Deep Reinforcement Learning Algorithms for Text based games

Prateek Chaudhry
pc26978

Shivam Garg
sg48957

Abstract

Text based games are interactive sequential games, in which a player and a game simulator interact through natural language, and the player has to achieve a goal by manipulating the game state through natural language commands. Inspired by previous work, we model these games as Partially Observable MDPs (POMDPs). We further implement various deep reinforcement learning algorithms and compare their convergence and performance by training on 100 different text games.

1 Introduction

Text based games are sequential games, in which a player is provided an environment description and a goal in natural language. The player has to reach the goal by interacting with the environment through actions, which are natural language commands. Each command potentially changes the environment, and the player receives a feedback for the action performed, depending on which further actions are to be performed. If the goal is reached within a fixed number of steps, the player wins the game.

We use TextWorld (Côté et al., 2018) as the framework for modeling games. In particular, we choose a class of problems defined in the official github of TextWorld. The environment of the problems is a description of a house, and the goal is to retrieve a hidden item, and put it on the stove. The player has an inventory of objects retreived. The player is also provided with a list of admissible commands, which are the commands the game interpreter can understand, and a list of possible entities with which the player can interact with. Table 1 depicts a typical example of an episode of such a game.

Obs	You are in a bedroom. A standard kind of place. You can make out a chest drawer. There's something strange about this being here ... There is a closed wooden door leading east.
Cmd	open chest drawer
Obs	You open the chest drawer, revealing an old key. Your score has just gone up by one point.
Cmd	take old key from chest drawer
Obs	You take the old key from the chest drawer. Your score has just gone up by one point.
Cmd	unlock wooden door with old key
Obs	...
Cmd	...

Table 1: Sample episode from a typical game

The text based games as described can be modelled as Partially Observable Markov Decision Processes (POMDPs) in the following way:

- **Observations:** Upon each interaction with the environment, the agent receives an observation in the form of a natural language string. We also have an initial observation string which describes the initial environment state and the target goal.
- **Actions:** At each step, the TextWorld simulator provides the agent with a list of admissible actions, from which it can choose an action.
- **States:** Current observation does not fully describe the state in these games, since the initial description and previous observations may contain information relevant to our current step. Therefore, we model the problem

as a POMDP, and use a function over the entire history of observations and actions in the running episode as a representation of the current state. It can be represented in dense vector form using recurrent neural network over the history.

- **Rewards:** We get intermediate rewards from the simulator to keep the rewards dense. The agent gets a reward of -1 if last action needs to be undone before resuming the quest, a reward of 0 if last action didn't affect the quest and a reward of +1 if last action brought us closer to completing the quest.

- **Goal:** The goal of the game is to put an item mentioned in the initial description on the stove in the kitchen. When this is achieved, the episode ends.

In the baseline code available on github, the above problem is solved by modeling the policy using RNN and training using REINFORCE with baseline. We explore other more advanced policy gradient methods such as PPO (Schulman et al., 2017) and SAC (Haarnoja et al., 2018), as well as value based methods such as DRQN (Hausknecht and Stone, 2015), and see how they compare with each other on this particular problem setting.

2 Literature Survey

2.1 Text-based RL agents

Narasimhan et al. (2015) first tried to solve text based games using reinforcement learning. They adapted deep Q-learning framework to the problem by their model called LSTM-DQN. The model encodes observations at each step by LSTM and then takes average of encodings at all steps to represent state. Actions are modeled as two worded outputs composed of verb followed by a word. They model Q value for both the words and take average for representing total Q value of an action. This is then trained by the DQN framework. Their model showed great success on text based games in the environment of Homeworld and Fantasyworld.

Text-World (Côté et al., 2018) is the simulator released by Microsoft Research which enables experiments with text-based games. A simulation on the platform involves a textual description of the environment and goal state. Each action is a natural language statement which is limited by design to a few specific templates. After each action, an observation about the state of the environment is given by the simulator. Yuan et al. (2019) proposed the task of QAit. They tackled the problem using DQN based approach. Also, they approximate state in POMDP model with current observation, question asked and previous command.

Yuan et al. (2018) showed success in a particular type of text based games which involved navigating through a chain of rooms and finding a coin, without revisiting dead ends. They used reward shaping using number of visits to a room. They solve the problem using DQN, with an LSTM over history of observations to encode the state.

