COMP5212 Machine Learning 2018 Fall programming project

Building a self-driving agent with Reinforcement Learning

Hok Chun Ng, 20272532, hcngac@connect.ust.hk
Shengyuan Zhang, 20565161, szhangcg@connect.ust.hk
Ge Chen, 20360858, gchenaj@connect.ust.hk

November 29, 2018

Abstract

In this project, we build a self-driving agent that only relies on vision using deep reinforcement learning algorithm. Due to the limitation of real-world hardware and data, we use an open source self-driving car simulator provided by Udacity [3] to generate training data and testing platform. We learn the elements of deep reinforcement learning and apply it to build our model. The results are presented with the loss of our deep reinforcement learning model and some sample test driving traces on the simulator.

1 Application and practical significance

Self-driving car has been a hot topic in recent years. Several industry giants, like Google [1] and Tesla Motor [2], have devoted significant efforts to the developing of self-driving cars.

Applications of self-driving agent can be wide. Example includes long-distance truck driving, smart taxi and bus, or even be incorporated into large-scale autonomous traffic systems.

Letting computers to drive can release human beings from the boring and tiring job of driving as well as reduce the frequency of traffic accidents. It also opens up new business and city-management opportunities including smart traffic system and smart taxi services. Applications are wide and undiscovered and the significance of perfecting self-driving agent is huge in opening the door to these huge possibilities.

However, designing a robust self-driving system is non-trivial as the real-world traffic conditions are diverse. Limited by hardware requirement and law, we find a solution to self-driving in a simulated game environment in the hope that the result can be transferred to real-world self-driving.

2 Problem formulation

We formulate the problem of self-driving as a continuous decision making process. The self-driving agent receives input from sensors, evaluate each possible actions and choose the best action.

For human drivers, the driving behavior can be formulated in a loop from an environment sensing input to action output. Our eyes and ears are the sensors to interpret the current environment, including the road view from wind screen (e.g. weather, traffic lights, walking people), view from side mirror (e.g. neighboring cars, following cars), car horns, and so on. Then our brain takes all these input signals to decide what action to take. The actions include turning the steering wheel, lighting turn signals, accelerating, braking, and so on. Once the actions are executed, the environment input changes, and again drivers will decide and take a new round of actions accordingly. This loop continues until the car arrives at the destination.

We try to mimic how human decision works with machine learning algorithms. The machine learning problem in self-driving can be: given a set of environment input, find the best actions to execute until arriving at destination.

3 Udacity Self-driving car simulator

In this project, we develop our algorithms on an open source self-driving car simulator provided by Udacity [3]. The simulator provides a near real-world environment of a racing car. The simulator can record video of every racing trace and then segment the video into image frames. The by default sample frequency is 30 frames per second. There are two modes in the simulator, one is "Training mode" for human controlled training records and the other is " Autonomous mode" for training by a machine agent. In Training mode, human player can drive the car under simulated environment, and there is a record button to capture video and store the images. We use "Training mode" to realize supervised learning and "Autonomous mode" to implement reinforcement learning. In Autonomous mode, the control of a car can be delivered by a python program with a web socket. The official document of Udacity provides a behavioral cloning repository [4] to interact with the simulation environment.

The controls to the car include the steering angle ranging from -25 to 25 and a throttle value of 0 and 1. To simplify the control, we set throttle value constantly to 0, and just apply the steering angle control. The end of a simulation is indicated by a sudden decrease of car speed (the highest speed in simulation is 30 miles per hour).

A sample image during training is shown in Figure 1, which is similar to a front view during human driving. The images are in RGB mode by default and the resolution can be set according to a user's preference. In our scenario, we choose the 320×160 as the resolution value to ease the training burden. To further reduce training overhead, we also convert the RGB images to gray-scale ones. After observing the sample images, we find that the gray-scale image data for the Red channel has better information for acting later on.

