment 1							
Run 1	Reduced input image size, reduced CNN size[10]	Gamma = 0.99, TD returns are normalized.	Exploration based on action probabilities	Default reward function based on gym-carla library[2]	Adam, lr = 0.01	Number of frames 1000	Action probabilities collapsed to single action, and loss function becomes NaN
Run 2						Added e, epsilon to action probabilities	Converged quickly to do only 1 action
Experiment 2							
Run 1	Changed CNN to 3x 64 [4] Additional observation, speed of vehicle is added as input	Gamma = 0.95[3], and partial-episode bootstrapping[5] to be used for non-time terminated episodes. To compensate for shorter sub-episodes,	Exploration based on action probabilities, with an additional probability of doing a random action	Rewards are rescaled so that the rewards will be in the range of +/- 1. To avoid premature convergence of action probabilities -> 0 or 1, do not update the policy if the actions probabilities are very small.	Adam, lr = 0.01. Due to the adversarial nature of actor-critic, do not update the policy when the critic loss is high[6]	Total number of frames of 600 to be split into 4 x 150 frames sub-episodes due to memory limitations	Situation observed where the agent converged to just speeding straight ahead.
Run 2				Rewards function changed			NaN values were still observed
Experiment 3							
Run 1	Back to original CNN used in Experiment 1	Gamma = 0.99[3], and partial-episode bootstrapping[5] to be used for non-time terminated episodes. To compensate for shorter sub-episodes,	Exploration based on action probabilities, with an additional probability of doing a random action	Rewards are rescaled so that the rewards will be in the range of +/- 1. To avoid premature convergence of action probabilities -> 0 or 1, do not update the policy if the actions probabilities are very small. Rewards for steering added	SGD, lr = 0.01 clipnorm = 1.. With SWA[8]. Due to the adversarial nature of actor-critic, do not update the policy when the critic loss is high[6]	Total number of frames of 600 to be split into 3 x 200 frames sub-episodes due to memory limitations	Critic loss function is increasing.
Run 2				Ratio of actor loss and critic loss changed by changing scale of rewards			Probabilities consistently in not too small a range, but agent not behaving ideally

	Problem observed	Reasoning	Actions taken
Experiment 1			
Run 1	Action probabilities collapsed to single action, and loss function becomes NaN	Loss function is log(prob), therefore if any action probabilities reaches a very small number, log(prob) will reach NaN (larger than float32 is able to handle)	Added e, a very small number to the action probabilities to prevent this from happening
Run 2	Converged quickly to do only 1 action	When the policy converges to only a single action, there is no longer any (or very very small) probability for any exploration. The critic function will converge to this “non-optimal” value, and without exploration, the actor and critic will agree with each other and no further learning will be done.	Added an exploration term to the policy during training

Experiment 2			
Run 1	Situation observed where the agent converged to just speeding straight ahead.	After analysis of the reward function, I found out that above a certain speed, the reward given by the speed outweighs the penalty for speeding. Therefore, a local maximal (of rewards) is formed where the agent will speed up until it collides.	To change the penalty of speeding to properly penalize speeding actions, as well as added a steering reward to incentivise exploration of steering (to facilitate lane following)
Run 2	NaN values were still observed	Diagnosis of the CNN learnt, the 1st conv layer's output are all zero even with a small number of episodes. This might be a case of vanishing gradients where the network is dead. Might be due to huge losses and gradients during initial episodes[7]	To introduce gradient clipping to our optimizers and change the activations from 'relu' to 'elu'. Optimizer changed to SGD, with SWA[8] added
Experiment 3			
Run 1	Critic loss is increasing	Monitoring of the loss functions showed that even though critic loss is increasing, the total loss is still decreasing. This is due to the decrease in the actor loss is greater than the increase in critic loss, causing the model to diverge. This is most likely due to the -logP calculations in the actor loss which can lead to large negative number	Adjust the scale of the total rewards so that the total loss is balanced
Run 2	Rewards/distance traveled did not show signs of improving.	Monitoring of action probabilities/loss functions shows that the model is not behaving like previously anymore; there are no NaN/inf values, outputs = 0 etc. But it also does not seem to be converging. This could be due to the high learning rate of the SGD (in the original paper for SWA, SGD with high learning rate or cyclical learning is recommended, but in the paper using SWA with reinforcement learning is not mentioned)	Sadly but end of project

Results

Rewards for both the SGD as well as the SWA weights are noisy but consistently not increasing or decreasing. Would be quite meaningless to show a graph which says nothing.

Discussions

Comparing experimental results with Jing Le, PPO, he encountered similar issues and also achieved similar results, which could be considered as expected as we are using similar environments with similar rewards functions. However, when comparing experimental results with Pamela, DQN, she has achieved better perceived results. However, since she is using a totally different environment, with a very different reward function, which only penalizes collision without any lane keeping functions, this could be the main difference as the model will be looking for totally different features.

Section 2: Proximal Policy Optimization

Introduction

An On Policy model - Instead of having a fixed dataset, Reinforcement Learning does not have a fixed dataset. Instead, its dataset is generated as the model makes a decision based on its policy. Therefore, policy finetuning is crucial in generating data it is done by rewards design.

