Text Game Agent Based on Reinforcement Learning and Natural Language Processing

Yiqing Huang[*]
51846926, Undergraduate, SIST
ShanghaiTech University
Email: huangyq@shanghaitech.edu.cn

Ruyue Hong
86530765, Undergraduate, SIST
ShanghaiTech University
Email: hongry@shanghaitech.edu.cn

Yilin Guo
72517706, Undergraduate, SIST
ShanghaiTech University
Email: guoyl@shanghaitech.edu.cn

Hongbin Ye
94770504, Undergraduate, SIST
ShanghaiTech University
Email: yehb@shanghaitech.edu.cn

ABSTRACT

Text-based games are promising platforms for improving the ability to learn optimal control policies in systems where action space is defined by sentences in natural language. Text-game based agent by reinforcement learning is a relatively cutting-edge field with development space. Mikulas Zelinka et al. proposed an algorithm SSAQN [1], which do text preprocessing first, then construct and train the neural network architecture. On the basis of SSAQN, we create three different models by adjusting the neural network architecture and training architecture. For neural network architecture, we refer to the approach proposed by Ji He et al. in [2] ; for training architecture, we use deep q-learning [3] and double DQN [4]. Finally, we test and analyze these three different models.

Keywords: *Natural Language Processing, Reinforcement Learning, Neural Networks, Text Games.*

1 Introduction

The research in the *Natural Language Processing (NLP)* is always a significant component of *Artificial Intelligence (AI)*. One of these tasks is to apply *Reinforcement Learning* on real *text-based games*.

In *text games*, which is known as *Interactive Fiction (IF)*, the player is given a description in natural language dialogues and chooses an action to update each state at each time step. The *text game* ends at the final state. Usually, the *text game* is sequential decision making tasks with states and actions included in natural language dialogues.

There are always multiple choices that a player can take, which lead to various endings eventually. Availability of the results is quite fit for *Reinforcement Learning* as well as the final evaluation. Although different algorithms can win the *text game*, we care more about these unseen data behind the model itself. We want our *text game* agents at least understand parts of the natural language dialogues by *Reinforcement Learning*.

2 Text Games

In this section, we introduce the fundamental concepts of *text games* by *Interactive Fiction (IF)*. And we use *pyfiction* [1] as framework to do further research in convenience.

2.1 Definition

Text game is a sequential decision-making task with both input and output spaces given in natural language.

Definition 2.1 (text game). *Define a text game as a graph $G = [H, H_t, S, A, D, T, R]$, where [1]*

- *H is a set of game states*
- *H_t is a set of terminating game states, $H_t \subseteq H$*
- *A is a set of possible action descriptions*
- *D is a function generating text descriptions*
- *$D : H \to (S \times 2^A)$*
- *T is a transition function, $T : (H \times A) \to H$*
- *R is a reward function, $R : S \to \mathbb{R}$*

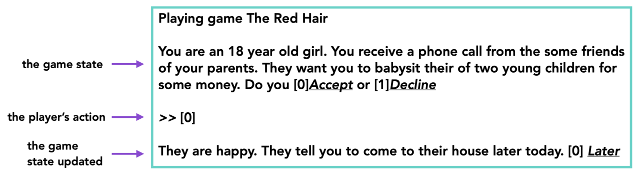
[*] All authors are equally important.

[1] `https://github.com/MikulasZelinka/pyfiction`

The properties of the game and its functions determines the task difficulty. The *text game* is deterministic if the transition function T and the description function D are deterministic. Generally, the *Reinforcement Learning* agents can be applied to other *text games* much more complex. Therefore, we finally use *Reinforcement Learning (RL)* models rather than use graph searching ways.

