2018 Spring Robot Lab
Undergraduate Final Report

Topic: Robots Gain Social Intelligence Through Reinforcement Learning

2018.07.06

Advisor: Li-chen Fu
Senior: Zih-yun Chiu
Student: Szu-Yu Mo

1. Outline

2. Project goal

We want to make Pepper a well-trained receptionist (Fig. 1). For example, if Pepper is a receptionist of a bank, we expect him to discover people coming in and try to greet them. Having some simple interactions with customers, Pepper would make higher value of user experience.

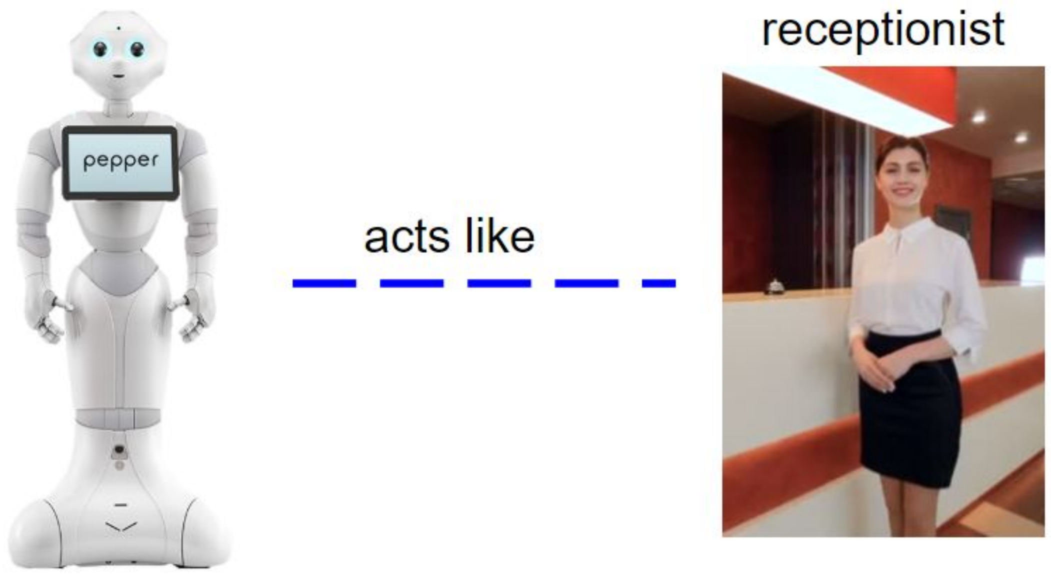


Fig. 1 Pepper as a receptionist

The project goal is to make Pepper learn some basic social skills through deep reinforcement learning. We define the state for Pepper to determine his action to be eight consecutively taken photos of his built-in camera, so that it mimics a person's viewing of the surroundings. As for deep reinforcement learning, the quality and the amount of training data are the most important. Unfortunately, there isn't any open source data relating to our environment setup, so we have to collect all data by ourselves, which is proved to be extremely time-consuming. Nevertheless, the experiments are setup as the following shows.

3. Environment setup

(1) Action space

Due to the difficulties to teach Pepper about social intelligence, we have only picked four actions for Pepper to learn. At the last semester, "do nothing", "shake hands", "say hello", and "say goodbye" are the actions. However, "saying goodbye" is somehow unnecessary because Pepper can just say goodbye right after shaking hands. Moreover, the actions lack for Pepper coming closer to the human. Therefore, in this semester, we redefine the four actions to be "do nothing", "wave hands and say hello", "moving forwards to human", and "shake hands" (Fig. 2), which is also known as action space.

