COMP219 - 2018 - Train Deep Learning Agents

Name: Hongyi Wu ID: 201376889

document explaining your connection to openAI universe/gym, your network architectures, your training parameters, and the results.

Check list:

1. Import an OpenAI universe/gym game
2. Creating a network
3. Connection of the game to the network
4. Deep reinforcement learning model
5. Experimental results

Import an OpenAI universe/gym game

The import phase is not complicated as there are plenty of games in gym, and we just need to import **gym** module:

```
1. import gym
2. env = gym.make('CartPole-v0').unwrapped
```

Note that we use unwrapped to make it unconstrained to get more information. After that, the game will be initialized.

As for the game, it terminates in three situations:

1. Pole Angle is more than ±12°
2. Cart Position is more than ±2.4 (center of the cart reaches the edge of the display)

3. Episode length is greater than 200

Creating a network

I created a **DQN** class which also contains CNN model. As for the construction of the network, it is inherited from DQN class by code:

```
super(DQN, self).__init__()
```

Then three convolutional layers are generated and nomarlised:

```
1. self.conv1 = nn.Conv2d(3, 16, kernel_size=5, stride=2)
2. self.bn1 = nn.BatchNorm2d(16)
3. self.conv2 = nn.Conv2d(16, 32, kernel_size=5, stride=2)
4. self.bn2 = nn.BatchNorm2d(32)
5. self.conv3 = nn.Conv2d(32, 32, kernel_size=5, stride=2)
6. self.bn3 = nn.BatchNorm2d(32)
```

After series of image processing, the model should forward to use the Rectified Linear function,

```
1. x = F.relu(self.bn1(self.conv1(x)))
2. x = F.relu(self.bn2(self.conv2(x)))
3. x = F.relu(self.bn3(self.conv3(x)))
```

Note that **torch.nn.functional** is abbreviated by F. after that the basic network is constructed.

Connection of the game to the network

As for the connection, the program is designed to detect and process the screen firstly.
In this project, all the frames are rendered and it makes use of the scenes rather than environment states. The **torchvision** package is ued for input extraction.
Firstly, the cart location is detected by **get_cart_location(screen_width)**, and the screen image after processing should be transposed by **get_screen.**

After that, we call **get_screen** function and get the information of scripted image by the code:

```
1. init_screen = get_screen()
2. _, _, screen_height, screen_width = init_screen.shape
```

As for DQN optimized algorithm, it aims to train a policy that tries to maximize the discount. Therefore, we need two networks participating in the training process.

```
policy_net = DQN(screen_height, screen_width).to(device)

target_net = DQN(screen_height, screen_width).to(device)

target_net.load_state_dict(policy_net.state_dict())
#load previous model
target_net.eval()
```

then we run the **ReplayMemory** class to replay the image of previous actions and results (default **times** = 10000)

Deep reinforcement learning model

The model we selected is DQN, which is a kind of Off-policy algorithms. It firstly samples a batch and join all the tensors. After the calculation, they are combined into the loss.

In the **optimized_model** function, it firstly transposes the batch to computable variable, and then compute a mask of states and concatenate the batch elements. Then compute the Q values and Huber loss.

As the target_net mentioned above, it is used to improve the stability. Normally, optimization selects a random batch (by **random()**), which is from **ReplayMemory**, and then train for a new policy. The **target_net** is also used in this phase to calculate result.

As for the parameters, they are:

```
BATCH_SIZE = 128
GAMMA = 0.999
EPS_START = 0.9
EPS_END = 0.05
EPS_DECAY = 200
TARGET_UPDATE = 10
```

The **select_action** function is worth mentioning. The function will generate a random number and sometimes just sample one uniformly rather than getting action from the model, which aims to reduce the error.

```
global steps_done
    sample = random.random()
```

```
    eps_threshold = EPS_END + (EPS_START - EPS_END) * \
        math.exp(-1. * steps_done / EPS_DECAY)
    steps_done += 1
    if sample > eps_threshold:
        #take action randomly
        #if random number satisfies, take action randomly

        with torch.no_grad():
            # select a largest reward as action
            return policy_net(state).max(1)[1].view(1, 1
)
    else:
        return torch.tensor([[random.randrange(2)]], dev
ice=device, dtype=torch.long)
```

Experimental results

As for the experimental results, a plot function is deployed during the training process.
The function plots the episode as X label, and duration as Y label. As mentioned in the figure, the blue one describes each performance of episode, and the orange one focuses on mean of performance when the episodes are more than 100.

The experiment aims to find the different results after various episode to find meaningful duration improvements.

