Deep Reinforcement Learning Homework#3

112062574 郭士平

Method: PPO-based

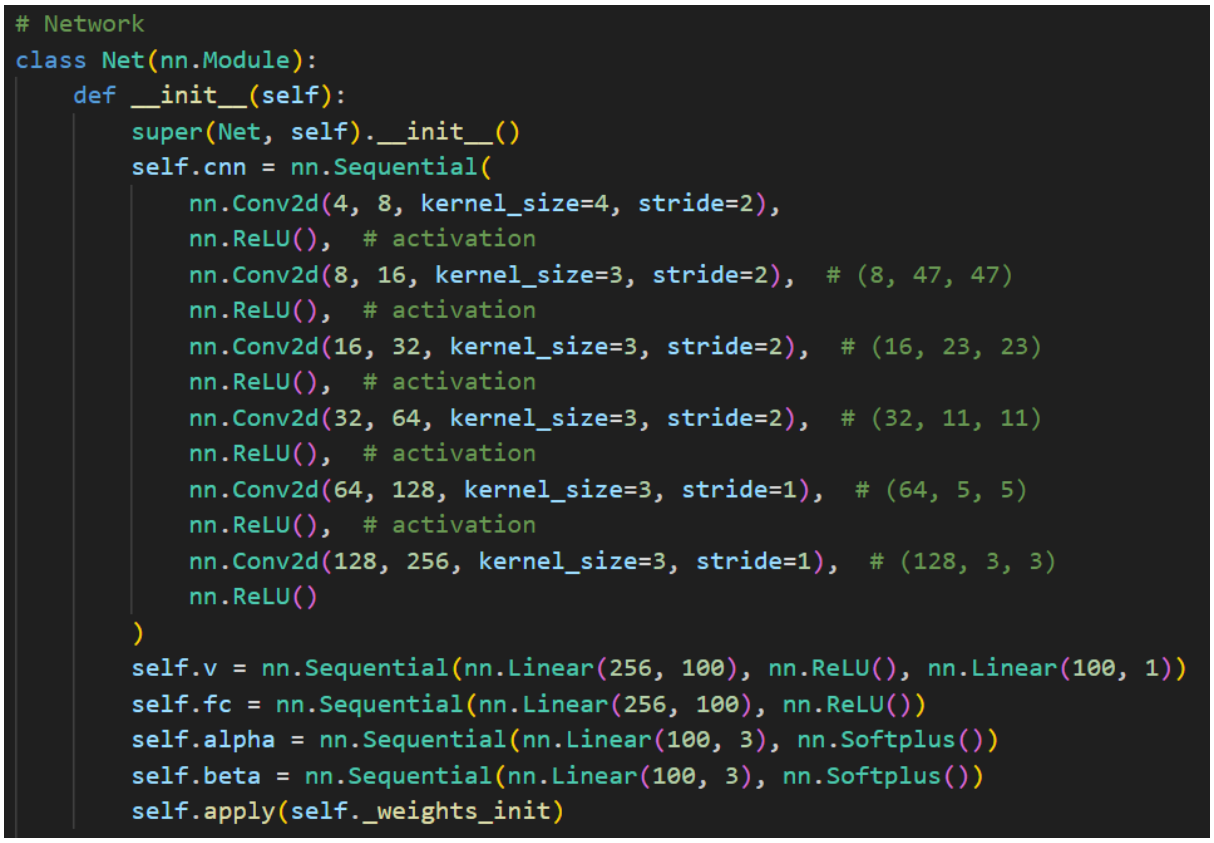
The following implementation is based on Proximal Policy Optimization (PPO), with 3510 training episodes (i.e. iterations). The actor played the whole game to collect minibatch. Once the minibatch is full, update the actor and value network.