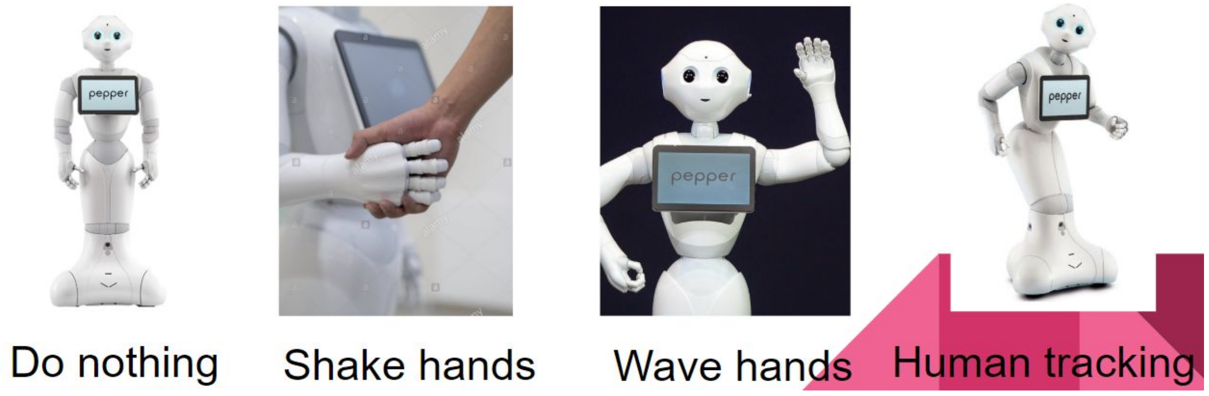


Fig. 2 Action space of Pepper

(2) Observation space

Observation space is another requirement for the reinforcement learning environment structure. It is mentioned that the state would be eight consecutively taken photo by Pepper in the previous section. The reason not to only record a single image is that we would like to collect data representing the time sequence. Not surprisingly, the observation space should be some constraints dealing with those images. The image resolution is chosen to be 120 pixels time 160 pixels, each with gray scale value 0 to 255, and these together construct the observation space. An example of a state is shown in Fig. 3.

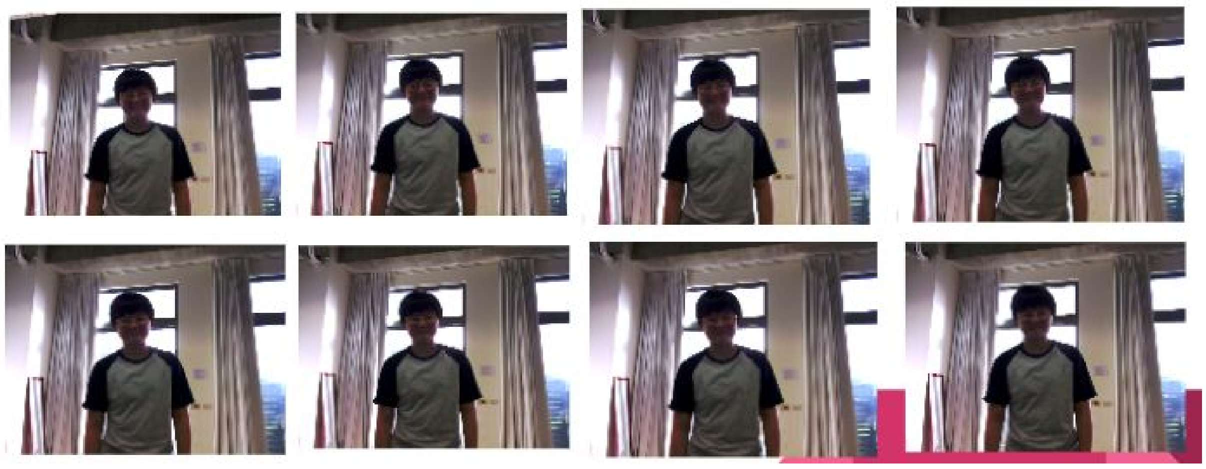


Fig. 3 An example of a state

(3) Triggering of Pepper

Pepper would not start an episode unless someone comes into his sight (which is a natural reaction of human). Otherwise, Pepper would be crazily picking his action no matter what is in front of him. It is very possible for him to collect a lot of useless data with images without human. So what’s the trigger for Pepper to start the interaction? The answer is human detection. Human detection is implemented with an object detection model “Detectron”, which is published by Facebook this spring. This tool is quite powerful in that a person can be detected within approximately five meters, which is much further than the built-in human detection SDK of Pepper. The latter can only find people within 1.8 meters. The following table is a comparison of two models (Table 1).

Model	Detectron	Human detection API
source	Facebook	Pepper SDK
detection distance	5 meters	1.8 meters
computing hardware	GPU	CPU
accuracy	High	Low

Table 1 Comparison between Detectron and Human detection API

(4) Default policy

Before any training, Pepper decides its manner by a raw model.

The raw model is to randomly choose an action from the action space. In an episode, the action would be removed from the action space after chosen by Pepper, in order to avoid redundancy of data. Only when Pepper tries to shake hands with human is the episode going to terminate.

(5) Reward

During the termination, the system (code written by us) will act as a teacher to score his performance. On one hand, If he successfully shakes hands with a human (judgement by the touch sensor on his right hand), he gets one point for reward. On the other hand, if he fails to shake, he gets minus point one point for penalty (Fig. 4).

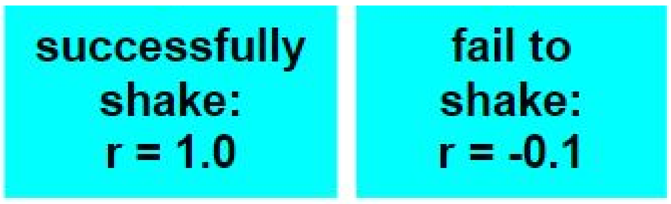


Fig. 4 Reward for Pepper

(6) Transition pair

Except for the eight consecutive pictures that Pepper has to record, the transition pair is also a really important data for training. It not only tells which pictures correspond to this state, but also tells the action Pepper has chosen, the reward he has got for this step, and whether this state terminates or not as well (Fig. 5). Those data are saved into a pickle file and then reloaded while training.