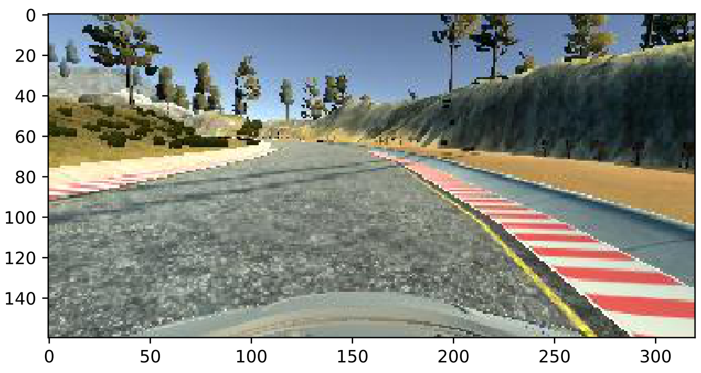
Figure 1: Sample image generated by the Udacity simulator

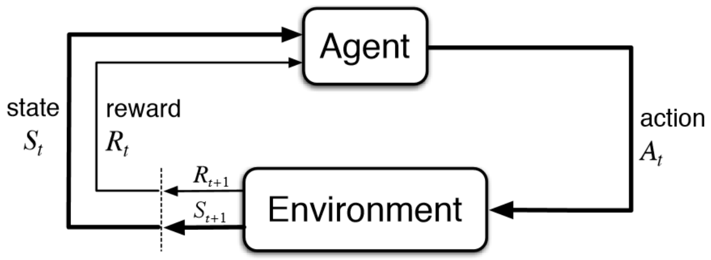

Figure 2: Reinforcement Learning Illustration [5]

4 Machine learning methods

4.1 Reinforcement learning

The environment-action loop in the driving experience resembles the basics of reinforcement learning, an area of machine learning concerned with taking actions in an environment in order to maximize some notion of cumulative reward. Unlike supervised learning, reinforcement learning does not require correct input/output pairs. Instead, reinforcement learning often requires a reward function in terms of different actions under the environment. For driving, the final reward can be no traffic accident and arrive at the destination safely.

Figure 2 shows the basic model of reinforcement learning [5]. At each time t, an agent receives the environment state $S_t \in \mathcal{S}$ and current scalar reward R_t. It then chooses an action $a_t \in \mathbb{R}^N$ from the set of available actions, which is sent to the environment.

The environment moves to a new state S_{t+1} and new reward R_{t+1} and feedback to the agent. The goal of a reinforcement learning agent is to gain as much as reward as possible. A policy π is defined to map states to a probability distribution over the actions $\pi : \mathcal{S} \rightarrow \mathcal{P}(\mathcal{A})$. The environment is stochastic

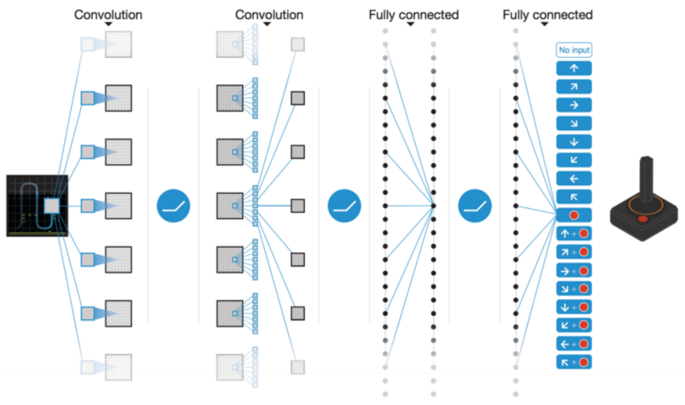

Figure 3: Deep Q-learning Network Illustration [7]

and we can model it as a Markov transition dynamics $p(s_{t+1}|s_t, a_t)$, and reward function $r(s_t, a_t)$. The return of a state is defined as the sum of discounted future reward $R_t = \sum_{i=t}^{T} \gamma^{(i-t)} r(s_i, a_i)$, where $\gamma \in \ [0, 1]$ is a discounting constant.