Algorithm 1 PPO, Actor-Critic Style

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

Network

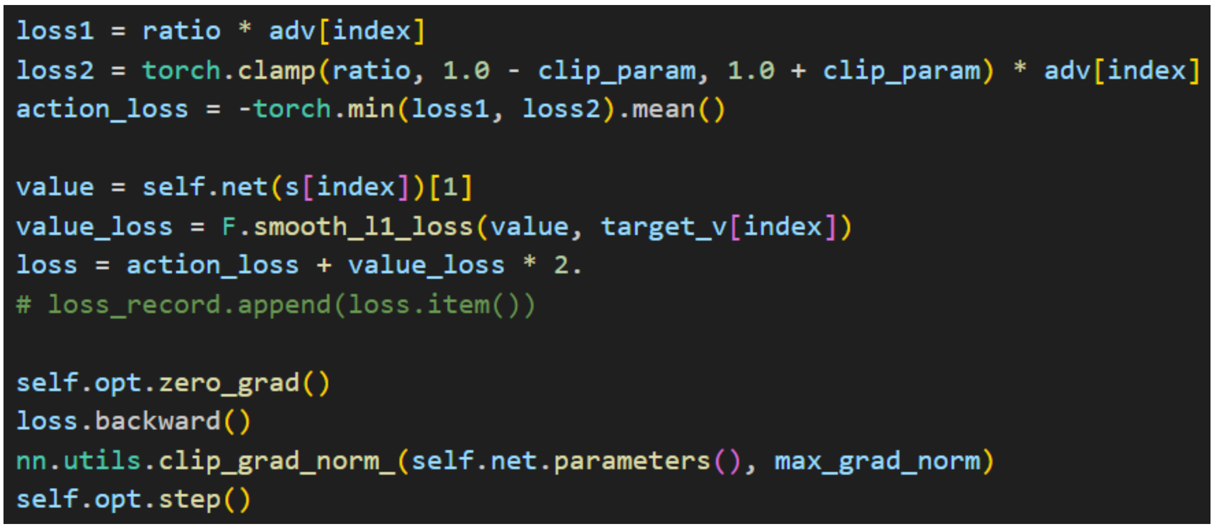
Compared to common PPO, a single network structure is used to represent both actor and critic. After several convolution layers, two heads of fully connected layers (`self.fc` and `self.v`) output the action and value respectively. The alpha and beta are for Beta distribution sampling.



```python
# Network
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.cnn = nn.Sequential(
            nn.Conv2d(4, 8, kernel_size=4, stride=2),
            nn.ReLU(),  # activation
            nn.Conv2d(8, 16, kernel_size=3, stride=2),  # (8, 47, 47)
            nn.ReLU(),  # activation
            nn.Conv2d(16, 32, kernel_size=3, stride=2),  # (16, 23, 23)
            nn.ReLU(),  # activation
            nn.Conv2d(32, 64, kernel_size=3, stride=2),  # (32, 11, 11)
            nn.ReLU(),  # activation
            nn.Conv2d(64, 128, kernel_size=3, stride=1),  # (64, 5, 5)
            nn.ReLU(),  # activation
            nn.Conv2d(128, 256, kernel_size=3, stride=1),  # (128, 3, 3)
            nn.ReLU()
        )
        self.v = nn.Sequential(nn.Linear(256, 100), nn.ReLU(), nn.Linear(100, 1))
        self.fc = nn.Sequential(nn.Linear(256, 100), nn.ReLU())
        self.alpha = nn.Sequential(nn.Linear(100, 3), nn.Softplus())
        self.beta = nn.Sequential(nn.Linear(100, 3), nn.Softplus())
        self.apply(self._weights_init)
```

$$L_{total} = L_{actor} + 2L_{critic}$$

Hence, the loss is summed up by the actor loss and the critic loss. F1 smooth loss is applied to update the whole network.