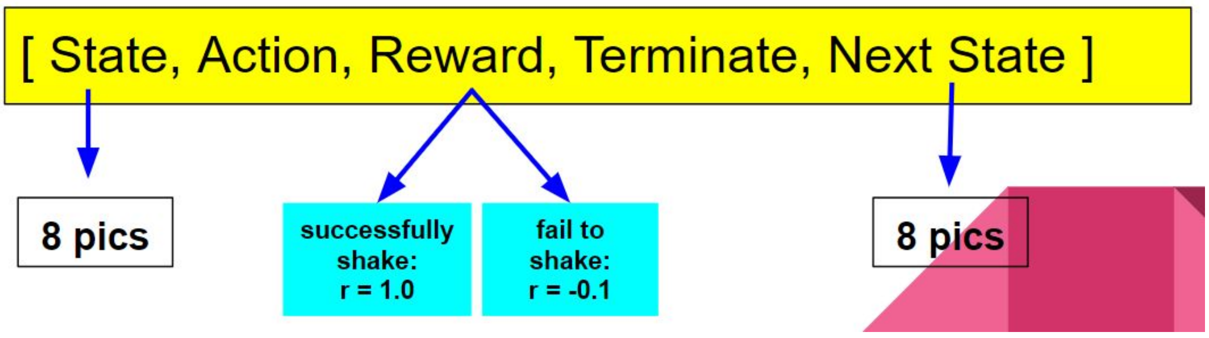

[State, Action, Reward, Terminate, Next State]
8 pics
successfully shake: r = 1.0
fail to shake: r = -0.1
8 pics

Fig. 5 Transition pair

4. Data collection

(1) Location

The data collection was carried out mainly at two locations. One is room 412 at AI center, and the other is the first floor of the same building, YongLin Biomedical Engineering Hall. Some images taken at these two places are shown in Fig. 6 & Fig. 7.