The Proximal Policy Optimization (PPO) algorithm combines ideas from Policy Gradient Methods and TRPO. The main goals of PPO is to prevent destructive policy changes and optimising updates. It does do by clipping the allowable policy changes and optimize changes with minibatch SGD after n epochs respectively. It also provides balance easy coding, sample efficiency and easy to tune. Data input wise, it does not relearn from the same sample - every sample is discarded after its interaction with the environment. For this project, we will be using Stable baselines3 library. Stable Baselines uses a basic 3 layers sequential CNN to process the input image.

Experiment 1

Run 1, the testing run, the model has the highest rewards 103 with at 34816 timesteps, at the rollout stages, which, I forced the model to stop training once the model starts to decline.

Run 2, I added Callback function to evaluate and save the best model to do implementation if possible. Looking at how the rewards is designed, the policy is designed to avoid collusion. Therefore, speed plays a big part of data generation to the observation in the environment. Reduce the total steps to 50k from 100k since the first peaked at 43k. This gives a periodical evaluation of the model and we successfully exported a model out with mean reward of 414 and episode length of 154.

Run 3, we activated a core function of PPO – value clipping. PPO prevents the model from making drastic changes to its policy using a clip range. This clip range (clip range value is 1) specific by how much the estimated advantage can move. Next, we reduce the epochs from 10 to 5 so the model can train faster given the limited resources. Results are better with a mean reward of 582.94 at with 161 episode length. This requires 44k steps of training.

Run 4, we increased the steps to 100k because 44k is too near 50k and there could still be improvements. We also set the clip range value at 0.2 to prevent overcompensating for bad results. The results kept on fluctuating but did not improve. We

suspect this is due to the model stuck at the local minimum. Thus, the model is stop prematurely.

Run 5, we increase the clip range value back to 1 but the model rewards and did not bother to increase. We think that we are stuck with an exploration problem because the model is stuck at a local minima with rewards not more than 300.

Run 6 and the run 7, we set a value for the entropy coefficient to force the model out of the local minima. Entropy helps to widen the loss function and encourages the model to explore (take risk). But both does not show significant improvement. The limitation factor, looking at previous results, could be the clipping of the value function. The intention of the clipping function is to prevent destructive policy changes but in this case, it is limiting policy changes. The other change was setting normalisation to false. What we were trying to test is if the images is not normalised, can the model identify if the object is further/closer to the vehicle.

Run 8 and 9, we clip only the action function and will revert the clipping of value function to 1 but results are still stagnant.

		Run2	Run3	Run3	Run4	Run5	Run6	Run7	Run8	Run9
Parameters										
	clip_range	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2
	clip_range_vf	None	1	1	0.2	1	0.2	0.2	1	NA
	ent_coef	0	0	0	0	0	1	0.5	0.5	0.5
	batch_size	64	64	64	64	64	64	64	64	64
	deterministic	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
	Normalisation	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE	FALSE	FALSE
Best Results										
	total_timesteps	22000	24000(24	44000	15000	81000	16000	51000	13000	9000
	mean_ep_length	154	123	161	83.4	69.4	85.8	71.2	67.2	149
	mean_reward	414	390.28	582.94	288	247	149.18	113.24	127.97	144.03

Significant parameters and results of first 9 runs

Experiment 2

Starting from run 10, we finally fix the Cuda driver enabling GPU, and the pygame image on the system – this allows the model and ourselves to observe more of the surroundings. We switched up the parameters significantly because the current hyper parameters are not performing as well as intended. But the learning rate is too slow – the car is moving too slowly.

Run 11, we increased the learning rate by 10x, from 0.0003 to 0.003. (On hindsight, this might not have been a good change because default parameters are tested rigidly and what was needed was more patience.) At the start of the training, the car does not move. We can see clearly that the rewards do get better but the thing is, the car terminates when it goes out of lane and the car for some reason does not turn back into lane and the episode ended. This could be caused by a negative steering rewards in the rewards function discouraging the model from steering into the right path.

Run 12 and 13 tested the collusion parameter of the rewards function. By changing the reward function, we hope to get better data and thus better results. Ignoring the rewards, what is important is the length the car can stay on the road. Another change that was made was the termination condition. To test if the car cannot turn back, we allowed the car to continue moving and true enough it didn't turn back.

We further tested the rewards function with Run 14, 15 and 16. Run 14 was showing good prospects but the lightning strike and the process was stopped. Looking at the video, the car is also moving slowly. Run 15 and 16 is very interesting because the car simply don't move after a period of training. The underlying reason could be firstly, the policy changes is not enough for it to gain a higher reward. Second, the magnitude of relative speed to desired speed is so high that it does not move. What we can conclude is the rewards of time and time+1 must have significant enough difference for the model to decide on an action to make a change. The solution could be to add a entropy coefficient to force a bigger random change or the increase the clipping.

Run 17, the model kept moving to the left because there are no termination restraints on the path and there are lesser cars. Thus, it has a good result (because its moving) but these results are meaningless because it needs to be on the path. Another observation is the car objective is to stay on the road and not to go to the objective – it goes straight on a path its supposed to turn.

Run 18, the initial final run, we are going to try to run at 20,000 max steps, the objective is to make sure the model has enough time to reach the destination. And, it should also do so in the right path, so out of path is activated so that the episode terminates and the policy is updated. Also, like observed previously, there is a tendency to not steer because of a penalty on steering, so the penalty on steering is reduced by 5x and the penalty of gg out of lane is increased by 10x. This will ensure the model rather turns back than go out of lane. Also speed reward is doubled because the car moves too slowly. However, this was the wrong assumption. I ran the model overnight, 1 episode took more than 4 hours and the vehicle didn't move an inch.