Many reinforcement learning apply the action-value function to decide the most valuable action given a state. It describes the expected return after taking an action a_t given s_t by following policy π:

$$Q^{\pi}(s_t, a_t) = \mathbb{E}_{r_i \geq t, s_{igeqt} \sim E, a_{i>t} \sim \pi}\left[R_t | s_t, a_t\right]$$

Moreover, many algorithms in reinforcement learning take the Bellman equation form of the action-value function, which is

$$Q^{\pi}(s_t, a_t) = \mathbb{E}_{r_t, s_{t+1} \sim E}\left[r(s_t, a_t) + \gamma \mathbb{E}_{a_{t+1} \sim \pi}[Q^{\pi}(s_{t+1}, a_{t+1})]\right]$$

4.1.1 Q learning

Q-learning [6] is a commonly used off-policy algorithm in reinforcement learning, which uses the greedy policy $\mu(s) = \arg\max_a Q(s, a)$. It defines a reward expectation function, Q, which provides the expected reward of an action a_t taken under the current state s_t. Assume function approximators aare parameterized by θ^Q, the Q learning tend to optimize by minimizing the loss:

$$L(\theta^Q) = \mathbb{E}_{s_t, a_t, r_t \sim E}\left[\left(Q(s_t, a_t|\theta^Q) - y_t\right)^2\right]$$

,where

$$y_t = r(s_t, a_t) + \gamma Q(s_{t+1}, \mu(s_{t+1})|\theta^Q)$$

4.1.2 Deep Q learning

The massive use of non-linear function approximators for learning value or action-value functions has been depreciated in the past since the convergence is no longer guaranteed, and thus the practical learning tend to be unstable.

To search for practical solution, Mnih et al. [?] apply large neural networks as function approximators, which is known as Deep Q learning. The motivation behind is that, in our case with the large state space $\mathcal{S}$ of front images during driving, it is impossible to directly compute Q. Alternatively, we can use a deep convolutional network to estimate Q.

Figure 3 illustrates a model of deep Q learning network [7]. First, several layers of convolution and max-pooling help reduce the imagery input into smaller feature vector. Then we have several fully-connected layer for the final decision making on the control of a car. The sample network shown estimates a function $Q : S \to A^n$, which takes a image input and output the expected reward for all possible actions.

One challenge when applying neural networks for reinforcement learning is that most optimization algorithms assume that the training samples are independently and identically distributed. However, this is no longer the case when exploring sequentially in a stochastic environment. Usually, the sequential actions result in states with strong correlation, which will easily lead to over-fitting. As in Deep Q learning, a replay buffer is used to address this issue. The replay buffer is a finite size cache containing randomly sampled historical tuples (s_t, a_t, r_t, s_{t+1}). When the replay buffer becomes full, the oldest samples will be discarded. At each epoch, the latest states and some from the replay buffer are mixed to form a minibatch, allowing the algorithm to learn across a set of uncorrelated transitions.

4.1.3 Continuous control for deep reinforcement learning

In this project, we use the method of deep reinforcement learning for continuous control [8]. The reason to apply this model is that our action space is the steering angle, which is a continuous real value from -25 to 25. One major challenge with continuous action space (e.g. steering angle) is exploration. Researchers in [8] claim to add an exploration policy μ' to solve this problem by adding noised sampled from a noise process $\mathcal{N}$ to the policy:

$$\mu'(s_t) = \mu(s_t|\theta_t^\mu) + \mathcal{N}$$

Now, in addition to a neural network estimating $Q(s, a|\theta^Q)$, we add another neural network to estimate our policy $\mu'(s_t|\theta^\mu)$.

5 Problem formulation

Now we show our problem formulation for both supervised learning and reinforcement learning with neural networks.