2.2 Game Structure

Text games usually consists of three parts, the game state, the player's action and the game state updated. The game state updates whenever the player takes an action to the text description in previous game state. Until the final state comes, the game keeps working on. (Fig. 1 [2])



Playing game The Red Hair

You are an 18 year old girl. You receive a phone call from the some friends of your parents. They want you to babysit their of two young children for some money. Do you [0]*Accept* or [1]*Decline*

>> [0]

They are happy. They tell you to come to their house later today. [0] *Later*

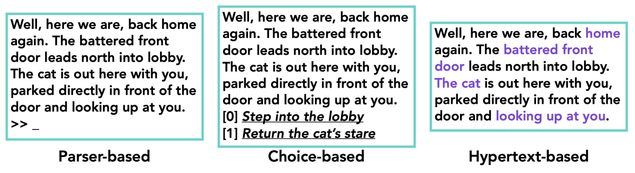
Fig. 1. A Classic Text Game

For convenience, we ignore the extra resources like sounds, images et al. These elements only improves the game experience but makes no effort on the process of artificial agents learning.

2.3 Genres And Types

Due to various game simulators, the input and output of the *IF games* varies. There are three genres of text input commonly used (Fig. 2 [2]):

- Parser-based, which accepts free text input
- Choice-based, which gives available choices
- Hypertext-based, which uses links in choices



Well, here we are, back home again. The battered front door leads north into lobby. The cat is out here with you, parked directly in front of the door and looking up at you.
>> _

Well, here we are, back home again. The battered front door leads north into lobby. The cat is out here with you, parked directly in front of the door and looking up at you.
[0] *Step into the lobby*
[1] *Return the cat's stare*

Well, here we are, back home again. The battered front door leads north into lobby. The cat is out here with you, parked directly in front of the door and looking up at you.

Fig. 2. 3 Types of Text Input

Among all types of *text games*, we focus on games with no additional parts that have no concern with complex input-output loops. The hyperlinks can be regarded as additional choices. Note that even a finite parameter parser-based games can be enumerated to the *choice-based games* at time steps, we only deal with *choice-based games*.

2.4 Rewards

IF games always have multi-endings based on multi-choices of player, therefore, it is significant to evaluate the rewards properly.

Some IF games use explicit numerical rewards to evaluate the results, such as *Six* [3]. The others output in textual form to reflect the player's performance, for example, *Star Court* [4].

In *Machine of Death* [5], each ending is a result of recapitulation from recent choices made by the player and it generates an exponential number of endings eventually. To avoid contingency, we test four IF games: *Saving John*, *Cat Simulator 2016*, *The Red Hair*, *Transit*.

In our games, we choose the explicit numerical rewards way to make it easier to do the final visualization. In order to determine the ending types, we only record the final states as feedback. In addition to the rewards of endings, each states has its own rewards based on the choice which the player made. We give an example of *Cat Simulator 2016* in the table below to show these rules directly.

Table 1. Cat Simulator 2016 rewards

Start of the ending text	Reward
this was a good idea	0
as good a place as any	-20
mine!	10
catlike reflexes	-20
finish this	-20
friendship	20
not this time, water	10
serendipity	10

2.5 Generalization

Up to now, there is no universal interface to different types of *IF games*, which cause a problem that it is hard to compare among different *IF games* with different engines.

Natural Language Processing (NLP) tasks focus more on the work with large amounts of data. Therefore, we call the pyfiction Python library to simplify access to the *IF games* and connect our games to *HTML* simulators.

[2]https://en.wikipedia.org/wiki/File:Zork_I_screenshot_video_game_Gargoyle_interpreter_on_Ubuntu_Linux.png

[3]http://ifdb.tads.org/viewgame?id=kbd990q5fp7pythi

[4]http://ifdb.tads.org/viewgame?id=u1v4q16f7gujdb2g

[5]http://ifdb.tads.org/viewgame?id=u212jed2a7ljg6hl

3 Algorithm

In the text game learning tasks, we normally utilize reinforcement learning (Fig. 3) to define the agent's object and consequently learn the the control policy.