Run 19 and 20 is a range of testing of a mixture of reward changes to be implemented to the model. In general, the problems that still exist is the failure of the model to stay in the lane, very slow start, often not moving for the first few episodes of 1000 timesteps, not being able to turn. Also, we discussed the results and I decided how about training on an empty road? Maybe it will help. Not all tests were recorded, some samples below:

```
# r = 200*r_collision + 1*lspeed_lon + 10*r_fast + 1*r_out + 0.2*r_lat + 1*speed - 0.1
# r = 200*r_collision + 10*r_fast + 1*r_out + r_steer*1 + 0.2*r_lat + 1*r_speed - 0.1 #run15
# r = 200*r_collision + 1*lspeed_lon + 1*r_out + r_steer*5 + 10*r_speed + 1000*r_des - 0.1 #run16
# r = 200*r_collision + 1*lspeed_lon + 10*r_fast + 1*r_out + r_steer*5 + 0.2*r_lat - 0.1
# r = 200*r_collision + 1*lspeed_lon + 10*r_fast + 1*r_out + r_steer*5 + 0.2*r_lat - 0.1 #run17
# r = 200*r_collision + 2*lspeed_lon + 10*r_fast + 10*r_out + 0.2*r_lat + r_steer*1 - 0.1 #run18/19
r = 200*r_collision + 2*lspeed_lon + 10*r_fast + 1*r_out + r_steer*4 + 0.2*r_lat - 0.1 #run20
```

Run 21, we wanted the car to look further and train faster. So we increased the distance visible (and by doing so increased the input frame) and reduced the timestep for each episode by half. As discussed in the above 20++++ runs and failed experiments. This run is significantly better. As the model trains, there are attempts to correct itself back unto the lane. This shows that it is picking up road lines on the floor. The other proof that it is picking up road signal is it doesnt know how to turn yet because at road junctions, there are no lines. I am sure with more training, the model will learn what other signals to look for. And more importantly, we can move in this direction.

Deployment

As of now, the model is not ready to be deployed in a real vehicle. This is a significant risk and lives will be lost. However, the model can be saved and deploy in Carla.Firstly, the best model is saved during run21 at around 18000 timesteps because of the code to save the best model. Secondly, we trained until 51000 timesteps but this model wasnt auto save (due to coding).. We loaded it back into Carla, the same environment and ran the test. U can see that the model at 18000 timesteps is still unstable altho it has a high reward value in video (*pygame window 2022-03-02 09-58-19 Deployment.mp4*). However, if you look at the training video, the vehicle has the ability to navigate as shown in video (*pygame window 2022-03-02 01-09-10 Run21 Final Video 3.mp4*). This further reinforce our understanding that the rewards does not matter as much as the length of the training. As more training is done, the model has the ability to self adjust and self correct. The reward function serves as a guide for the model but it is not for us to determine if the model is performing.

Summary

1. The different factors in the rewards function should not overpower each other. That means the weight of a factor cannot be too high because the model will focus only on one weight. In a lot of trials, the car just don't move.
2. Patience. Usually the model takes awhile to train, the first few episodes of 1000 timesteps, the car will usually be slow or stagnant. Then it will slowly start to move. Thus, it takes a very long time to finetune the hyperparameters. Most of the training was forced to stop because there are no improvements in rewards and timesteps. However, there could be more progress if we continue training like shown in Run 3. Or it can just be luck because the road is clear.
3. We need visuals. The first 9 runs were done without visuals which sadly, we can only rely on the figures to tell if the model is doing great or not. But given visuals from run 10 onwards, we can tell if the car decides to not move, or the car moves but because there is no obstacle in place, it has recorded a high reward and timeframe, but it is not meaningful. This also means the car needs good visuals in the form of inputs. By increasing the distance the car is analyzing, there seems to be better performance (Run21). And we can further test normalization - in theory, it can help us analyze the object no matter the distance.
4. Reward function, hyperparameters and available resources need to be balanced. Given enough time, there are a lot of things that we can try but it's not just about computational power. Time is needed to visually monitor the vehicle, the tunning of the rewards function and hyperparameters play a part as shown in runs 10-20.
5. Environment and resources. Unlike supervised learning, where there is a dataset, the RL model is thrown into the environment, and it learns itself. This needs the environment to be properly setup (original environment from gym carla had some bugs) and it requires a lot of resources to run multiple episodes repeatedly. This requires understanding of the environment, the models, the APIs, and even to extends of other programming languages like MS-doc. Furthermore, there can be many variations of the environment. In this case, they are the cars on the road, pedestrians, maps, weather etc.
6. Along the training process, things become clearer that there might be an issue with not knowing the model enough and not doing proper evaluation. It is a wide universe to comprehend and there are many things happening together. Examples are time needed to let the model run on its own, visually looking so we can better adjust the parameters etc. Our understanding also improve by learning from videos and examples from the internet. For example, one of the youtubers ran his model on 96GB VRAM for 5 days and his car still crashes every few seconds.

Future works

Mentioned in the summary, the topic is very wide and there are a lot other things that we can do. Examples are State Dependent Exploration (SDE) instead of action noise exploration, rewards function, model environments like cars, pedestrians, weather, map etc, hyperparameters like kl divergence, ADAM optimizer instead of SGD, custom convolution layers and many many more.