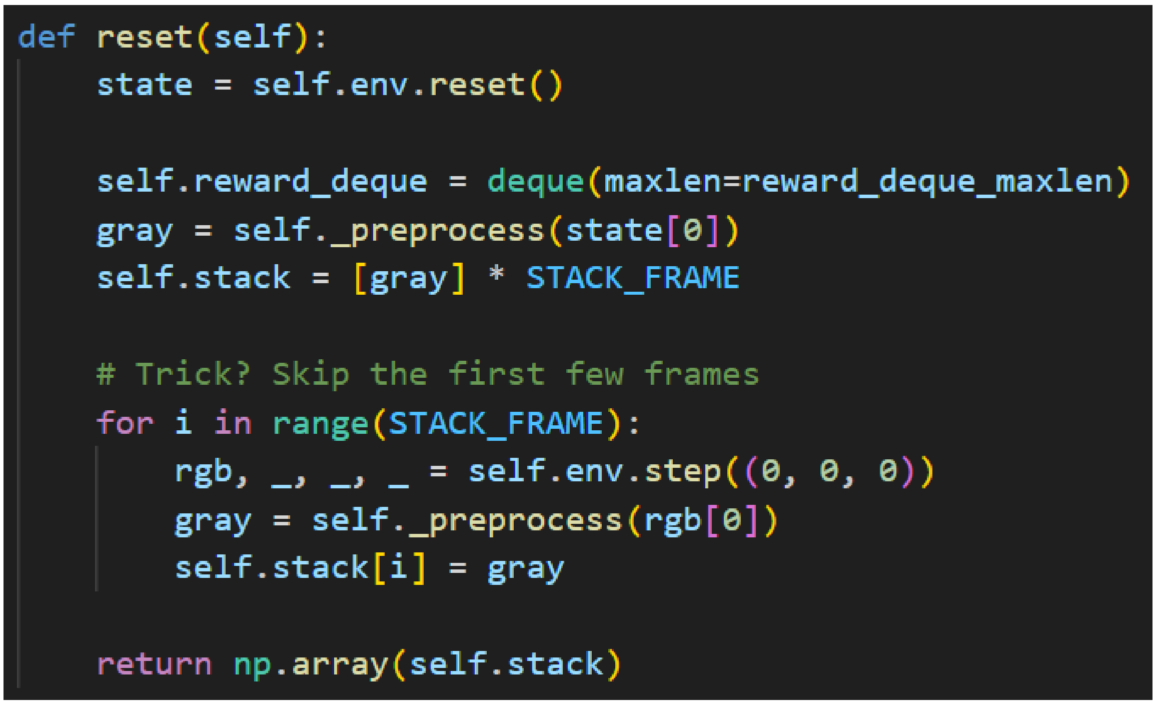


```python
loss1 = ratio * adv[index]
loss2 = torch.clamp(ratio, 1.0 - clip_param, 1.0 + clip_param) * adv[index]
action_loss = -torch.min(loss1, loss2).mean()

value = self.net(s[index])[1]
value_loss = F.smooth_l1_loss(value, target_v[index])
loss = action_loss + value_loss * 2.
# loss_record.append(loss.item())

self.opt.zero_grad()
loss.backward()
nn.utils.clip_grad_norm_(self.net.parameters(), max_grad_norm)
self.opt.step()
```

Preprocessing

The input state of the network is processed by following steps:

1. Gray scale: Convert the input frame from RGB to gray scale
2. Normalize: For every pixel in the gray-scale image, the pixel value is between -1 and 1
3. Skip frame: Every 4 steps choose the exact same action
4. Stack frame: 4 frames are stacked as the input state



```python
def reset(self):
    state = self.env.reset()

    self.reward_deque = deque(maxlen=reward_deque_maxlen)
    gray = self._preprocess(state[0])
    self.stack = [gray] * STACK_FRAME

    # Trick? Skip the first few frames
    for i in range(STACK_FRAME):
        rgb, _, _, _ = self.env.step((0, 0, 0))
        gray = self._preprocess(rgb[0])
        self.stack[i] = gray

    return np.array(self.stack)
```

Other consideration during training:

1. Skip first few frames:

   First few frames rotate and demonstrate the whole map. It may be confusing for the agent. Hence, I decided to skip those frames though I'm not sure the decision is leading the better result.