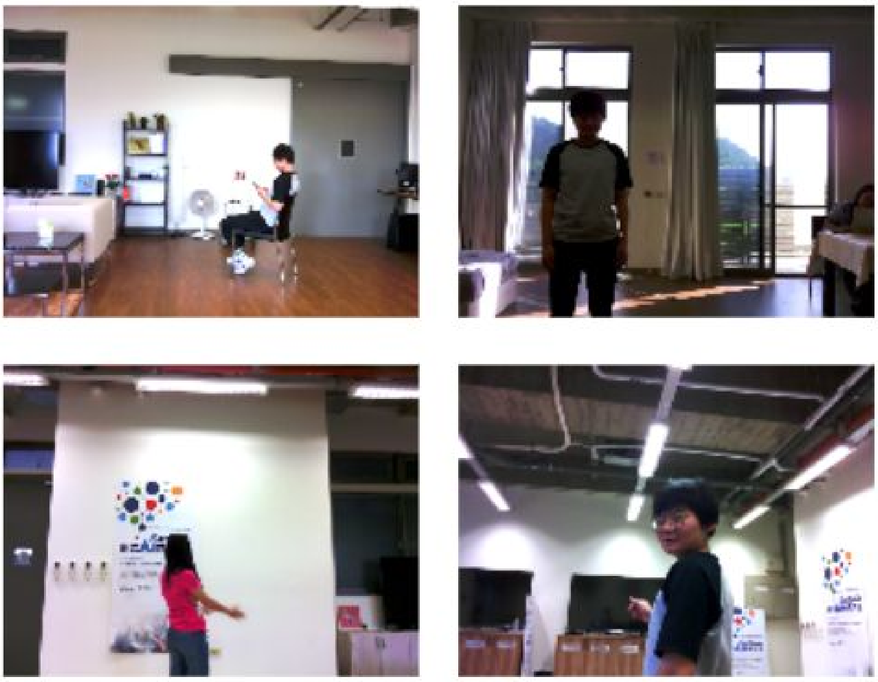


Fig. 6 Images shot at room 412

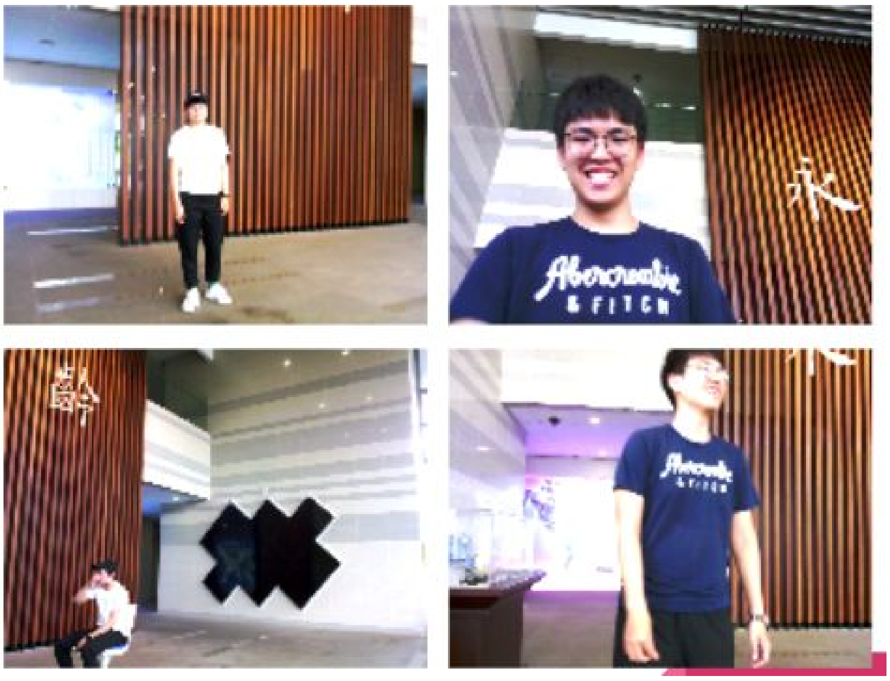


Fig. 7 Images shot at YongLin first floor

The reasons that only two locations are utilized are that it is really difficult to carry Pepper around, and plus, strong wifi connection is required for transmitting data between him and computer.

(2) Some scenarios demonstrating

The number of scenarios can be infinite. People can have tons of reactions after the same action done by Pepper in different circumstances. Two scenarios along with the pictures taken during that episode are shown in Fig. 8 & Fig. 9. Fig. 8 shows a situation in which the person is not showing willingness to talk with Pepper. Fig. 9 tells a circumstance that the person is very excited to interact. It should be noted that an episode could contain two to five transition states. Thus, the number of images in an episode is not constant.