Section 3: Deep Q-Network (DQN)

In DQN, we use a neural network to approximate the Q-value function. The state is given as the input and the Q-value of all possible actions is generated as the output. The general steps involved in reinforcement learning using DQN is that all the past experience is stored in memory, subsequently the next action is determined by the maximum output of the Q-network, then update Q-values as derived from the Bellman equation and the network updates its gradient using backpropagation to reach convergence. Every step an agent takes comes not only with a plausible prediction (depending on the epsilon for that step which correlates to exploration), but also with a fitment. We are training and predicting at the same time, and we want our agent to get the most frames per second (FPS) as possible.

DQN achieves stable learning by mainly four techniques.The first technique is experience replay. DNS may easily overfit current episodes, and subsequently this may lead to difficulty in producing various experiences. To solve this, instead of running Q-learning on state-action pairs as they occur during the simulation, Experience Replay stores the agent's experiences including state transitions, rewards and actions, which are necessary data to perform Q learning, and makes mini-batches to update neural networks. For our experiments, we put the last 100,000 video frames into a video buffer and randomly sample a mini-batch of samples from this to train the deep network, making the data more independent of each other. The benefits of this is that it reduces correlation between experiences in updating DNN, increases learning speed with mini-batches, and reuses past transitions to avoid catastrophic forgetting.

The second technique is the target network technique. In TD error calculation, the target function is continuously changing with each iteration in DNN. We depend on the policy or value functions to sample actions, but this is frequently changing as we continuously learn what to explore, and as we get to know more about the ground truth values of states and actions the output is changing. However, an unstable target function makes training difficult. Thus in DQN, two deep networks are created where one retrieves Q values and the other includes all updates in the training, and after C iterations the first deep network is synchronized with the second. As the target network has the same architecture as the function approximator but with frozen parameters, the Target Network technique fixes parameters of target function and replaces them with the latest network (parameters from the prediction network are copied to the target network) every C iterations, leading to more stable training as the target function is fixed (for C iterations).

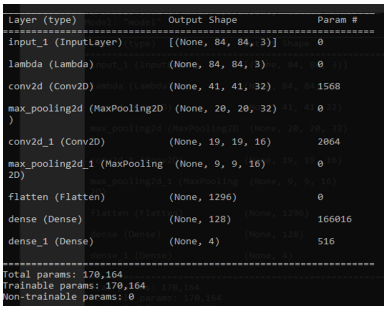
The third technique is clipping rewards, where all positive rewards are set +1 and all negative rewards are set -1. The fourth technique is skipping frames where the DQN calculates Q values every n frames and uses the past n frames as inputs, reducing computational costs and gathering more experience.

The step method takes an action, and then returns the observation, reward, done, any_extra_info as per the usual reinforcement learning paradigm. Action 0, 1, 2 and 3 refers to breaking, driving straight, steering left and steering right respectively. The 4-neuron output, that is each possible action for the agent to take, is added. To begin, we only train if we have a bare minimum of samples in replay memory. If there is a bare minimum number of samples in replay memory, we grab a random minibatch, and from there grab our current and future Q-values.

For the first experimental run, we observed that using arbitrary rewards such as -50 for a collision and +1 for each frame the car was driving sent the Q-values and loss exploding. Rewards kept between -1 and +1 for each frame helps curtail exploding Q-values and losses. As such, the reward for subsequent runs was set as follows:

Action	Detector	Detection	Reward
Stop	Front side proximity detector	Detects an object	+0.1
		Does not detect an object	-0.1
Move forward	Front side proximity detector	Detects an object	-0.7
		Does not detect an object	+0.7
Move left	Left side proximity detector	If previous action was to move right	0
		Detects an object	-0.22
		Does not detect an object	+0.22
Move right	Right side proximity detector	If previous action was to move left	0
		Detects an object	-0.22
		Does not detect an object	+0.22

For the second experimental run, we observed that simplifying the CNN, reducing the number of trainable parameters, gave improved results. For a typical supervised learning framework, more parameters work well as the neural network makes comparisons to the actual ground-truth that remains constant throughout. For reinforcement learning however, it starts with random actions to interact with the environment, and using a neural network with millions of weights to calculate the loss against changing ground-truths may easily drive it to a local-minima. For reinforcement learning, we are fitting a model and also fitting for Q-values. Thus, we opt for a less complex neural network for subsequent runs as follows:

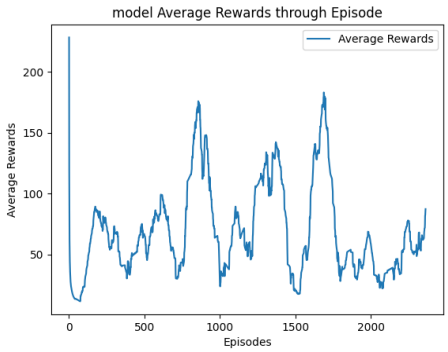


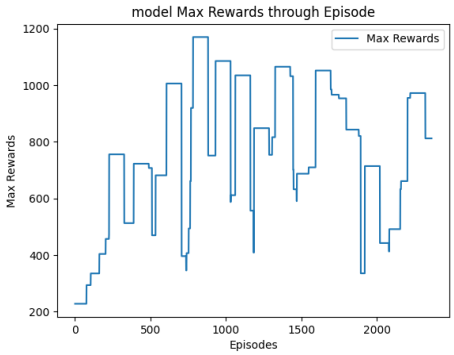
Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 84, 84, 3)]	0
lambda (Lambda)	(None, 84, 84, 3)	0
conv2d (Conv2D)	(None, 41, 41, 32)	1568
max_pooling2d (MaxPooling2D)	(None, 20, 20, 32)	0
conv2d_1 (Conv2D)	(None, 19, 19, 16)	2064
max_pooling2d_1 (MaxPooling2D)	(None, 9, 9, 16)	0
flatten (Flatten)	(None, 1296)	0
dense (Dense)	(None, 128)	166016
dense_1 (Dense)	(None, 4)	516