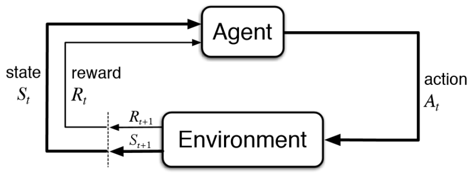


Fig. 3. Interaction between the agent and the environment [5]

There are different approaches to learning the optimal policy, here we focus on model-free methods [6] which prove to perform well in game-related tasks.

The optimal policy, denoted as π^* can be chosen from the actions with maximum Q-Values:

$$\pi^* = \max_a Q^*(s,a) \tag{1}$$

The optimal Q-values will obey the Bellman equation [7]:

$$Q^*(s_t,a_t) = E_{s_{t+1}}[r_t + \gamma \max_{a_{t+1}} Q^*(s_{t+1},a_{t+1})|s_t,a_t] \tag{2}$$

Consequently, the update rule for improving the estimate of the Q-function is as follows [5]:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha_t \cdot (r_{t+1} + \gamma \max_{a_{t+1}} Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)) \tag{3}$$

However, this assumes all Q-values for all state and action pairs are stored in a table, which is infeasible for complex tasks, where the state and action spaces are simply too large to store.Thus, we prefer applying approximate Q-Learning. And we choose the functions approximators as neural networks which can extract better feature compared with linear function approximators. The Q-function is parametrised as:

$$Q^*(s_t,a_t) \approx \theta_t(s_t,a_t) \tag{4}$$

the θ function is realized by a neural network.(see section 3.3)

In this chapter, we will present the architecture of our agent, which will be analyzed layer by layer.

3.1 Text Preprocessing

The individual text games are relatively small compared to other NLP tasks, thus we just employ a few simple rules to help reduce the vocabulary size.

We follow these rules for preprocessing:

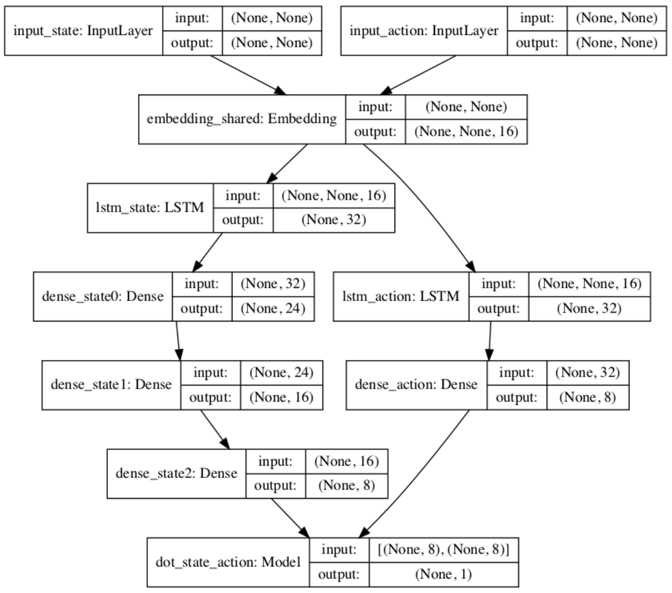


Fig. 4. Model Architecture

- Convert: all characters X should be the lower case x
- Split numbers: split 45787 to 4 5 7 8 7
- Insert space: convert expressions X's to X 's
- Expand: complete contracted expressions like $'ve$ and $'ll$
- Remove special characters : only preserve alphanumerical and whitespace characters, quotes and hyphens

3.2 Text Vectorize

At the beginning, we initialize the agent's vocabulary by either randomly sampling the simulators or specifying a list of words. Based on this vocabulary, we split the text after preprocessing and assign a unique token to each unique string to get a vector.

3.3 Agent Architecture

3.3.1 Embedding Layer

The inputs of the Embedding layer are the text after preprocessing(see section 3.1) and tokenized(see section 3.2).

We set our embedding dimensions to 16 since the word vocabulary is relatively small.

