Autonomous Driving Simulation

Due date: Dec 7th, 2017
Group member: Zezhou Sun, Siqi Zhang

Project Overview

Developing a robust autonomous driving system for automobile is always a popular topic in the field of computer vision and artificial intelligence. In this project, our team will take on the task of creating an autonomous driving system for a car racing simulation game with the help of feature extraction algorithms as well as the deep learning models.
Figure at right shows one frame sample of the game.

Figure 1. Game UI

Problem Definitions

- Develop an agent that can accept only game image(s) and current score as input and should return the actions (keypress: UP, DOWN, LEFT, RIGHT, N) to take under current condition.
- Goal: Gain score as high as possible. Score can also reflect game completion. To get high score, agent must learn to finish the game as much as possible and avoid crashing. Agent should reach the destination as fast as possible, and time limitation for this is 70 seconds.

Problem Analysis

1. Agent can only accept image frames captured from the game and current score, no other information can be used to make decision
2. To get higher score, according to the instruction of the game, we need to make sure car stay on road as much as possible and avoid crashing with any obstacles that could exist on road.
    - To avoid crashing, agent need to find out if there exist obstacles before the car, and make decision to avoid hit with it.
    - To keep the car in road, agent should have the ability to find out where the road is, and make decisions to keep car on road.
3. In original DQN for ATARI, they use the difference between two frames (motion energy) to track the object. But in this game, there are more environment noise (environment lighting changes, fog on road, background changes) in this game. Unlike those 2D ATARI game, the perspective view in this game make motion energy no longer enough for DQN to learn features on images. So pre-processing and feature extraction is very important in this condition.
   It is necessary to preprocess the images before feeding them into the model due to several of noises. Figure 2 shows that the brightness of the road can vary because of the illumination from the headlights and streetlights. And Figure 3 is the case that the brightness can also be affected by random fog on the road surface.

Figure 2. Multiple illumination sources

Figure 3. Fog

Besides the brightness invariance, the background could be problematic. Figure 4 demonstrates noise brought by background. It will be helpful for DQN model training If we can subtract the background because it will reduce number of states for analysis. But the perspective view is always changing, which makes it inappropriate to use a certain line to cut image to get road from frame image. Figure 5 and 6 shows the case where the horizontal lines of road are quite different on every frame. So we need a well designed algorithm to cut road out of frame.

Figure 4. Background noise

Figure 5. Driving uphill

Figure 6. Driving downhill

System Architecture

There are two major parts in our system: Universe[1] Environment and Agent Training Model.

- Universe[1] Environment: Universe will use Docker container which can be ran on remote system to create an simulation environment for Dusk Drive[2]. Our program will run as a client to communicate with this remote component via network to get game frames and score, and these frames will also sent to screen for display and human observation purpose. Figure 7 shows a sample of the display result. Once our agent make decision about what actions to take, client component will send actions to remote component so actions can take effect in the game. With this virtual environment, we can easily access the display stream of the game being play within the environment. On the other hand, we can also interact with the Universe environment by defining the input actions from keyboards and mouse.
- Agent Training Model: Model receive observations (Game frames) and rewards (score). Model first apply some pre-processing on observations, then record these features as next_state. Then push them to Neural Network for analysis. Neural network will output score for each action. Then we can based on some rules (e.t. max score) to take actions. Also model need to record all events happened, so they can be used to train DQN.

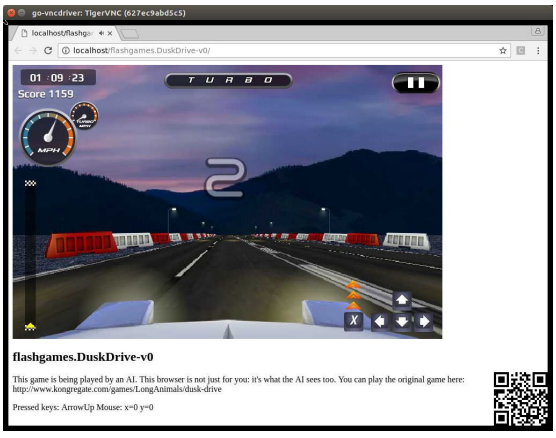

Figure 7. Universe environment

Then we initialize the parameters for learning model and start the iterative process. The learning process can be described by the following pseudo code and diagram.