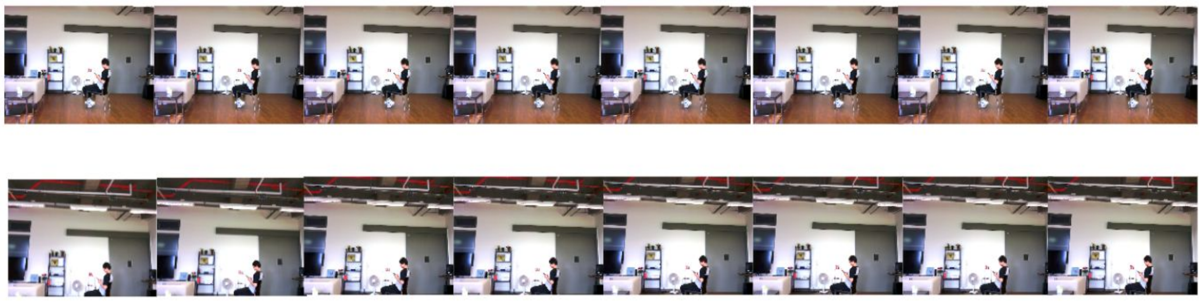


Fig. 8 People unwilling to interact scenario

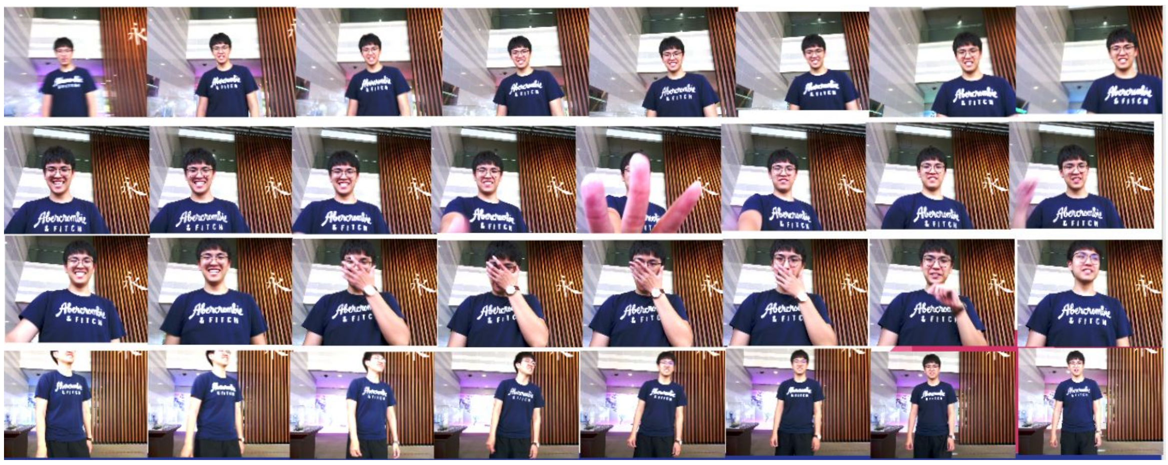


Fig. 9 People willing to interact scenario

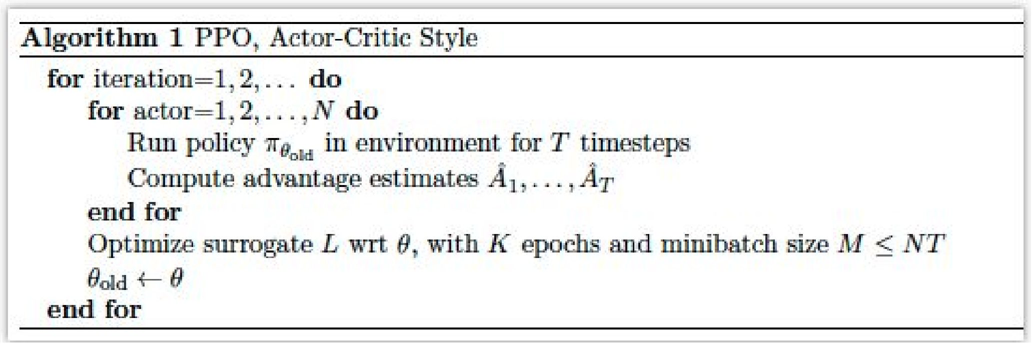
(3) Progress up to now

After a lot of efforts and consuming time, until now, 130 episodes as well as 304 transition pairs are recorded. Unfortunately, they might not be enough for training deep reinforcement learning model…

5. PPO1 (proximal policy optimization)

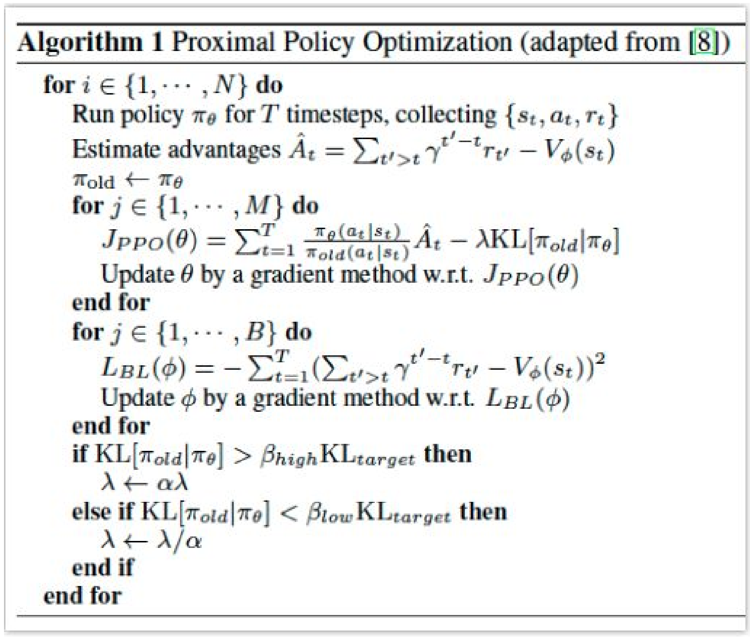
This is a deep reinforcement learning package that is open-sourced by OpenAI. It is based on actor-critic algorithm, which kind of fuses deep Q network and policy gradient together. The original paper of actor-critic algorithm states that they are stochastic gradient algorithms on the parameter space of the actor. However, the learning rate is difficult to determine. If the learning rate is too big, policy won't converge; if it is too small, we will wait for day and nights for training.

PPO1 solves the problem of the uncertainty of learning rate in policy gradient by limiting the range that new policy updates below the percentage of new/old policy. Actually, Google AI and OpenAI were competing with each others at the time they both released the paper regarding to this algorithm (Fig. 10 & Fig. 11). The both algorithms published by them are similar but not the same totally. The main difference is the gradient to update actor. OpenAI chose to clip surrogate objective (Fig. 12), while Google AI plugged in KL penalty (Fig. 13). Both of them can reach the same goal.



```
Algorithm 1 PPO, Actor-Critic Style
for iteration=1, 2, ... do
    for actor=1, 2, ..., $N$ do
        Run policy $\pi_{\theta_{\text{old}}}$ in environment for $T$ timesteps
        Compute advantage estimates $\hat{A}_1, \dots, \hat{A}_T$
    end for
    Optimize surrogate $L$ wrt $\theta$, with $K$ epochs and minibatch size $M \leq NT$
    $\theta_{\text{old}} \leftarrow \theta$
end for
```