Importantly, the weights of the state and action branches are shared.Thus the same word in action description and state description will be converted to the same vector.

3.3.2 LSTM Layer

The inputs of the LSTM layer are the word embedding vectors(see section 3.3.1) with dimension of 16.

We set our LSTM dimensions to 32.

Long Short-Term Memory network is a representation generator to connect and recognize long-range patterns between words in text. They are more robust than Bag-of-Words and are able to capture underlying semantics of sentences to some extent.

3.3.3 Dense Layer

The inputs of the Dense layer(also called fully-connected layer) are the LSTM layer.

This layer plays a crucial role in building differentiated representations for state input and action input.

We use three hidden dense layers for the state branch, reducing the dimension from 32 to 24, from 24 to 16, from 16 to 8. And one hidden dense layer for the action state, directly reducing the dimension from 32 to 8. We choose different layer dimensions for them for the original state text descriptions usually carry more information than action descriptions.

3.3.4 Merging Layer

In the last layer, we apply cosine similarity to merge the action and state dense activations to get our final Q-value.

The cosine similarity equivalents is defined as below:

$$cs(\vec{x},\vec{y}) = \frac{\vec{x}\cdot\vec{y}}{\|\vec{x}\|_2\,\|\vec{y}\|_2} = \frac{\sum_{i=1}^{n}\vec{x}_i\vec{y}_i}{\sqrt{\sum_{i=1}^{n}\vec{x}_i^2}\sqrt{\sum_{i=1}^{n}\vec{y}_i^2}} \tag{5}$$

Obviously, the range of the Q-values are $[-1,1]$.In order to be consistent with the rewards received in the game environment we will scale the rewards from $[-r,r]$ to $[-1,1]$, where r is the environment parameter that denotes the max reward the user can obtain.

3.3.5 Loss Function And Gradient Descent

We apply "mean squared error" for the estimated Q-value and the target Q-value. We define the loss function as

$$L_t = (r_t - \gamma\cdot Q(s_{t+1},a_t))^2 \tag{6}$$

For gradient descent, we make use of the Adam optimiser that has shown to perform well in NLP tasks that contains both the benefits from RMSProp and AdaGrad.

3.4 Action Selection

Now foucus on the agent's exploration behavior. We update the parameter $\theta(state, action)$ in our Q-network for hundreds of times. In each iteration, we fix the parameter and select an action to explore.There are several algorithm for selection.

First is standard ε-greedy approach [8]. At the beginning of the training process, since we know little about the environment, we often initialize ε with a large probability (like 1), to encourage exploration, that is, at first, the agent's policy is completely random in training process. The value is then annealed down to a small constant (like 0.1) by multiply some constant (ε-decay) each iteration, to realise exploitation. While testing, the agent is greedy, i.e. ε is set to 0 and the agent always chooses the action with the maximum Q-value for the given state.

Algorithm 1 Stardard ε-greedy Approach

```
1: Input: state, actions, $\varepsilon$
2: if random() < $\varepsilon$ then
3:     return random action
4: else
5:     Qvalues = $Q(state, action, \theta)$ for action in actions
6:     return max Qvalues and action with max Qvalues
7: end if
```

Despite the prevalence of usage that it enjoys, this method is far from optimal, since it takes into account only whether actions are most rewarding or not.

Then we update the algorithm to softmax approach [9], which not choose action with maximum Q-value but choose action with probability according to their Q-value. That is, our algorithm changes from deterministic to probabilistic.

Algorithm 2 Softmax Approach

```
1: Input: state, actions, $\varepsilon$
2: if random() < $\varepsilon$ then
3:     return random action
4: else
5:     for action in actions do
6:         Compute the softmax equation for action:
7: $P_t(state, action) = \frac{exp(Q(state, action, \theta))}{\sum_{i=1}^{n} exp(Q(state, action_i, \theta))}$
8:     end for
9:     Choose one action $a$ in actions where each action
       with probability $P_t(state, action)$
10: end if
11: return $Q(state, a, \theta)$ and $a$
```

To balance the exploration and exploitation better, we update this method to Boltzmann Approach [10]. We utilize an additional temperature parameter (τ) which is annealed over time. This parameter controls the spread of the softmax distribution, such that all actions are considered equally at the start of training, and actions are sparsely distributed by the end of training.