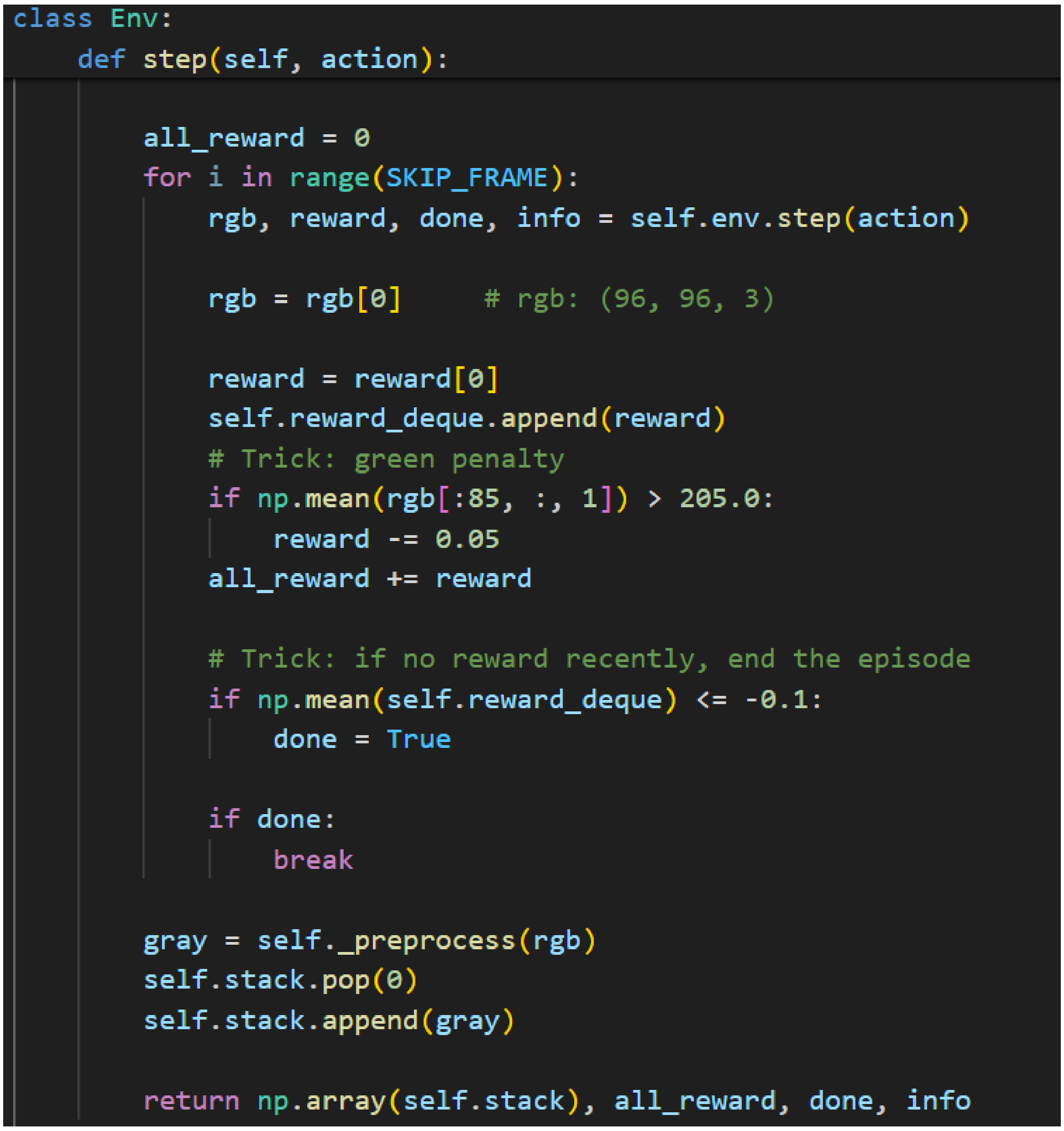
2. Green penalty:

   When the car is out of track, add some negative reward. If the mean of the green channel of a cropped frame exceeds a certain threshold, we consider the car out of track.

3. Early stop:

   If the recent T steps have no positive reward, we stop the episode beforehand. Set T = 100 to reduce training time.

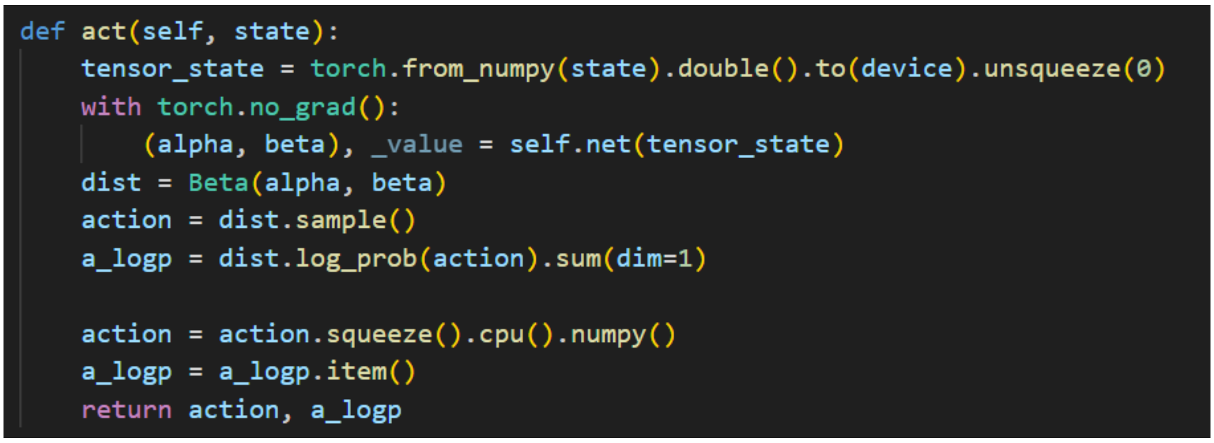
Refer to the following code snippet for better understanding the frame skipping and the stack frame part. It also shows the detailed implementation about green penalty and early stop. The green penalty constants are tested several times, and it fits most cases.



```python
class Env:
    def step(self, action):

        all_reward = 0
        for i in range(SKIP_FRAME):
            rgb, reward, done, info = self.env.step(action)

            rgb = rgb[0]     # rgb: (96, 96, 3)

            reward = reward[0]
            self.reward_deque.append(reward)
            # Trick: green penalty
            if np.mean(rgb[:85, :, 1]) > 205.0:
                reward -= 0.05
            all_reward += reward

            # Trick: if no reward recently, end the episode
            if np.mean(self.reward_deque) <= -0.1:
                done = True

            if done:
                break

        gray = self._preprocess(rgb)
        self.stack.pop(0)
        self.stack.append(gray)

        return np.array(self.stack), all_reward, done, info
```