Fig. 10 Algorithm published by OpenAI



```
Algorithm 1 Proximal Policy Optimization (adapted from [8])
for $i \in \{1, \cdots, N\}$ do
    Run policy $\pi_\theta$ for $T$ timesteps, collecting $\{s_t, a_t, r_t\}$
    Estimate advantages $\hat{A}_t = \sum_{t'>t} \gamma^{t'-t} r_{t'} - V_\phi(s_t)$
    $\pi_{\text{old}} \leftarrow \pi_\theta$
    for $j \in \{1, \cdots, M\}$ do
        $J_{PPO}(\theta) = \sum_{t=1}^{T} \frac{\pi_\theta(a_t|s_t)}{\pi_{old}(a_t|s_t)} \hat{A}_t - \lambda \mathrm{KL}[\pi_{old}|\pi_\theta]$
        Update $\theta$ by a gradient method w.r.t. $J_{PPO}(\theta)$
    end for
    for $j \in \{1, \cdots, B\}$ do
        $L_{BL}(\phi) = -\sum_{t=1}^{T} (\sum_{t'>t} \gamma^{t'-t} r_{t'} - V_\phi(s_t))^2$
        Update $\phi$ by a gradient method w.r.t. $L_{BL}(\phi)$
    end for
    if $\mathrm{KL}[\pi_{old}|\pi_\theta] > \beta_{high} \mathrm{KL}_{target}$ then
        $\lambda \leftarrow \alpha\lambda$
    else if $\mathrm{KL}[\pi_{old}|\pi_\theta] < \beta_{low} \mathrm{KL}_{target}$ then
        $\lambda \leftarrow \lambda/\alpha$
    end if
end for
```

Fig. 11 Algorithm published by Google AI

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t\left[\min(r_t(\theta)\hat{A}_t, \operatorname{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)\right]$$

Fig. 12 Clipped surrogate objective released by OpenAI

$$L^{KLPEN}(\theta) = \hat{\mathbb{E}}_t\left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}\hat{A}_t - \beta \operatorname{KL}[\pi_{\theta_{\text{old}}}(\cdot \mid s_t), \pi_\theta(\cdot \mid s_t)]\right]$$

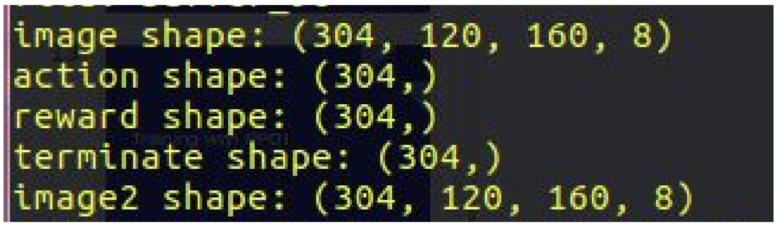
Fig. 13 KL penalty released by Google AI

6. Training

I modified the code of PPO1, which was used to train Atari games originally, to the version I could use on my data. The followings are the efforts I made.

(1) Loading data

Each state contains eight pictures with shape=(120, 160,3), where “3” stands for three channels of RGB. I convert the RGB image into grayscale, so the image shape becomes (120, 160), omitting the only channel in the last index. After that, I concatenate eight pictures together to represent the state. Finally, the shape of the image is (120, 160, 8). Of course, other data such as action, reward, and terminate are also loaded (Fig. 14).

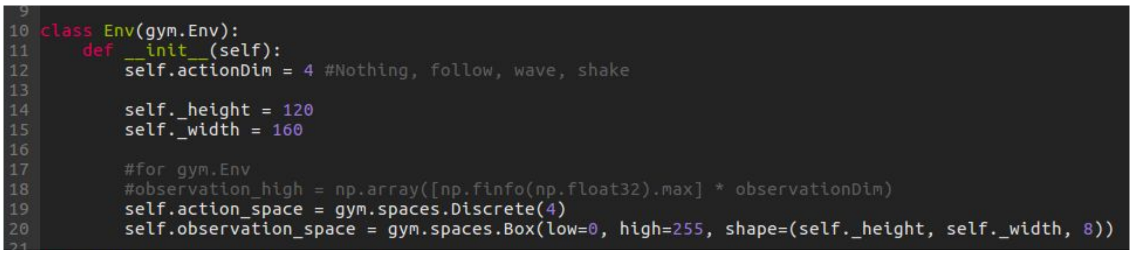


```
image shape: (304, 120, 160, 8)
action shape: (304,)
reward shape: (304,)
terminate shape: (304,)
image2 shape: (304, 120, 160, 8)
```

Fig. 14 Loading data

(2) Environment class

I have to reconstruct a class that fits our experiment environment. In the meanwhile, the environment has to inherit gym.env, which is a class that the original environment with Atari also inherits. I set the action dim, height and width of an image, action space, and observation space (Fig.15).