2.2 Value based Methods

Deep Q Networks (Mnih et al., 2015) showed great success in incorporating deep convolutional networks for feature representation in Q learning for Atari video game playing. They stabilized training by storing episodes in an experience buffer and reusing it for training by uniformly sampling from it. They also used different networks for target Q values and trainable Q values, which proved to be essential for stable training. After some training epochs, the trained Q network weights are copied to target Q network, new episodes are sampled, and training is continued. This was the first time neural networks showed success with Q learning.

2.3 Policy Gradient Methods

Proximal Policy Optimisation(PPO) (Schulman et al., 2017) is a variant of the policy gradient algorithms in which the gradient update is clipped to handle the variance of off-policy importance smapling ratio. The importance sampling ratio is clipped to 1-ϵ, 1+ϵ, to avoid very large gradients updates. The algorithm outperformed existing policy gradient algorithms on Atari and provided overall better performance metrics.

2.4 Deep RL for POMDP

Hausknecht and Stone (2015) propose using recurrent networks to handle POMDPs. Agents in partially obseravble environment do not observe the whole current state and thus need to remember previous states encountered for better performance. The DQN network was modified by adding a LSTM unit in the last layer of the network so that it can help in aggregation of relevant context.

Igl et al. (2018) propose using variational autoencoders to learn a representation of the enviroment. They also describe a new loss function based on evidence based lower bound to jointly train the policy and the variational autoencoder.

3 Approaches

We describe the neural network architecture we use for modeling policy followed by details of different approaches we tried.

3.1 Network Architecture

We model the policy using Gated Recurrent Units (GRUs) (Cho et al., 2014), which are better than simple RNNs for training over longer sequences. First, we tokenize and embed current observation using randomly initialized word embeddings. These embeddings are then passed through a GRU to obtain encoding vector of the current observation. Another GRU takes these encoding vectors at each timestep as input. This higher level GRU acts as a function over history of all observation encodings in current episode, and is used to obtain an encoding vector, which acts as the state representation at current time step. All commands in the admissible command lists are also encoded using GRU, just like observation. Then, the state representation vector obtained from the higher level GRU is used to attend over the admissible command list to obtain an attention distribution. This attention distribution is used as the policy from which actions are sampled.

3.2 REINFORCE - baseline

For REINFORCE, we use the official implementation provided with the TextWorld framework in their github. REINFORCE aims to maximize the following function:

$$J(\theta) = \sum_t log\ \pi_\theta(a_t|s_t) * A_t$$

where A_t is the advantage function. For defining value estimate for calculating A_t, a linear projection from state representation from GRU is learnt.

3.3 Proximal Policy Optimization

We implemented the Proximal policy optimization (Schulman et al., 2017) algorithm to learn the policy network. The advantage function was calculated in the similar way as calculated in REINFORCE baseline. The main motivation to use PPO was to study whether stable gradient updates and training paradigm would help in better average reward per episode and training speed. The objective function optimized at update step k is:

$$J(\theta) = min(\frac{\pi_\theta(a_t|s_t)}{\pi_k(a_t|s_t)}, clip(1-\epsilon, 1+\epsilon, \frac{\pi_\theta(a_t|s_t)}{\pi_k(a_t|s_t)}))A_t$$

where A_t is the advantage function, π_k denotes the policy from which the trajectories are sampled using which the updates are done and π_θ denotes the policy with the current policy parameters which are obtained in successive batch SGD updates.

3.4 Soft Actor Critic

We also implement Soft Actor-Critic (SAC) (Haarnoja et al., 2018) to learn the policy via policy gradient methods. The baseline implementation currently uses a large number of samples(1000 episodes) to learn the policy for a simple setting of the environment. We experiment with SAC to analyze whether sample efficient nature of SAC due to it being off-policy in nature is helpful in the current setting. For SAC, we implement four neural networks, one for policy, one for value function and two for Q function to implement double Q-learning setup. SAC is based on maximum entropy reinforcement learning framework, in which apart from the usual reward maximization, entropy of the policy is also maximized. We implement the version of SAC as mentioned in the OpenAI spinning up in RL [1]. SAC in the original paper is given for problems with continous action spaces, but since our problem involves discrete actions, we modified the algorithm to adapt it for discrete action spaces (Christodoulou, 2019). Instead of using actions sampled from the policy to estimate the value network estimate and policy optimization objective, we use the expected value according to the action chosen by the policy. The following two steps in the pseudocode are modified:

1. In step 12 of the pseudo code, we change the calculation of the value estimate from

$$V(s) = \min_{i=1,2} Q_i(s, \widetilde{a}) - \alpha log \pi_\theta(\widetilde{a}|s)$$

$$\widetilde{a} = \pi_\theta(.|s)$$

to

$$V(s) = E_{a=\pi_\theta(.|s)}[\min_{i=1,2} Q_i(s,a) - \alpha log \pi_\theta(a|s)]$$

[1] https://spinningup.openai.com/en/latest/algorithms/sac.html

Algorithm 1 Soft Actor-Critic

Input: initial policy parameters θ, Q-function parameters ϕ_1, ϕ_2, V-function parameters ψ, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\psi_{\text{targ}} \leftarrow \psi$
repeat
  Observe state s and select action $a \sim \pi_\theta(\cdot|s)$
  Execute a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  If s' is terminal, reset environment state.
  if it's time to update **then**
    for j in range(however many updates) **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute targets for Q and V functions:

$$y_q(r, s', d) = r + \gamma(1 - d)V_{\psi_{\text{targ}}}(s')$$
$$y_v(s) = \min_{i=1,2} Q_{\phi_i}(s, \tilde{a}) - \alpha \log \pi_\theta(\tilde{a}|s), \qquad \tilde{a} \sim \pi_\theta(\cdot|s)$$

      Update Q-functions by one step of gradient descent using

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} (Q_{\phi,i}(s, a) - y_q(r, s', d))^2 \qquad \text{for } i = 1, 2$$

      Update V-function by one step of gradient descent using

$$\nabla_\psi \frac{1}{|B|} \sum_{s \in B} (V_\psi(s) - y_v(s))^2$$

      Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} \Big(Q_{\phi,1}(s, \tilde{a}_\theta(s)) - \alpha \log \pi_\theta \left(\tilde{a}_\theta(s)|\, s\right) \Big),$$

      where $\tilde{a}_\theta(s)$ is a sample from $\pi_\theta(\cdot|s)$ which is differentiable wrt θ via the reparametrization trick.
      Update target value network with

$$\psi_{\text{targ}} \leftarrow \rho\psi_{\text{targ}} + (1 - \rho)\psi$$

    end for
  end if
until convergence

Figure 1: Pseudocode for soft actor critic from Spinning up in RL website

2. In step 15, we change the loss function to the expectation of the objective over all actions for the given state instead of using an unbiased estimate which samples from the policy

$$J(\theta) = \sum_{s \in B} E_{a=\pi_\theta(.|s)}(Q_1(s, a) - \alpha log \pi_\theta(a|s))$$

   This objective(J) is maximised using SGD updates.

3.5 Deep Recurrent Q Network

Finally, we also implemented a value based method to evaluate it's performance compared to the policy based methods described above. Since our problem is a POMDP, we decided to implement Deep Recurrent Q Learning (Hausknecht and Stone, 2015).

To model Q values, we modified the policy network described above to output the Q values instead of probabilities. Instead of converting the final attention scores obtained into a distribution using softmax, we straightaway use the attention scores as the Q values of the different actions.

We sample n episodes from the ϵ-Greedy policy according to current Q network and put them in a replay buffer. Then we iterate over the buffer multiple times , accumulate the following loss for each step of an episode, and do single optimization step for entire episode:

$$L(\theta) = (r_t + \gamma * Q_{\overline{\theta}}(s_{t+1}, a_{t+1}) - Q_\theta(s_t, a_t))^2$$

Here, $\overline{\theta}$ indicates a fixed set of weights using which the target Q values are calculated. After the buffer is used to update the network p times, $\overline{\theta}$ is updated to the new values of θ.

4 Results and Analysis

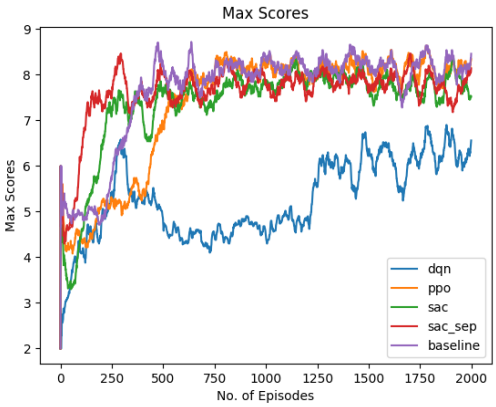

Figure 2: Max scores during training

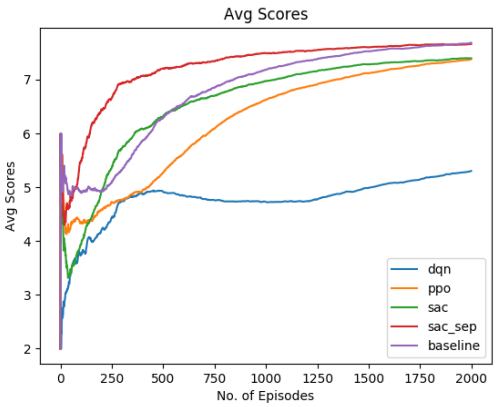

Figure 3: Average scores while training

We focus on five experiments, one for baseline, one for DQN , one for PPO and two for SAC implementation. We evaluate the performance in three ways:

1. Max Scores: While training, for each episode we calculate the maximum score achieved by the agent duirng the episode.

2. Avg Scores: The average score obtained by agent during training. This metric is the aver-

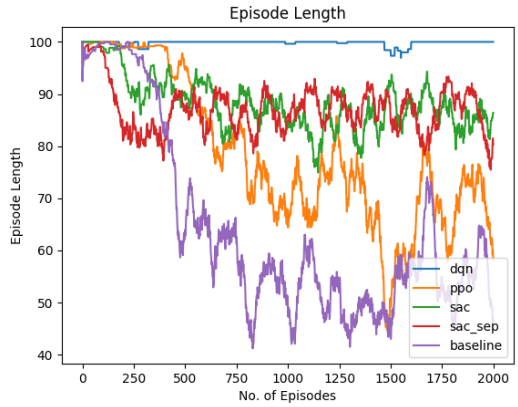

Figure 4: Length of episodes while training

age of the final scores achieved by the agent during training on each episode.

3. Episode Length: The steps taken by the agent to complete the game.

Different approaches are trained in a similar way with all the controllable parameters kept similar to ensure fair comparison. Some details common to all approaches are:-

- The base network for all the approaches is similar with same hyperparameters, except for one experiment of SAC implementation.
- All experiments are run for 2000 episodes.
- Each update step of the respective algorithm involves computing the loss on a single episode in the replay buffer.
- Adam optimizer is used to train all algorithms.

The algorithm specific details are here under:-

- Reinforce-Baseline: The baseline implementation updatea the network on every 10 steps of the episode. We change this to update after every episode.
- PPO: The training paradigm is similar to Reinforce . The difference here is four episodes are sampled and the PPO loss is optimized on this batch of episodes for two epochs.
- DQN: The replay buffer usage and updates is similar to PPO except for the fact that we use Bellman error as the optimization objective to minimize. We sample episodes using ϵ-Greedy policy with $\epsilon = 0.2$.
- SAC: We run two experiments with different neural network acrhitectures. The first one is similar to the baseline, with just two FC layers being added over the baseline network to model Q function for double Q-learning. This network has very little extra computation added. This experiment is referred to as sac in the graphs. Second experiment uses separate networks for Q function approximation. The architecture of base models for separate Q networks is exactly similar to the one modeling the policy. This approach is referred to as sac_sep in the graphs.

Our findings from the experiments are as follows:-

- The policy gradient methods perform equivalently, with each achieving a score of around 8.5 when trained for 2000 episodes. SAC with separated networks(sac_sep) learns quickly and reaches the peak in about 300 episodes as shown in the red curve in Figure 2. SAC converges in about 600 episodes. PPO and reinforce although learn slowly, end up being a bit higher in the final leg of training with purple and orange lines in Figure 2 describing the behavior.
- SAC with separated networks has the highest average score which describes that SAC is able to learn from less number of samples as visible in Figure 3. SAC which shares the model backbone with the policy network performs not so well with it lagging consistently behind the separated version. We suspect that since the backbone is being affected by gradient updates for policy and value loss as well, the Q networks there are not able to provide a stable estimate of the state-action value.
- The REINFORCE-baseline approaches the SAC separated version in the limit of 2000 episodes. The PPO algorithm converges slowly, which is expected since the PPO clipping objective stabilises the training instead of making large updates. Although the gap in the average scores of different approaches tends to narrow and may converge when trained for even longer.
- The baseline learns to complete the game in smallest number of steps with average of about 50 steps per episode. PPO follows shortly, with average epsiode length being

around 60 steps. The two versions of SAC have similar episode lengths of around 85 steps per episode. One possible reason for SAC approach to take large number of steps is that the objective in SAC tries to maximise a combination of entropy and score. A better hyperparameter search might allow SAC to perform equivalently. Moreover, the length of the episode should have been taken into account by discount factor in the reward calculation, but maximising the entropy term would have been preferred by the network.