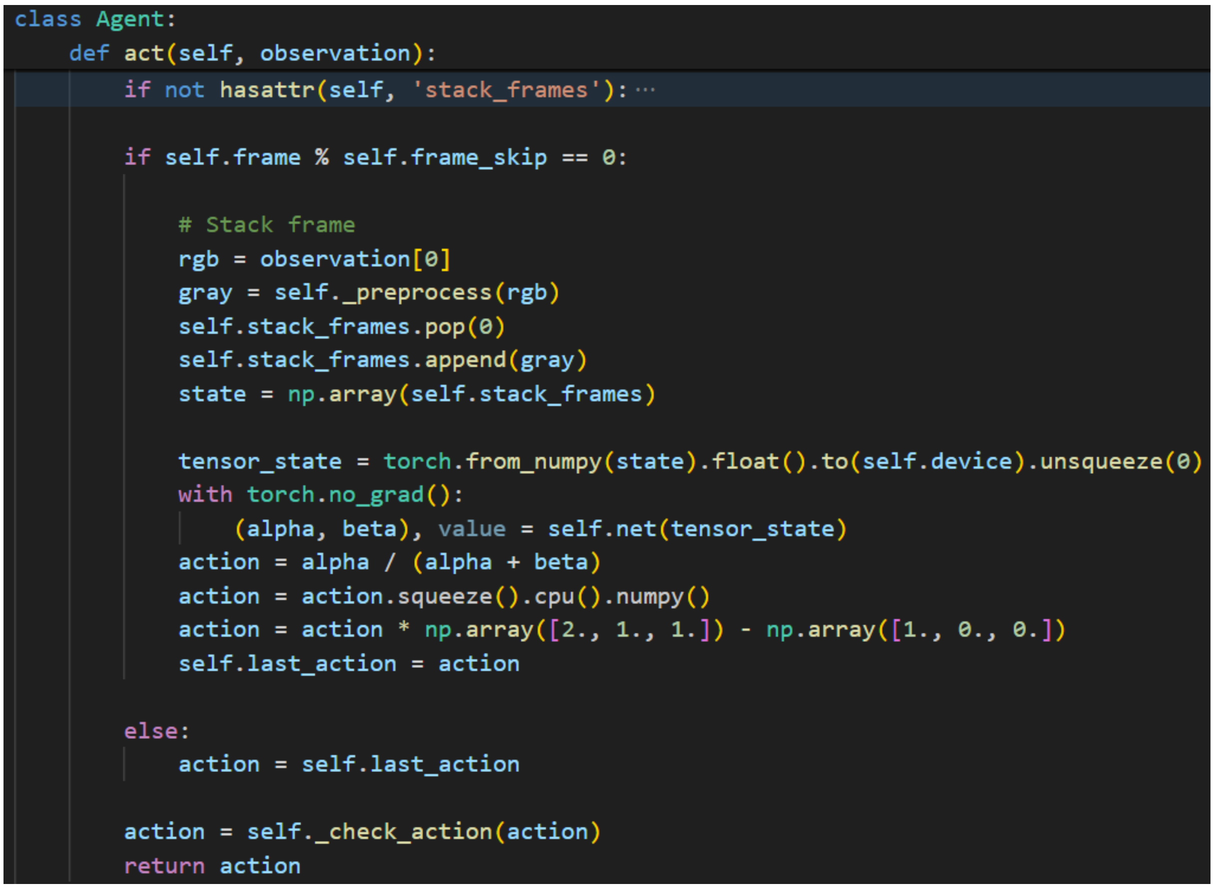
PPO Agent

The PPO agent would use the main network and Beta distribution to sample the action. The following screenshot is from `train.py` .



```python
def act(self, state):
    tensor_state = torch.from_numpy(state).double().to(device).unsqueeze(0)
    with torch.no_grad():
        (alpha, beta), _value = self.net(tensor_state)
    dist = Beta(alpha, beta)