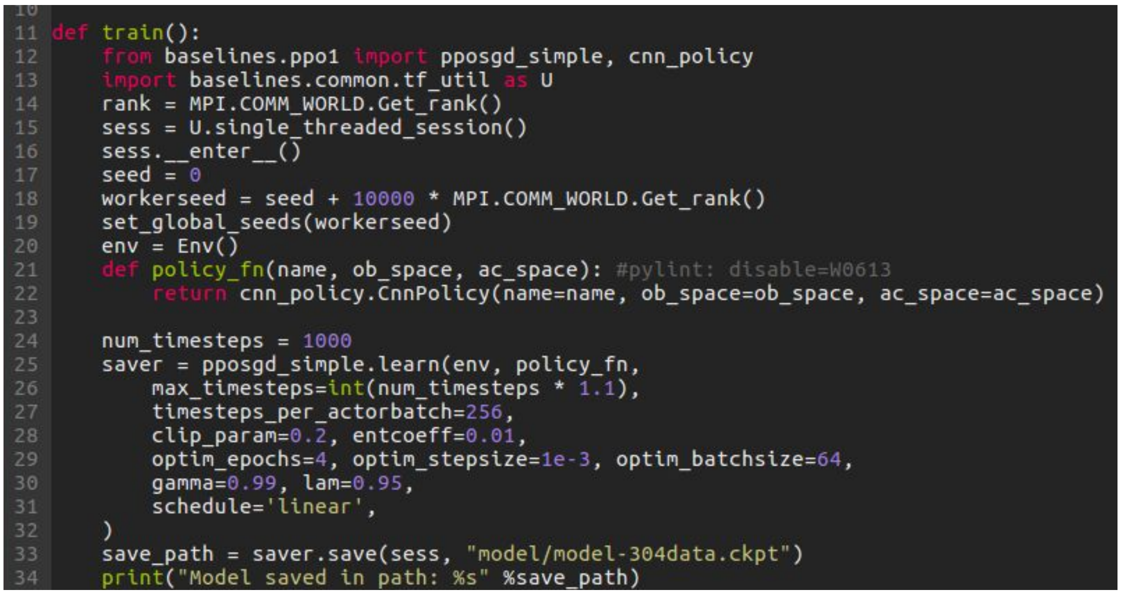


```python
class Env(gym.Env):
    def __init__(self):
        self.actionDim = 4 #Nothing, follow, wave, shake

        self._height = 120
        self._width = 160

        #for gym.Env
        #observation_high = np.array([np.finfo(np.float32).max] * observationDim)
        self.action_space = gym.spaces.Discrete(4)
        self.observation_space = gym.spaces.Box(low=0, high=255, shape=(self._height, self._width, 8))
```

Fig. 15 Rewriting environment

(3) On-line learing to off-line learning

At first, PPO1 was designed to train on-line. Unlike this project, they continue to receive data while acting with the environment. It can be trained with more and more data during training process. However, in our case, the data are collected in advance. We need to train off-line. Below is the code that I modified to make it off-line training (Fig. 16).

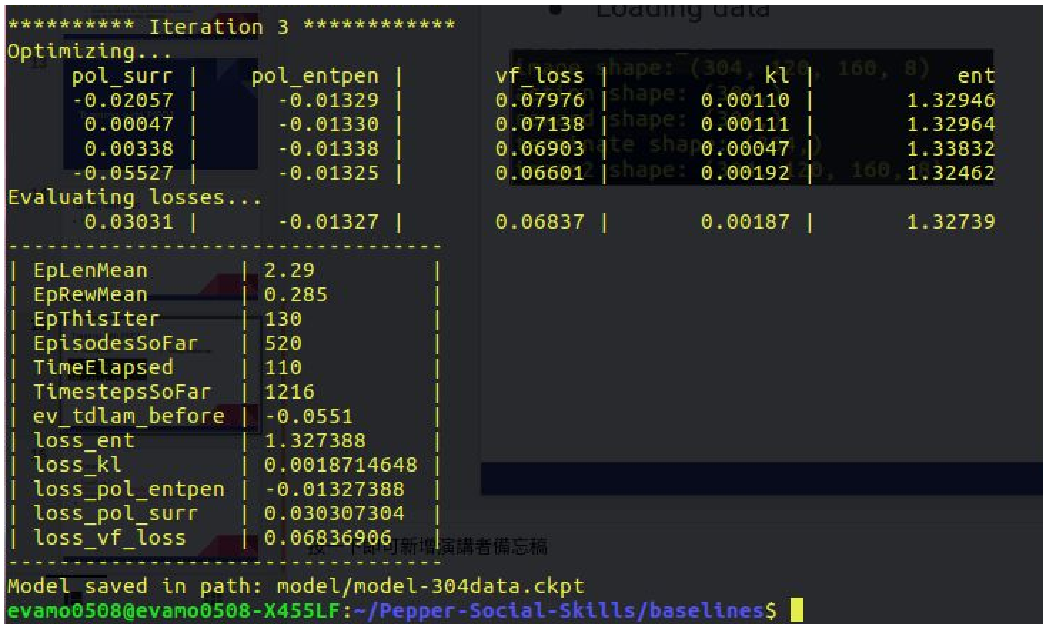


```python
def train():
    from baselines.ppo1 import pposgd_simple, cnn_policy
    import baselines.common.tf_util as U
    rank = MPI.COMM_WORLD.Get_rank()
    sess = U.single_threaded_session()
    sess.__enter__()
    seed = 0
    workerseed = seed + 10000 * MPI.COMM_WORLD.Get_rank()
    set_global_seeds(workerseed)
    env = Env()
    def policy_fn(name, ob_space, ac_space): #pylint: disable=W0613
        return cnn_policy.CnnPolicy(name=name, ob_space=ob_space, ac_space=ac_space)

    num_timesteps = 1000
    saver = pposgd_simple.learn(env, policy_fn,
        max_timesteps=int(num_timesteps * 1.1),
        timesteps_per_actorbatch=256,
        clip_param=0.2, entcoeff=0.01,
        optim_epochs=4, optim_stepsize=1e-3, optim_batchsize=64,
        gamma=0.99, lam=0.95,
        schedule='linear',
    )
    save_path = saver.save(sess, "model/model-304data.ckpt")
    print("Model saved in path: %s" %save_path)
```