Environment and variable Initialization
While episode_n < max_episode
    state = None
    Next_state = None
    Feed rewards and observations to model
    Model process next_state
    Record event (state, next_state, reward)
    state=next_state
    Forward state in neural network
    Agent take action(s) based on NN output
    Environment receives the action(s)
    Model update DQN based on events
End of while
Save learning model

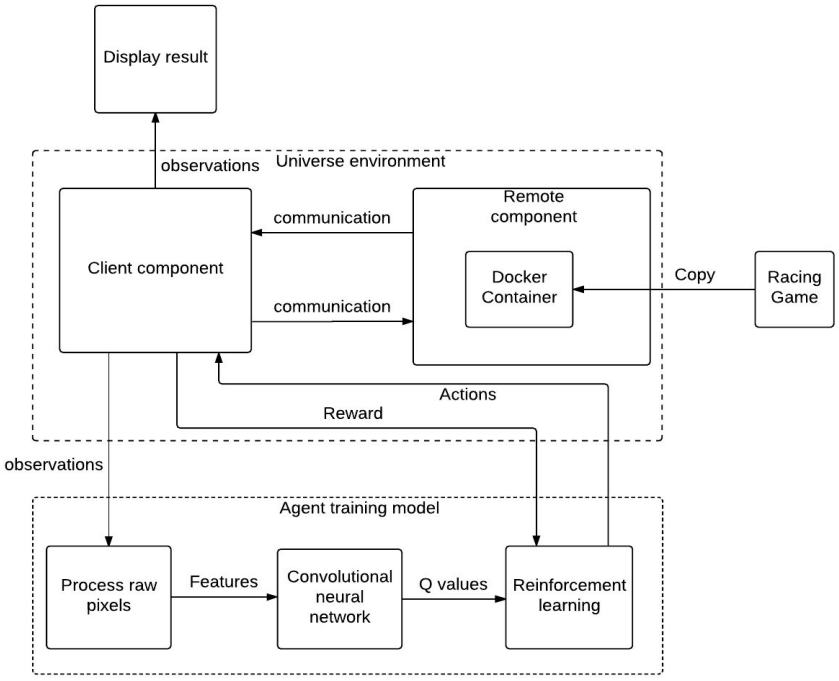

Diagram 1. Component Diagram

Algorithms Details

1. Process raw pixels
    a. Road segmentation.
    After analysis of image pixels, we found the road pixels' rgb follows regression $v = [t, t, t]^T$.
    Thus we can calculate distance from every pixel to this regression line, and set a threshold, if distance smaller than threshold, then this pixel belongs to road. The formula to calculate this distance is:

$$d = \frac{|(x-x_0)\times(x-x_1)|}{|(x_1-x_0)|}$$

    Where x is the pixel's rgb coordinate. And x_0, x_1 are two points on regression line. In the implementation of this algorithm, we use $x_0 = (0,\ 0,\ 0)\ and\ x_1 = (1,\ 1,\ 1)$ for simplification. And symbol $\times$ represent cross product of two vector.
    And we set threshold as 15. For pixel with $d <= 15$, we say pixel belongs to road.
    Then apply dilation and erosion on output binary image. Here is an example of road segmentation result.

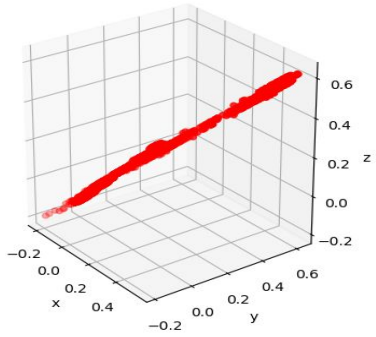

Figure 8. Pixel Regression Line

Figure 9. Road Segmentation

b. Feature representations

Furthermore, we tried several feature representations for the processed image and we will have a combination of them as part of the input for Neural Networks. Figure on left shows the result of applying adaptive thresholding with morphology. Middle Figure shows the result of using Canny edge detection. Right Figure shows the result of Scale-invariant Feature Transform(SIFT).

Figure 10. Thresholding + morphology

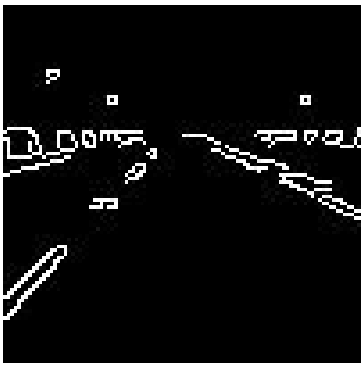

Figure 11. Canny

Figure 12. SIFT

c. Other tricks.

i. Image concatenate and Resize.