- DQN approach performs poorly in our case. These results surprise us. The following factors may have contributed to the poor performance of DQN:
  - In DQN, actual Q function is learnt by the network through which policy is determined. Learning exact Q values is harder in comparison to learning just the relative value of actions, which the policy gradient methods do.
  - While training, the distribution over the actions in a state changes rapidly. A gradient update can make the value of an action largest and the ε-greedy action selection criterion would now place a disproportionate weight on the new action, thus changing the training dynamics.
  - While training, we observe that the Q value estimates change a lot while training, especially when the episodes are terminated abruptly on reaching max steps. Since the agent sees complete episodes only for a fraction of the training time, the value of intermediate state-action pairs fluctuates making training harder.

We also test the performance of the algorithms on an unseen test set which has 20 games drawn from a different distribution than the one used for training set. The results for those are given in Table 2. The policy gradient algorithms generalize well on the test set. DQN performs porrly in line with it poor performance on the training set. We do not find any gains in the performance by using different policy gradient algorithms when compared to REINFORCE.

Agent	Average Score
Random	4.4
BaseLine	8.8
DQN	6.6
PPO	8.8
SAC	8.8
SAC-SEP	8.8

Table 2: Results on test games

5 Future Work

In the current project, we explored the dense reward setting in which the agent is rewarded at each time step. This is an easier version of the problem. There is a sparse reward setup wherein the reward is received only at the end of the game, i.e. whether you win or lose the game. This is a more challenging problem and more applicable in real life since rewards are not always available in real life settings. One direction that can be explored in the sparse reward setting is to use inverse reinforcement learning (Abbeel and Ng, 2004) to learn a reward function which the expert tends to follow. This reward function could help stable and faster convergence of the policy gradient algorithms.
Another restriction of the problem setting we tackled is that we are provided a limited set of actions at each step. A more challenging task is when the action space is the entire set of natural language strings, in which case we will also have to train a decoder to generate the command. This setting should be much more harder to solve with RL.

6 Conclusion

In this project, we explore the problem of playing text based games via RL-agents. The problem is a trimmed down version of full blown interactive task completion where the number of actions are constrained. The problem has two aspects: the NLP aspect of understanding the natural language observations and commands, and the RL aspect of planning and carrying out various actions. We studied the efficacy of policy gradient algorithms and Deep Q Learning for this problem, analyzing the training stability and performance. We found that the off-policy methods like SAC benefit the problem with faster convergence and sample efficiency. Further, we find that all policy gradient methods converge in the limit, while the value based DQN is relatively difficult to train.

References

Pieter Abbeel and Andrew Y Ng. 2004. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the twenty-first international conference on Machine learning*, page 1. ACM.

Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*.

Petros Christodoulou. 2019. Soft actor-critic for discrete action settings. *arXiv preprint arXiv:1910.07207*.

Marc-Alexandre Côté, Ákos Kádár, Xingdi (Eric) Yuan, Ben Kybartas, Tavian Barnes, Emery Fine, James Moore, Matthew Hausknecht, Layla El Asri, Mahmoud Adada, Wendy Tay, and Adam Trischler. 2018. Textworld: A learning environment for text-based games. In *Computer Games Workshop at ICML/IJCAI 2018*, pages 1–29.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. 2018. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint arXiv:1801.01290*.

Matthew Hausknecht and Peter Stone. 2015. Deep recurrent q-learning for partially observable mdps. In *2015 AAAI Fall Symposium Series*.

Maximilian Igl, Luisa Zintgraf, Tuan Anh Le, Frank Wood, and Shimon Whiteson. 2018. Deep variational reinforcement learning for pomdps. *arXiv preprint arXiv:1806.02426*.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529.

Karthik Narasimhan, Tejas Kulkarni, and Regina Barzilay. 2015. Language understanding for text-based games using deep reinforcement learning. *arXiv preprint arXiv:1506.08941*.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Xingdi Yuan, Marc-Alexandre Cote, Jie Fu, Zhouhan Lin, Christopher Pal, Yoshua Bengio, and Adam Trischler. 2019. Interactive language learning by question answering. *arXiv preprint arXiv:1908.10909*.

Xingdi Yuan, Marc-Alexandre Côté, Alessandro Sordoni, Romain Laroche, Remi Tachet des Combes, Matthew Hausknecht, and Adam Trischler. 2018. Counting to explore and generalize in text-based games. *arXiv preprint arXiv:1806.11525*.