Coac / **never-give-up** Public

<> **Code** Issues 1 Pull requests Actions Projects Security ···

main ▾ ···

never-give-up / **train.py** / <> Jump to ▾

Coac Add working config **History**

1 contributor

161 lines (124 sloc) 5.11 KB ···

```python
import numpy as np
import torch
import torch.optim as optim
import wandb
from gym import Wrapper
from gym_maze.envs.maze_env import MazeEnvSample5x5

from config import config
from embedding_model import EmbeddingModel, compute_intrinsic_reward
from memory import Memory, LocalBuffer
from model import R2D2
```

x_t in paper

from R2D2 code, produced by LSTM → handles POMDP

```python
def get_action(state, target_net, epsilon, env, hidden):
    action, hidden = target_net.get_action(state, hidden)

    if np.random.rand() <= epsilon:
        return env.action_space.sample(), hidden
    else:
        return action, hidden
```

→ ε- greedy

```python
def update_target_model(online_net, target_net):
    target_net.load_state_dict(online_net.state_dict())
```

← straight copy of online network weights

```python
class Maze(Wrapper):
    def step(self, action: int):
        obs, rew, done, info = super().step(["N", "E", "S", "W"][action])
        self.set.add((obs[0], obs[1]))
        if rew > 0:
            rew = 10
```

Move the agent

```
        return obs / 10, rew, done, info
```

termination flag

state

reward

```
    def reset(self):
        self.set = set()
        return super().reset()


def main():
    env = Maze(MazeEnvSample5x5())

    torch.manual_seed(config.random_seed)
    env.seed(config.random_seed)
    np.random.seed(config.random_seed)
    env.action_space.seed(config.random_seed)
```

→ Note for reproducibility
Set random seeds within every class

```
    wandb.init(project="ngu-maze", config=config.__dict__)

    num_inputs = env.observation_space.shape[0]
    num_actions = env.action_space.n
    print("state size:", num_inputs)
    print("action size:", num_actions)
```

see model.py

```
    online_net = R2D2(num_inputs, num_actions)
    target_net = R2D2(num_inputs, num_actions)
    update_target_model(online_net, target_net)
```

Same weights at the start

```
    embedding_model = EmbeddingModel(obs_size=num_inputs, num_outputs=num_actions)
    embedding_loss = 0
```

embedding-model.py → see left side of fig 1 in paper

```
    optimizer = optim.Adam(online_net.parameters(), lr=config.lr)

    online_net.to(config.device)
    target_net.to(config.device)
    online_net.train()
    target_net.train()
    memory = Memory(config.replay_memory_capacity)
    epsilon = 1.0
    steps = 0
    loss = 0
    local_buffer = LocalBuffer()
    sum_reward = 0
    sum_augmented_reward = 0
    sum_obs_set = 0
```

number of episodes

```
    for episode in range(30000):
        done = False
        state = env.reset()
        state = torch.Tensor(state).to(config.device)

        hidden = (
            torch.Tensor().new_zeros(1, 1, config.hidden_size),
            torch.Tensor().new_zeros(1, 1, config.hidden_size),
        )

        episodic_memory = [embedding_model.embedding(state)]

        episode_steps = 0
        horizon = 100
        while not done:
            steps += 1
            episode_steps += 1

            action, new_hidden = get_action(state, target_net, epsilon, env, hidden)

            next_state, env_reward, done, _ = env.step(action)
            next_state = torch.Tensor(next_state)

            augmented_reward = env_reward
            if config.enable_ngu:
                next_state_emb = embedding_model.embedding(next_state)
                intrinsic_reward = compute_intrinsic_reward(episodic_memory, next_st
                episodic_memory.append(next_state_emb)
                beta = 0.0001
                augmented_reward = env_reward + beta * intrinsic_reward

            mask = 0 if done else 1

            local_buffer.push(state, next_state, action, augmented_reward, mask, hid
            hidden = new_hidden
```

replace

running in NGU mode

see embedding-model.py

```
            if len(local_buffer.memory) == config.local_mini_batch:
                batch, lengths = local_buffer.sample()
                td_error = R2D2.get_td_error(online_net, target_net, batch, lengths)
                memory.push(td_error, batch, lengths)

            sum_reward += env_reward
            state = next_state
            sum_augmented_reward += augmented_reward

            if steps > config.initial_exploration and len(memory) > config.batch_siz
                epsilon -= config.epsilon_decay
                epsilon = max(epsilon, 0.4)

                batch, indexes, lengths = memory.sample(config.batch_size)
                loss, td_error = R2D2.train_model(online_net, target_net, optimizer,
```

RL training

```
                if config.enable_ngu:
                    embedding_loss = embedding_model.train_model(batch)
```

training embedding

```
                memory.update_priority(indexes, td_error, lengths)

                if steps % config.update_target == 0:
                    update_target_model(online_net, target_net)

```

slow update after fixed number of updates of online model

```
            if episode_steps >= horizon or done:
                sum_obs_set += len(env.set)
                break

        if episode > 0 and episode % config.log_interval == 0:
            mean_reward = sum_reward / config.log_interval
            mean_augmented_reward = sum_augmented_reward / config.log_interval
            metrics = {
                "episode": episode,
                "mean_reward": mean_reward,
                "epsilon": epsilon,
                "embedding_loss": embedding_loss,
                "loss": loss,
                "mean_augmented_reward": mean_augmented_reward,
                "steps": steps,
                "sum_obs_set": sum_obs_set / config.log_interval,
            }
            print(metrics)
            wandb.log(metrics)

            sum_reward = 0
            sum_augmented_reward = 0
            sum_obs_set = 0


if __name__ == "__main__":
    main()
```

useful for tracking if you haven't tried it