Only current frame is not enough for DQN to make judgement what current direction and speed is, so we need to send neural network more history frames. We concatenate rgb channels with 4 history canny edge detection results and 1 road segmentation through axis 2, so we finally get output in format Height*Width*Channels=512*800*8. And we apply resize on every channel to reduce output size, then we get final output 64*100*8.

2. Neural Network and DQN

a. Neural Network Analysis and Architecture

i. Input:

Eventually, we had to give up the adaptive thresholding and SIFT based on the following considerations:

- Adaptive thresholding and morphology does not eliminate the noise very well while Canny edge detection achieves better performance overall.

- SIFT captures the feature points very well and those feature points are very helpful in determining states. Unfortunately SIFT requires heavy computations, which slows down the training significantly.

After resize and concatenate of image, the final input have shape 64*100*8. In which 8 channels consist of 3 rgb channels, 4 history canny edge detections and 1 road segmentation.

ii. NN architecture:

We construct our Neural Networks based on the paper published by DeepMind Technologies and Pytorch tutorial, with some modifications. Table 1 shows the detailed settings for our Neural Networks.

Table 1. Neural Networks Architecture

Type	# of class/filter	Class/filter size	Stride	Activation
Conv-1	16	3 x 3	2	ReLU
Conv-2	32	4 x 4	2	ReLU
Conv-3	64	5 x 5	2	ReLU
FC-1	300	3200 x 300	N/A	ReLU
FC-2	# of Actions	300 x #of Actions	N/A	Linear

iii. Output:

There are totally 8 possible actions to take, but considering long training time and limited time left for this project, we only take 3 actions (UP, UP+LEFT, UP+RIGHT) which involve forward moving for training.

Given a state input, network should output scores for each action. Since we are using 3 actions at here, network should output 3 scores. Then we can apply some algorithm to make decision which action to take based on these scores. In this model, we actually take max score's corresponded action as final action to take.

b. DQN update and action selection

i. DQN and NN Update

The deep reinforcement learning uses a Neural Network to approximate function which take state as input and return score for every actions for decision make. The value for that action is actually the expected sum of discount rewards that we can get if we select to take this action under current state. Every time we make a decision on action to take, we record current state and action we take, also the next-state (we can get this when next state input into our model) as an event. Then we push this event to experience pool[10]. Then we randomly select events from experience pool and update DQN using these events. Based on bellman equation, the target Q value for state s and action a is

$$Q_{target}(s,\ a) = r + \gamma max_{a'} Q(s',\ a')),\ where\ s'\ is\ next\ state$$

Then we update current Q(s, a)

$$Q_{new}(s,\ a) = Q(s,\ a) + \alpha[Q_{target}(s,\ a) - Q(s,\ a)]$$

Then we use current $Q(s,\ a)$ and $Q_{new}(s,\ a)$ to define loss function for back propagation.

At here we use smooth_l1_loss as loss function, which is defined as following[7][11]:

$$loss(x,\ y) = \frac{1}{n}\sum_{} \begin{cases} 0.5 * (x_i - y_i)^2, & \text{if } |x_i - y_i| < 1 \\ |x_i - y_i| - 0.5, & \text{otherwise} \end{cases}$$

ii. Action Selection

To explore the solution space as much as possible, we use epsilon-greedy[10] algorithm for action selection, that is:

Define a epsilon which is in (0, 1) and will decay with time, take a random number in (0, 1). If this random number if larger than epsilon, we use the action select by the network as action we are going to take. Otherwise we randomly take an action. After a long period of training, the epsilon will be very small and most action will be selected by DQN.

c. DQN rewards design

To reach our goal, we need to define a good reward which can reflect the quality of current state. After many test, we combined following 3 partial rewards as final reward. And the final rewards is defined as

$$R = R_{as} + R_{cd} + R_{or}$$

i. Average speed.