    action = dist.sample()
    a_logp = dist.log_prob(action).sum(dim=1)

    action = action.squeeze().cpu().numpy()
    a_logp = a_logp.item()
    return action, a_logp
```

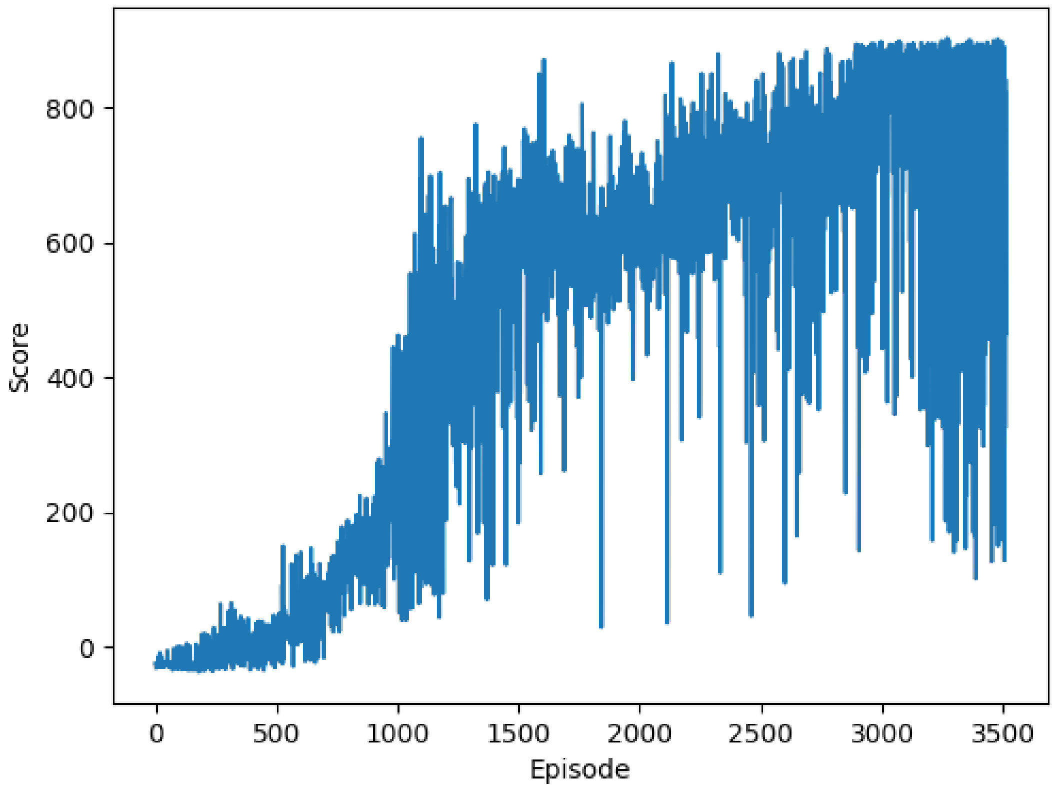
During test time, the agent will preprocess the frame as we do during training. After all the preprocessing, the agent uses stacked frames to choose a proper action.