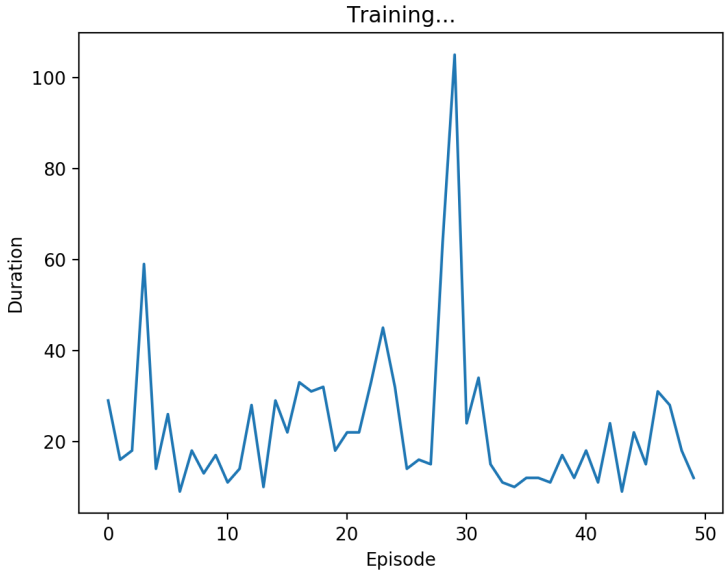

Figure 1: training with 50 episodes

As the figure above, it is difficult to find out any improvement since it peaks at about 30 episode and it does not influence later performance. Given this situation, more episode experiments are taken into consideration.

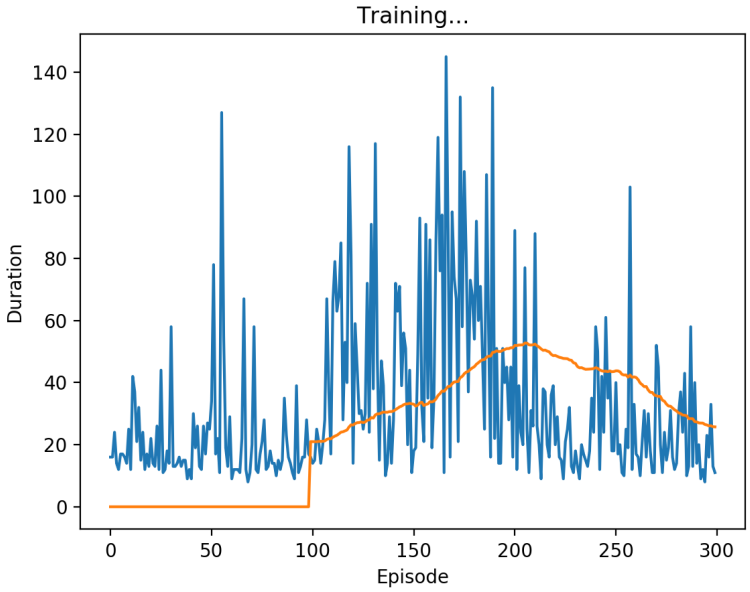

Figure 2: training with 300 episodes

From figure 2 we can easily find that the model improves the performance after 100 episodes. However, after 200 episode there is a down trend, to check this problem, we conducted a new experiment, which took about 5 hours to run the model.

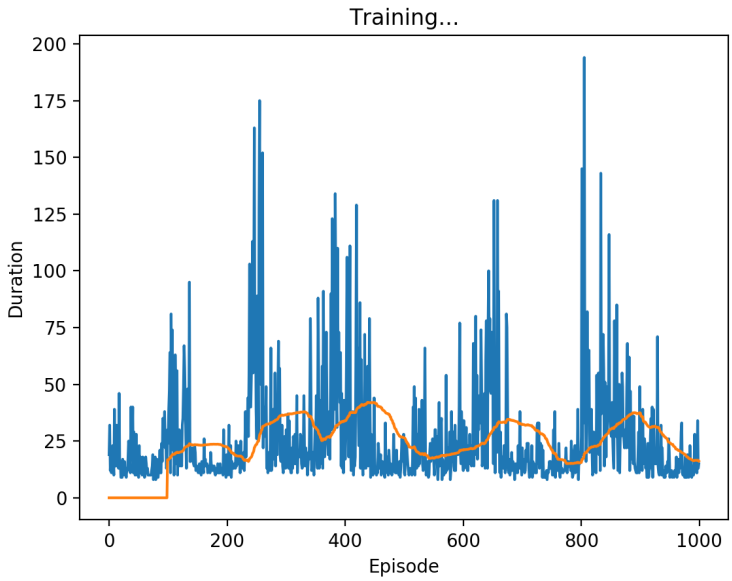

Figure 3: final experiment with 1000 episodes

The result shows that it fluctuates and the model is not stable enough as the highest duration reached at 200, but after the peak it sharply reduced.

For the future work, the model should be improved to find out what is the problem with the fluctuation and how to make it stable in a rational duration.