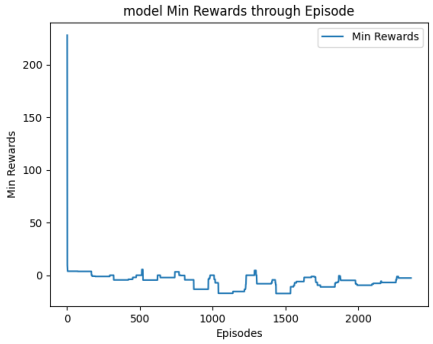
Total params: 170,164
Trainable params: 170,164
Non-trainable params: 0

For the third experimental run, we found that when epsilon was set to 0, agents tend to perform only one action, such as the agent driving in a circle. Thus, we find that a high epsilon close to 1 at the beginning of training forces the car agent to perform random actions in an attempt to explore its environment, while a decent rate of decay of epsilon gives more “weight” to the learned Q-values as the number of steps increases and the agent has already performed a certain degree of exploration of the environment. When we start off, we have a high epsilon which translates to a high probability of choosing an action randomly, rather than predicting with the neural network. We have a main network which is constantly evolving, and the target network which we update every n steps.

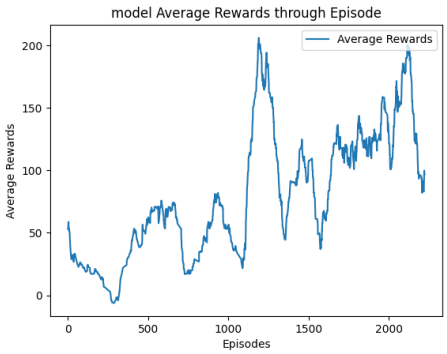
The fourth experimental run incorporated the learnings from the learnings from the previous three runs, and was run for approximately 24 hours and the following graph results are obtained. Note that the graphs of average, max and min rewards through episode are on a rolling 100-episode basis. The rolling 100-episode average reward through episode has numerous peaks and falls back to previous minimums throughout the run. Similarly in the graph of rolling 100-episode maximum rewards through episode, there does not appear to be a general upward or downward trend, but rather there are peaks that reach similar highs and falls that reach similar lows as previous episodes. For the graph of rolling 100-episode minimum rewards through episode, generally these minimum rewards are negative and do not seem to change trend-wise, which may indicate episodes where the car agent was dropped into an unfortunate setting. Comparing the first two graphs, it appears that some episodes have performed exceptionally well, pushing the rolling 100-episode maximum rewards to be between approximately 400 to 1,200 for some intervals, versus the rolling 100-episode average rewards of between approximately 25 to 170 for majority of the intervals. There might have been cases where the car agent is dropped into a fortunate setting with wide lanes such that no matter what action the car agent takes, there is no collision for a number of episodes. The car was observed in the simulation to make a number of interesting turns on the road, which provided some degree of confidence.

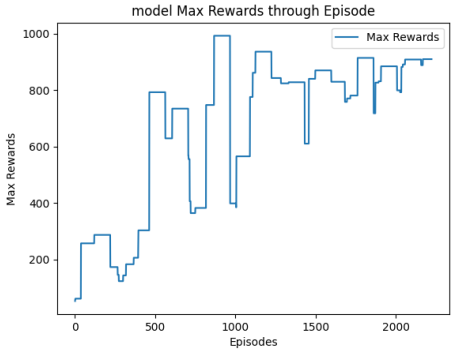


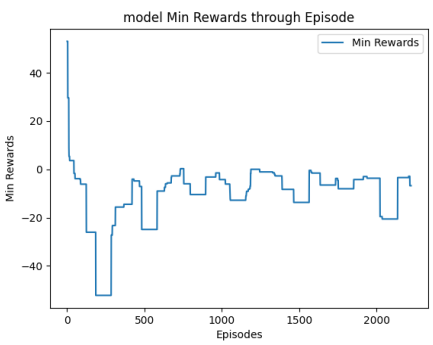


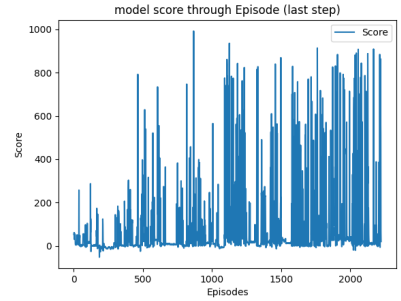


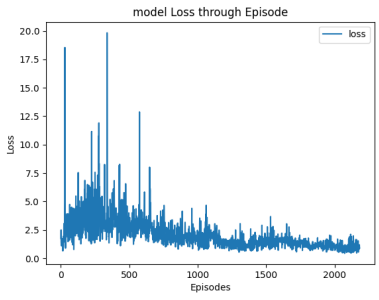
In the fifth experimental run, the minimum epsilon hyperparameter was increased from 0.1 to 0.2, and was run for approximately 24 hours. We observed that the fifth run performed better than the fourth run, with the rolling 100-episode average reward through episode showing a general upward trend and reaching above 200 at two peaks. Similarly the graph of rolling 100-episode maximum rewards through episode shows an increasing trend where the maximum rewards for later episodes are generally higher. The graph of rolling 100-episode minimum rewards through episodes fluctuates around the same negative reward levels throughout the episodes. A possible reason for improved performance in this run is that with the probability that a random action is selected being doubled, this encourages more exploration for episodes at the tail-end of the run and this exploration remains an important factor throughout the entire reinforcement learning process especially when the environment is large and there are numerous actors involved.