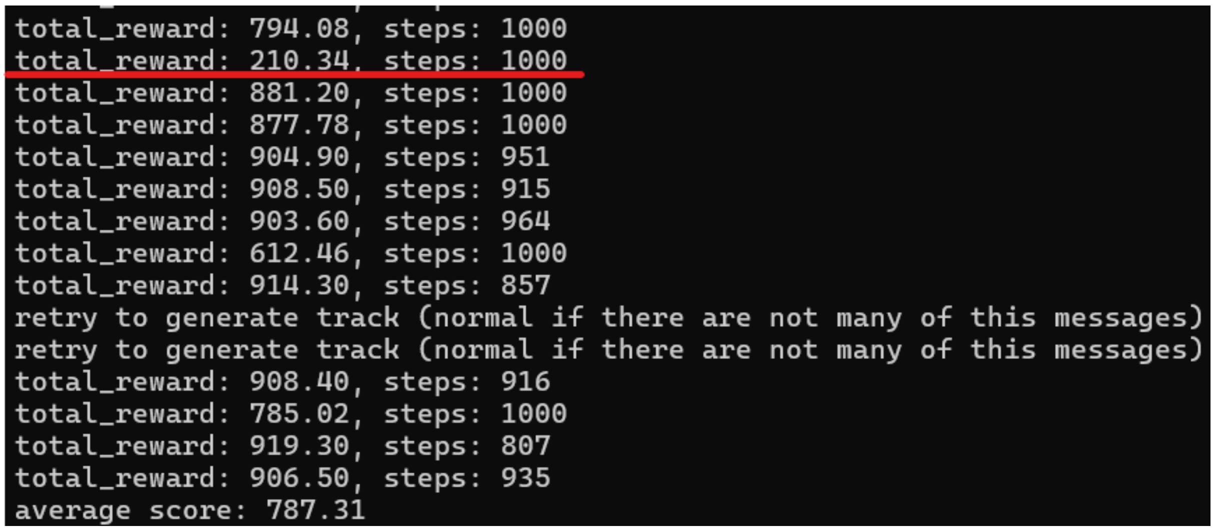
```python
class Agent:
    def act(self, observation):
        if not hasattr(self, 'stack_frames'):···

        if self.frame % self.frame_skip == 0:

            # Stack frame
            rgb = observation[0]
            gray = self._preprocess(rgb)
            self.stack_frames.pop(0)
            self.stack_frames.append(gray)
            state = np.array(self.stack_frames)

            tensor_state = torch.from_numpy(state).float().to(self.device).unsqueeze(0)
            with torch.no_grad():
                (alpha, beta), value = self.net(tensor_state)
            action = alpha / (alpha + beta)
            action = action.squeeze().cpu().numpy()
            action = action * np.array([2., 1., 1.]) - np.array([1., 0., 0.])
            self.last_action = action

        else:
            action = self.last_action

        action = self._check_action(action)
        return action
```

Result

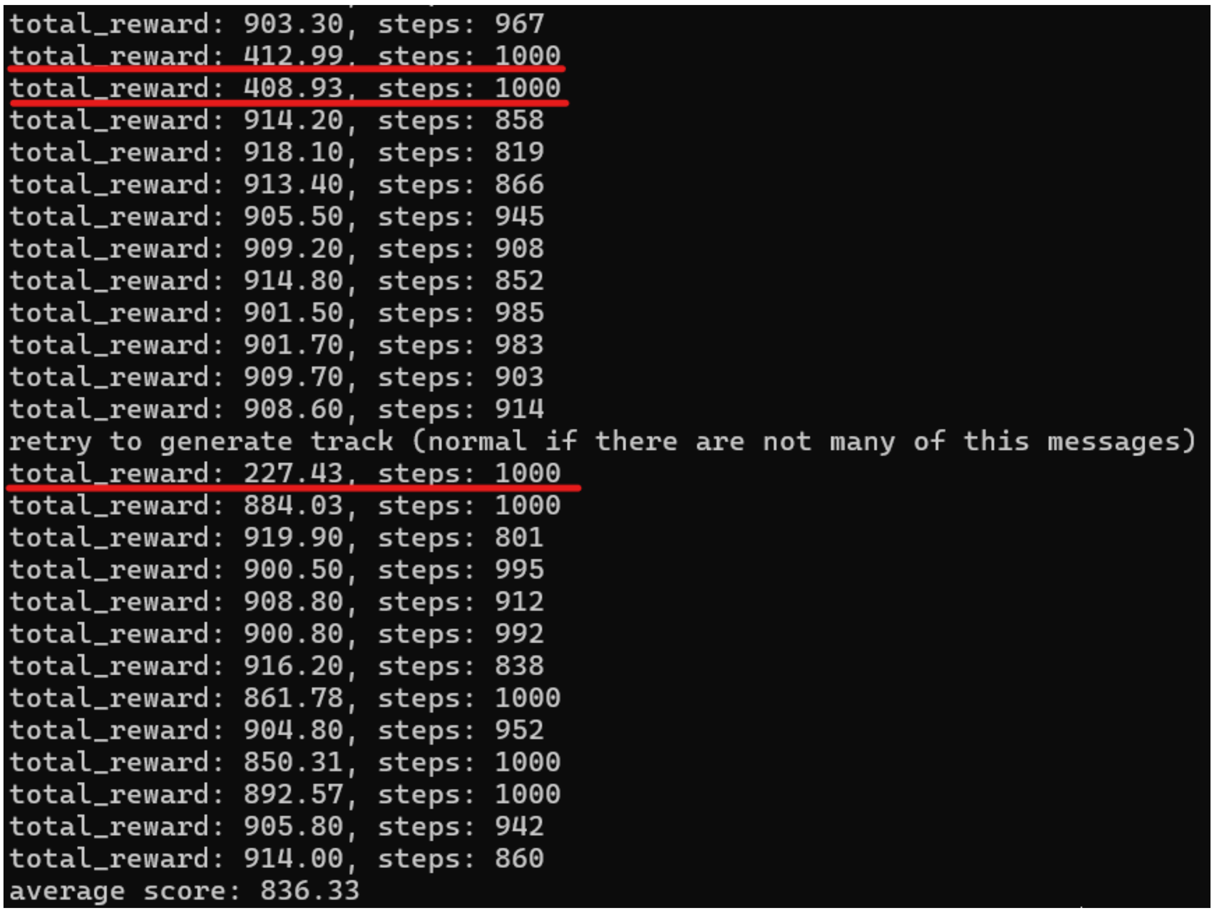
It seems that the limitation of the environment is around 800 900. The score wasn't getting better, yet the variance of score seemed to increase. Therefore, I decided to stop training at episode 3510. The training costs about 12 hours on CPU.