The average of score changes during this observation. This can reflect if the car is driving in a good condition. We want the program to drive the car as fast and as smooth as possible, so this is very important. But we just find the score distributed on a large range, which brings difficulty for us to use this directly. So we use log function to smooth it. The reward of this part is defined as following:

$$R_{as} = log(mean(Scores) + 1)$$

ii. Crash detection

Because we can only get score changes during a short time. And we can use this to do crash detection. Because score changes represent distance the car moved between last frame and current frame. So according to the physical definition $v = s/t$ Score changes actually reflect average speed during this short period. And if we calculate derivative of v to t, we get $a = \partial v/\partial t$ which is the acceleration of the car. If the car move in constant velocity or accelerate, a should greater or equal to 0. Otherwise a should be smaller than 0. If a is smaller than 0, that means there must be something happened caused speed decrease. At here we make a simple assumption that crash is the main reason that cause the decrease of car speed. So to calculate the acceleration of this, we only need to calculate changes of score changes, notate it as ac. Then we define reward for crash detection as following:

$$R_{cd} = -p,\ when\ ac <= -50\ ,\ where\ p > 0$$
$$= 0,\ when\ ac >= -50$$

iii. On road detection

This reward is actually based on a manually defined rule: Car should drive on road. When car run off the road, we will give the model a little penalty. But to detect if the

car has run off the road is another big problem. Because we can only get images and scores from environment, we have no idea what happened inside the program. So we can only use what we have to design such rewards. Just like we mentioned above, we use the same method as road segmentation to do this. First we cut a small area in front of the car for detection. Then apply road pixel judgement on it. For this block, if there are more than 20 percent pixels don't belong to road, we can say car has run off the road. Here is the confusion matrix for this method. A whole episode ran for test.

Table 2. Confusion Matrix

	Label: Car off road	Label: Car on road
Algorithm: Car off road	103	19
Algorithm: Car on road	4	125

False positive rate of this algorithm $fp = 19/(19+125) = 13\%$

True positive rate of this algorithm $tp = 4/(4+103) = 3.7\%$

Accuracy of this $accuracy = (103+125)/(103+125+4+19) = 90.8\%$

Because we only want to punish the model when car run off road and also run at a low speed, so true positive rate is more important to us. Then this algorithm can actually works well. And we define the reward for crash detection as following:

$$R_{or} = -p,\ when\ over\ 20\%\ of\ that\ area\ not\ on\ road,\ p > 0$$
$$= 0,\ otherwise$$

Experiment and Future Work

1. Experiments Result
   We don't have much time for training, and current training result showed below. Left figure is the result that test result changes with time. Test is totally depend on DQN action selection, no random action selection will take place. We can see the longer we train, the less bad test scores we get. Right figure shows total score changes with training episode. We can see a increasing trend of it.

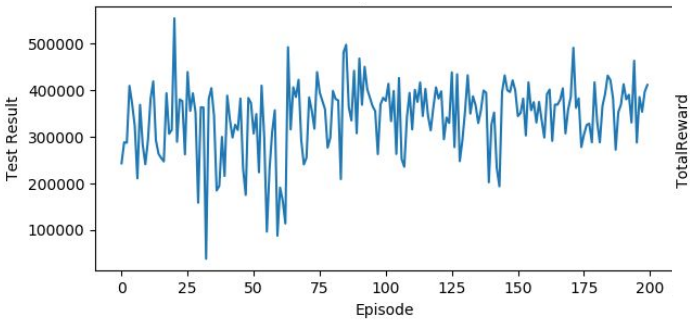

Figure 13. Test result changes

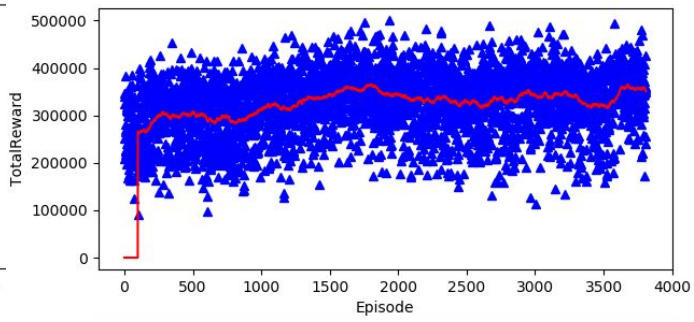

Figure 14. Total Reward changes

   We post a video on youtube which shows our agent's training result. https://youtu.be/qcY6T6Hn0jE.
2. Interesting facts
   a. Reinforcement Learning in this game will use certain abnormal rules in this game to increase its performance. For example, in this game, agent learned to hit lamp posts to trigger a reset mechanism which will reset the car in the middle of the road but don't decrease the car's speed.

b. When there are lots of cars on the road, agent will choose to run into grass near the road to avoid hitting with any of them.