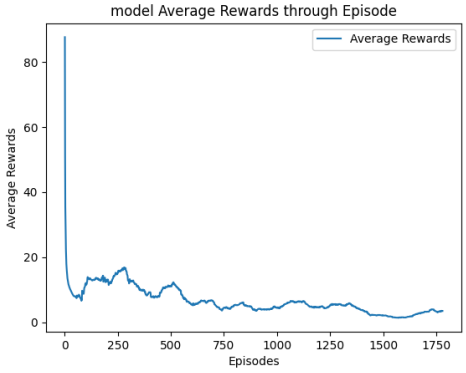


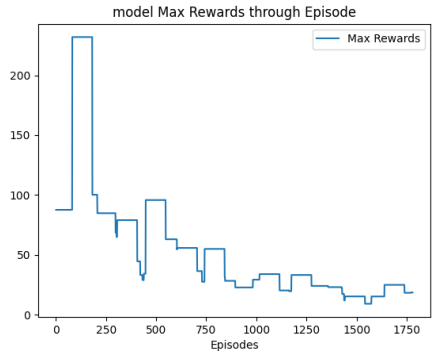


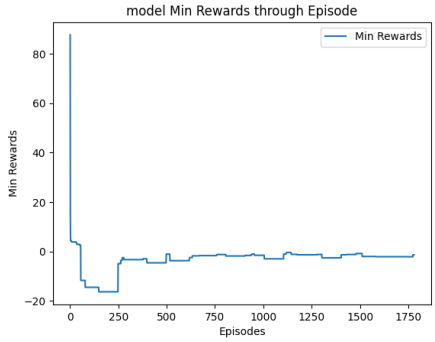


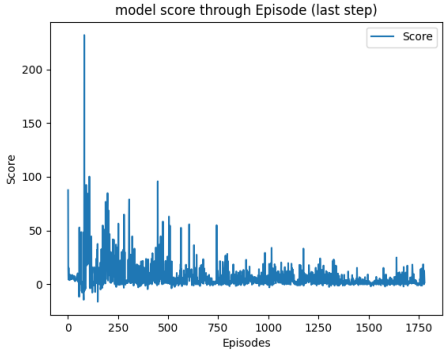


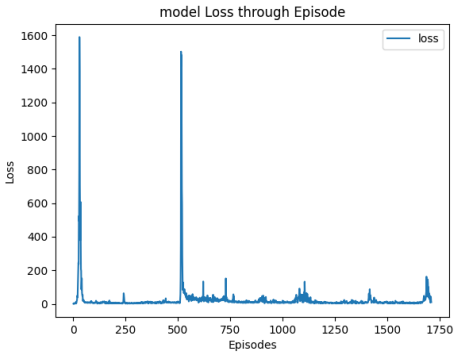
In the sixth experimental run, the image dimension hyperparameter was increased from 84 X 84 to 168 X 168, and was run for approximately 12 hours. We observed that this run performed worse than the fourth and fifth runs. The graphs of rolling 100-episode average and maximum rewards declined consistently. The graph of score through episode also showed a downward trend. For the DQN, we want decent accuracy, however taking note that at the beginning the car actor is taking random actions for exploration and slowing learning and updating the Q-values and thus a high accuracy at this stage might be a concern. And similar to the rationale in the second experimental run, fewer trainable parameters is favored in the context of calculating loss against changing ground-truths in DQN.

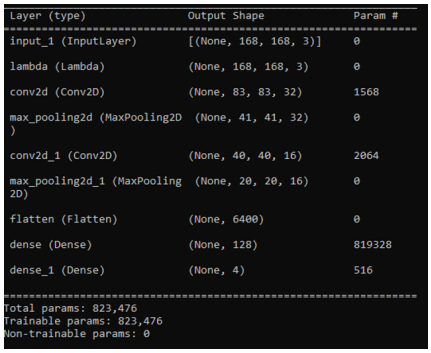












Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 168, 168, 3)]	0
lambda (Lambda)	(None, 168, 168, 3)	0
conv2d (Conv2D)	(None, 83, 83, 32)	1568
max_pooling2d (MaxPooling2D)	(None, 41, 41, 32)	0
conv2d_1 (Conv2D)	(None, 40, 40, 16)	2064
max_pooling2d_1 (MaxPooling2D)	(None, 20, 20, 16)	0
flatten (Flatten)	(None, 6400)	0
dense (Dense)	(None, 128)	819328
dense_1 (Dense)	(None, 4)	516