5.0.1 Supervised learning

In supervised learning, we first record a set of images and actions taken by human players on the simulator. We start the simulator in “Training mode” and record all images frames into a local data folder, along with the statistics of steering angle, speed, throttle. We collect a total of XXX image frames. We separate the whole data set of images by a training-test ratio of 4:1. To reduce overhead, we discretisize the action space of steering angle into limited

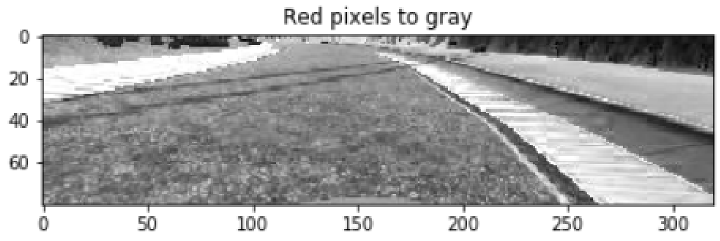

Figure 4: Gray-scale image data input

intervals, and use these intervals to label each image frame. Now, a sample of data (X, y) is defined as: X is a image frame, y is the action (interval id from 0 to N) taken by human player. This becomes a multi-class image classification problem similar to PA2.

The neural network architecture we use is described in the following:

Component	Layer	#Filters	Kernel size	Strides	Input Shape	Output Shape
Encoder	ConvLayer	32	3×3	1	$320 \times 80 \times 1$	$320 \times 80 \times 1$
	MaxPooling	-	2×2	2	$320 \times 80 \times 1$	$320 \times 80 \times 1$
	ConvLayer	32	3×3	2	$320 \times 80 \times 1$	$320 \times 80 \times 1$
	ConvLayer	32	3×3	1	$320 \times 80 \times 1$	$320 \times 80 \times 1$
	MaxPooling	-	2×2	2	$320 \times 80 \times 1$	$320 \times 80 \times 1$

Table 1: Encoder for Supervised Learning

Component	Layer	#Units	Input Shape	Output Shape
Encoder	shown in table 1			
Predictor	FC	1024	$320 \times 80 \times 1$	$320 \times 80 \times 1$
	FC	512	$320 \times 80 \times 1$	$320 \times 80 \times 1$
	FC	25	$320 \times 80 \times 1$	$320 \times 80 \times 1$

Table 2: Encoder and Predictor for Supervised Learning

5.0.2 Reinforcement learning

Environment setting Action space Reward function

6 Experiment and performance evaluation

TODO

6.1 Neural Network structure

The sample input of our training/test model is show in Figure 4. input layers output

6.2 Hyper-parameters

6.3 Loss function/ final loss value

6.4 Sample trained agent

sample outputs of each layer if possible.

Our experiment involves deep Q network training and performance evaluation.

Training is a process of random discovery. The agent is allowed to randomly choose actions in the game until game over. States, action and reward is recorded in the process and is used to train the deep Q network. We evaluate the performance in fixed game intervals to show the learning progress. Training stops when the performance does not increase over certain extend.

In performance evaluation, we let the self-driving agent control the car according to the output of the deep Q network. The agent feed the image display to the network and choose the action with the highest output value in each step, until game over. We run the process for a number of times to get an average performance in each evaluation. Performance is defined as the score obtained at game over.

References

[1] Google Self-Driving Car, `https://www.google.com/selfdrivingcar`

[2] Tesla Self-Driving Hardware, `https://www.tesla.com/autopilot`

[3] Udacity Self-driving Simulator, `https://github.com/udacity/self-driving-car-sim`

[4] Udacity CarND Behavioral Cloning, `https://github.com/udacity/CarND-Behavioral-Cloning-P3`

[5] Sutton, Richard S., Andrew G. Barto, and Francis Bach. `Reinforcement learning: An introduction.` MIT press, 1998.

[6] Christopher Watkins, Peter Dayan. `Q-learning.` Machine learning, 1992.

[7] Volodymyr Mnih, Koray Kavukcuoglu, et al. `Human-level control through deep reinforcement learning.` Nature, 2015.

[8] Timothy Lillicrap, Jonathan Hunt, et al. `Continuous control with deep reinforcement learning` ICLR, 2016.