After testing the trained model, the PPO agent performed well on the majority of cases. The only drawback was that the PPO agent appeared to lack the ability to control the vehicle when it accidentally slid onto the grass. Specifically, the agent has yet to learn how to control the sliding car. As a result, a sudden drop in score occurs sometimes.



```
total_reward: 794.08, steps: 1000
total_reward: 210.34, steps: 1000
total_reward: 881.20, steps: 1000
total_reward: 877.78, steps: 1000
total_reward: 904.90, steps: 951
total_reward: 908.50, steps: 915
total_reward: 903.60, steps: 964
total_reward: 612.46, steps: 1000
total_reward: 914.30, steps: 857
retry to generate track (normal if there are not many of this messages)
retry to generate track (normal if there are not many of this messages)
total_reward: 908.40, steps: 916
total_reward: 785.02, steps: 1000
total_reward: 919.30, steps: 807
total_reward: 906.50, steps: 935
average score: 787.31
```



```
total_reward: 903.30, steps: 967
total_reward: 412.99, steps: 1000
total_reward: 408.93, steps: 1000
total_reward: 914.20, steps: 858
total_reward: 918.10, steps: 819
total_reward: 913.40, steps: 866
total_reward: 905.50, steps: 945
total_reward: 909.20, steps: 908
total_reward: 914.80, steps: 852
total_reward: 901.50, steps: 985
total_reward: 901.70, steps: 983
total_reward: 909.70, steps: 903
total_reward: 908.60, steps: 914
retry to generate track (normal if there are not many of this messages)
total_reward: 227.43, steps: 1000
total_reward: 884.03, steps: 1000
total_reward: 919.90, steps: 801
total_reward: 900.50, steps: 995
total_reward: 908.80, steps: 912
total_reward: 900.80, steps: 992
total_reward: 916.20, steps: 838
total_reward: 861.78, steps: 1000
total_reward: 904.80, steps: 952
total_reward: 850.31, steps: 1000
total_reward: 892.57, steps: 1000
total_reward: 905.80, steps: 942
total_reward: 914.00, steps: 860
average score: 836.33
```

After testing 50 episodes and calculating the average score, it was discovered that the average score is impacted heavily by the out-of-control situation.

Reference

[1] Proximal Policy Optimization Algorithms https://arxiv.org/pdf/1707.06347.pdf

[2] https://www.youtube.com/watch?v=hlv79rcHws0&t=1000s

[3] https://github.com/philtabor/Youtube-Code-Repository/tree/master/ReinforcementLearning/PolicyGradient/PPO/torch

[4] https://github.com/xtma