Fig. 16 Code for off-line training

(4) Training

Due to the fact that the amount of data is quite a little, I only trained the network for 4 iterations. Rather than overfitting, I would prefer to make Pepper learn a little it further than the beginning. Below is the last iteration of training (Fig. 17).



```
********** Iteration 3 ************
Optimizing...
     pol_surr |    pol_entpen |       vf_loss |         kl |         ent
     -0.02057 |      -0.01329 |       0.07976 |    0.00110 |     1.32946
      0.00047 |      -0.01330 |       0.07138 |    0.00111 |     1.32964
      0.00338 |      -0.01338 |       0.06903 |    0.00047 |     1.33832
     -0.05527 |      -0.01325 |       0.06601 |    0.00192 |     1.32462
Evaluating losses...
      0.03031 |      -0.01327 |       0.06837 |    0.00187 |     1.32739
----------------------------------
| EpLenMean       | 2.29          |
| EpRewMean       | 0.285         |
| EpThisIter      | 130           |
| EpisodesSoFar   | 520           |
| TimeElapsed     | 110           |
| TimestepsSoFar  | 1216          |
| ev_tdlam_before | -0.0551       |
| loss_ent        | 1.327388      |
| loss_kl         | 0.0018714648  |
| loss_pol_entpen | -0.01327388   |
| loss_pol_surr   | 0.030307304   |
| loss_vf_loss    | 0.06836906    |
----------------------------------
Model saved in path: model/model-304data.ckpt
evamo0508@evamo0508-X455LF:~/Pepper-Social-Skills/baselines$
```

Fig. 17 Performance of the last iteration

7. Demo video

After the first training with three hundred transition pairs, the model is implemented into Pepper. Now, for the rest of data collection process, Pepper would not have to choose an action anymore. Instead, which action to pick is now computed by computer using the model. In that way, we can always retrain our model with new data, looking forward to an extreme accurate model making Pepper a perfect receptionist.

Below are three links of demo video, demonstrating how Pepper acts after implementing the first trained model into it.

(1) An example with successful interaction: https://goo.gl/rVepYb
(2) Another example with successful interaction: https://goo.gl/Vkz6hL
(3) An unsuccessful interaction: https://goo.gl/h6Rnv4

We can see that Pepper really likes to shake hands with people now. Maybe it is because successfully shaking hands earns many points for him, so he tends to choose this action. Obviously, the model isn't accurate enough.

8. Future improvement

There are many ways to improve the accuracy of the model. The followings are some of them:

(1) Collect more and more data.

(2) Increase the resolution of the image so that we can extract features such as face orientation or body gesture from it.
(3) Create virtual image by combining different parts of a person and do the labeling by ourselves.

Those methods seem to be nice. Nevertheless, method (1) is really, really time-consuming, and Pepper is too difficult to carry around to raise the environment variety. For method (2), taking high resolution photos would cause a great delay during interactions, which would definitely make users feel uncomfortable. As for method (3), I haven’t done any experiments with it yet. Maybe It will bring out well, but I assume the images won’t be too natural.

9. Conclusion

A deep reinforcement learning model was trained and implemented to Pepper. Although the accuracy is not high enough due to the difficulties of collecting sufficient data, Pepper did learn something such as moving forwards to the person before shaking hands. More techniques with getting data are indispensable and would promise a better model through retraining with those data.

10. Reference

(1) OpenAI ppo1：https://github.com/openai/baselines/tree/master/baselines/ppo1
(2) Detectron：https://github.com/facebookresearch/Detectron
(3) Tensorflow：https://www.tensorflow.org/get_started/eager
(4) Morvan Python DPPO：https://morvanzhou.github.io/tutorials/machine-learning/reinforcement-learning/6-4-DPPO