Total params: 823,476
Trainable params: 823,476
Non-trainable params: 0

Summary of results

	Experiment 1	Experiement 2	Experiment 3	Experiment 4 (~24 hours)	Experiment 5 (~24 hours)	Experiment 6 (~12 hours)
Reward function	Collision: -50 Driving: +1	Between -1 and +1				
CNN complexity	(conv2d + max_pooling)*2, dense(128), dense(4)	(conv2d + max_pooling)*2, dense(128)*2, dense(4)	(conv2d + max_pooling)*2, dense(128), dense(4)			
Epsilon	1 (rate of decay at 0.999985)		0	1 (rate of decay at 0.999985)		
Epsilon minimum	0.1		0	0.1	0.2	
Image dimensions - CNN	84 X 84					168 X 168
Episodes (Steps)				2,610 (698,344)	2,543 (721,357)	1,820 (95,135)
Reward				See graphs		
Observation and learning points	Arbitrary large rewards caused Q-values and loss explosion. For future runs, keep rewards around -1 and +1 for each frame.	Using more complex CNN caused consistently low rewards. For future runs, use simpler CNN with fewer learnable parameteres.	When epsilon set to 0, car agent tend to perform only one action. For future runs, use high epsilon at the beginning for exploration.	Run using best hyperparameters from experiments 1 to 3.	Exploration for episodes at the tail-end of the run is encouraged, an important factor throughout the entire reinforcement learning process when the environment is large and numerous actors involved.	CNN with fewer trainable parameters favoured in the context of calculating loss against changing ground-truths in DQN.

Note: Highlighted in green are the best hyperparameters

In terms of future work, a possible avenue is to spawn more complex environments such as a high-traffic city, having the car agent detect and navigate through multiple moving objects. In addition, since there are multiple city environments to choose from, it is possible to choose a random city every n episodes. This could help the DQN not to generalize to a single city environment. Another avenue to explore is incorporating more complex reward functions that add more elements to simulate real driving expectations, such as a higher reward for driving above a certain speed, which is expected to have the effect of car agents maintaining minimum speeds to quickly reach destinations versus simply avoiding crashes.

Group member contribution

Member	Section
21A466A - Ong Joo Kiat Kenneth	Actor Critic
21A471K - Shee Jing Le	Proximal Policy Optimization
21A530M - Pamela Sin Hui	Deep Q-Network

Actor Critic

[1]A. Nandan, “Implement Actor Critic Method in CartPole Environment,” Keras, 13-May-2020. Available: https://keras.io/examples/rl/actor_critic_cartpole/.
[2]J. Chen.”gym-carla”, Github repository, https://github.com/cjy1992/gym-carla
[3]J. L. McClelland, “Temporal-Difference Learning,” in Explorations in Parallel Distributed Processing: A Handbook of Models, Programs, and Exercises. 2nd ed. draft, Stanford University, 2015, ch.9.
[4]Longer-term model results - Self-driving cars with Carla and Python,” Python programming tutorials, https://pythonprogramming.net/trained-model-self-driving-autonomous-cars-carla-python/.
[5]F. Pardo, A. Tavakoli, V. Levdik, P. Kormushev, “ Time Limits in Reinforcement Learning”, arXiv:1712.00378 (2018).
[6]S. Parisi, V. Tangkaratt, J. Peters, M. E. Khan, “TD-Regularized Actor-Critic Methods”, Mach Learn 108, 1467–1501 (2019).
[7]R. Pascanu, T. Mikolov, Y. Bengio, “ Understanding the exploding gradient problem”, arXiv:1211.5063v1 (2012).
[8]E. Nikishin, P. Izmailov, B. Athiwaratkun, D. Podoprikhin, T. Garipov, P.I Shvechikov, D. Vetrov, A. G. Wilson, “Improving Stability in Deep Reinforcement Learning with Weight Averaging”, (2018)
[9]S. Bach, A. Binder, G. Montavon, F. Klauschen, K. Müller, W. Samek, “On Pixel-Wise Explanations for Non-Linear Classifier Decisions by Layer-Wise Relevance Propagation”, PLOS ONE 10(7): e0130140, (2015)
[10]Stable Baselines3, https://stable-baselines3.readthedocs.io/en/master/guide/custom_policy.html#off-policy-algorithms

PPO

[1] Stable_baselines3.Ppo.Ppo — Stable Baselines3 0.6.0a6 Documentation. https://stable-baselines3.readthedocs.io/en/latest/_modules/stable_baselines3/ppo/ppo.html. Accessed 22 Feb. 2022.
[2] Stable_baselines3.Common.Callbacks — Stable Baselines3 0.6.0a6 Documentation. https://stable-baselines3.readthedocs.io/en/latest/_modules/stable_baselines3/common/callbacks.html. Accessed 22 Feb. 2022.
[3] Foster, Chris. Entropy Loss for Reinforcement Learning. https://fosterelli.co/entropy-loss-for-reinforcement-learning. Accessed 22 Feb. 2022.
[4] https://github.com/cjy1992/gym-carla
[5] https://arxiv.org/pdf/1707.06347.pdf
[6] https://upcommons.upc.edu/bitstream/handle/2117/355912/tfm-h-ctorizquierdo.pdf?sequence=1&isAllowed=y
[7] Proximal policy optimization¶. Proximal Policy Optimization - Spinning Up documentation. (n.d.). Retrieved February 23, 2022, from https://spinningup.openai.com/en/latest/algorithms/ppo.html
[8] Carla.readthedocs.io. 2022. Python API reference - CARLA Simulator. [online] Available at: <https://carla.readthedocs.io/en/latest/python_api/#instance-variables_31> [Accessed 2 March 2022].
[9] Stable-baselines3.readthedocs.io. 2022. Custom Policy Network — Stable Baselines3 1.4.1a3 documentation. [online] Available at: <https://stable-baselines3.readthedocs.io/en/master/guide/custom_policy.html#multiple-inputs-and-dictionary-observations> [Accessed 2 March 2022].
[10] Medium. 2022. Understanding and Implementing Proximal Policy Optimization (Schulman et al., 2017). [online] Available at: <https://towardsdatascience.com/understanding-and-implementing-proximal-policy-optimization-schulman-et-al-2017-9523078521ce> [Accessed 2 March 2022].
[11] Spinningup.openai.com. 2022. Proximal Policy Optimization — Spinning Up documentation. [online] Available at: <https://spinningup.openai.com/en/latest/algorithms/ppo.html> [Accessed 2 March 2022].
[12] An introduction to Policy Gradient methods - Deep Reinforcement Learning