3. Discussion
   We developed this model based on [9] and [11]. We tried to apply their model on this game, but it don't work. Actually the longer we train it, the worse it performance. So we implement current method. As a result, we are able to get an acceptable result: after 3800 episodes of training, the agent is able to gain a score over 450K and a race completion above 85%. At current stage, the agent is able to dodge some cars when driving along straight lane. It is also able to steer when it's need. However there are certain critical cases that it has not yet learned the optimal action:
   - It doesn't know how to respond to sharp turning.
   - In the last half of the race, there is a downhill followed by a right-turning. The agent will run out of the lane almost every time because the high velocity caused by the gravitational force will gain a high reward automatically therefore encourage the agent to keep moving forward instead of turning. Also, the inertia caused by the physics makes turning even harder.
   - Training of the model is super slow, will take days for training.
   - The way we define exploration method is too simple which result in that our model trained well on first half of the race, but not much trained on rest.
   - The way we input history frames is very simple. We just pushed history frames to CNN directly without any processing. Actually we should use a LSTM network after CNN to deal with this time sequence data.
4. Future Work
   Although our current agent is not able to gain 100% completion for the race, we are hopeful that it will achieve the goal. There are a few things we would like to try in the near future:
   - Replace the Deep Q-learning model with a Double Deep Q-learning model or Dueling network architecture.
   - Use different image preprocessing methods. For example, a more precise road segmentation method will be helpful for agent to learning to avoid hitting obstacles beside the road.
   - Define a better reward for agent to let it learn to finish the game as we want.
   - Use a new Neural Networks architecture that requires less computations so that each learning iteration can be done faster(This is also related to the choice of image preprocessing methods).
   - Because this is actually very complex for DQN training, so it took so much time on exploring environment on first half of the race. When it can finally reach the end of the lane, epsilon almost decay to 0, which means there is not much random exploration for the agent to explore rest racing lane. Maybe we need to define a smarter way to explore whole space.

Conclusion

Throughout the project, we realized appropriate pre-processing method play an important role in image processing, and also learned how to use some of the methods we learned in this semester to facilitate the machine learning. We have experienced how powerful reinforcement learning is and how much it is able to do. The most amazing thing we have learned is that reinforcement learning can work within an environment without having any presumptions if the state and reward is well designed(i.e.Our model can work with any car racing game, though performance will vary): it gets in the environment without any knowledge of it and learns about it by trying -- This is exactly how we human learn things.

References

1. Universe Library, OpenAI <https://github.com/openai/universe>
2. Dusk Drive, LongAnimals, <http://www.kongregate.com/games/longanimals/dusk-drive>.
3. Fredman Lex, M.I.T. “6.S094 Introduction to Deep Learning and Self-Driving Cars”. <http://selfdrivingcars.mit.edu/>
4. Li Fei-fei, Johnson Justin, Yeung Serenda, Stanford University, “CS231n: Convolutional Neural Networks for Visual Recognition”. <http://cs231n.stanford.edu/>
5. Robert Chuchro, Deepak Gupta. Stanford University, CS231n project report “Game Playing with Deep Q-Learning using OpenAI Gym” <http://cs231n.stanford.edu/reports/2017/pdfs/616.pdf> Accessed on Nov 5th 2017
6. DeepMind Technologies, NIPS Deep Learning Workshop,“Playing Atari with Deep Reinforcement Learning”, published in 2013 <https://arxiv.org/pdf/1312.5602v1.pdf> Accessed on Nov 5th 2017
7. Pytorch, “Reinforcement Learning(DQN) Tutorial” <http://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html>
8. Useful link for using Universe library: https://medium.com/@hoidnflo/off-policy-q-learning-in-openai-universe-part-1-setting-up-openais-baseline-dqn-6ae00f08a049
9. Mnih, Volodymyr, et al. "Human-level control through deep reinforcement learning." Nature 518.7540 (2015): 529-533.
10. Mnih, Volodymyr, et al. "Playing atari with deep reinforcement learning." arXiv preprint arXiv:1312.5602 (2013).
11. Girshick, Ross. "Fast r-cnn." Proceedings of the IEEE international conference on computer vision. 2015.