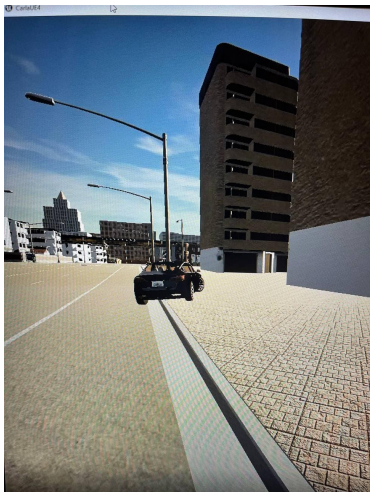
Deep Q-Network

[1]https://github.com/EmanueleBenedettini/DQN-Autonomous-Vehicle-on-CARLA
[2]https://towardsdatascience.com/welcome-to-deep-reinforcement-learning-part-1-dqn-c3cab4d41b6b
[3]https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/
[4]https://pythonprogramming.net/trained-model-self-driving-autonomous-cars-carla-python/
[5]https://www.tensorflow.org/agents/tutorials/0_intro_rl
[6]https://towardsdatascience.com/deep-q-learning-tutorial-mindqn-2a4c855abffc
[7]https://towardsdatascience.com/dqn-part-1-vanilla-deep-q-networks-6eb4a00febfb
[8]https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html
[9]https://jonathan-hui.medium.com/rl-dqn-deep-q-network-e207751f7ae4

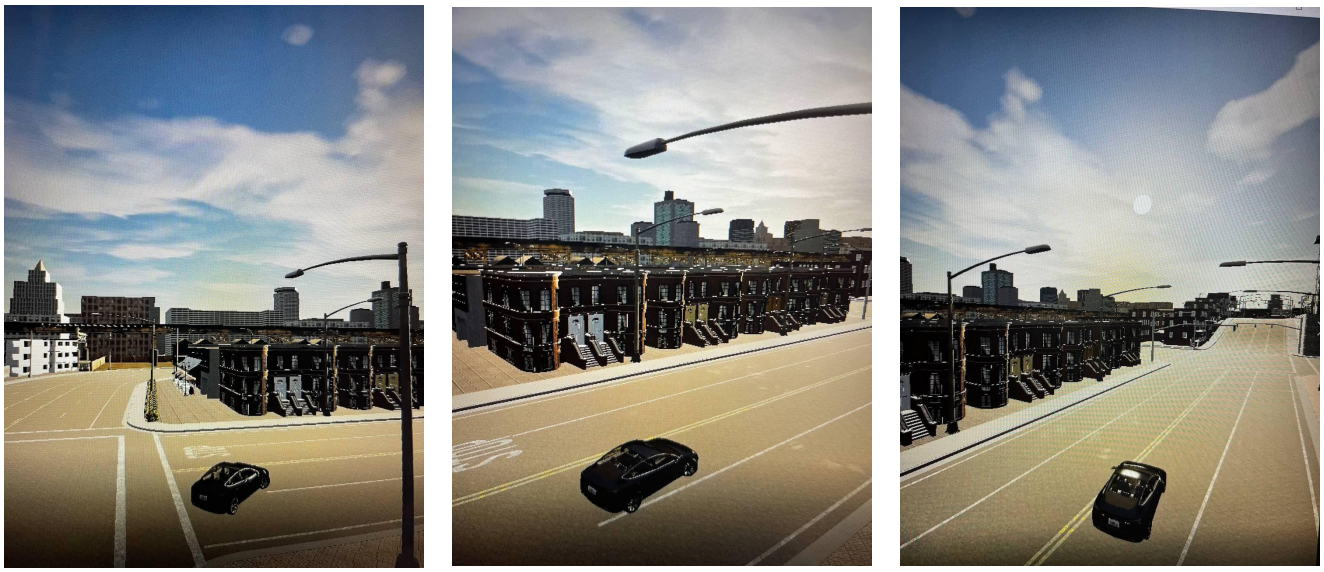
Appendix

DQN demo
https://youtu.be/G1Y9H1iqY8c
https://youtu.be/tDV37bbLN9w
https://youtu.be/L5CRrl758Yc

(a) DQN setup - An instance where the car agent went into the curb, resulting in it spawning for a fresh episode shortly after



(b) DQN setup - An instance of the car agent handling a turn on the road



(c) DQN setup - An instance of the car agent driving in circles