Algorithm 3 Boltzmann Approach

```
1: Input: state, actions, $\varepsilon$
2: for action in actions do
3:     Compute the Boltzmann softmax equation for action
       (here we set $\tau = \varepsilon$):
4:     $P_\theta(state, action) = \frac{exp(Q(state, action, \theta)/\tau)}{\sum_{i=1}^{n} exp(Q(state, action_i, \theta)/\tau)}$
5: end for
6: Choose one action $a$ in actions where each action with
   probability $P_\theta(state, action)$
7: return $Q(state, a, \theta)$ and $a$
```

There is another method to reach our goal [9], with the help of history function $h(state, action)$. $h(s,a)$ returns the number of times the agent selected action a in state s in the current run which means this function penalises the already visited state-action pairs. The scope of the history function is reset every time a game ends.

Algorithm 4 Updated ε-greedy Approach

```
1:  Input: state, actions, $\epsilon$, $h$
2:  if random() < $\epsilon$ then
3:      return random action
4:  else
5:      Qvalues = $Q(state, action, \theta)$ for action in actions
6:      Qvalues = (Qvalues + 1) / 2
7:      $q_i = q_i^{h(state, action_i)+1}$ for $action_i$ in actions
8:      Qvalues = (Qvalues * 2) - 1
9:      return max Qvalues and action with max Qvalues
10: end if
```

3.5 Training loop

Now, we can describe the agent's learning algorithm formally.

We first use a variant of DQN with experience replay and prioritised sampling of experience tuples with positive rewards. NIPS DQN uses the experience pool (Experience Replay) based on the classic Deep q-learning algorithm. The training data are stored and then randomly sampled to reduce the correlation of data samples, which improve the performance. [3]

Algorithm 5 Training Use NIPS DQN

```
1:  Input: episodes num, b, p, $\epsilon$ = 1, $\epsilon$-decay
2:  Initialise experience memory $\mathcal{D}$ (experience pool)
3:  Initialise the parameter $\theta$ in neural network $Q$ randomly
4:  Initialise game simulators and load the vocabulary
5:  for i $\in$ [0,episodes num]  do
6:      Select action for current game state using the algorithm in 3.4, execute action and observe reward. If $s_{j+1}$ is the terminal state, restart the game.
7:      Store tuple (state,action,reward,next_state,next_actions, is_terminal) into $\mathcal{D}$
8:      Sample b tuples $(s_j, a_j, r_j, s_{j+1}, a_{j+1})$ from $\mathcal{D}$ as mini batch, where a fraction of p have $r_j > 0$.
9:      for $i, (s_j, a_j, r_j, s_{j+1}, actions\_next)$ in batch do
10:         Check if $s_{j+1}$ is terminal state.
11:         Set $y_i = \begin{cases} r_i, \text{ if } s_{i+1} \text{ is terminal state;} \\ r_i + \gamma\, max_{a_{i+1}} Q(s_{i+1}, a_{i+1}; \theta), elsewise \end{cases}$
12:     end for
13:     Loss function $\mathcal{L}(\theta) = (y_i - Q(s_i, a_i; \theta))^2$
14:     Perform gradient descent step on loss $\mathcal{L}(\theta)$
15:     $\epsilon = \epsilon \cdot \epsilon\_decay$
16: end for
```

One improvement to Nature DQN is the addition of the Target Q network (Q'). In other words, we use a specific target Q network to calculate the target Q value, instead of directly using the pre-updated Q network. The purpose of this is to reduce the correlation between the target Q-value and the current value. [11]

Algorithm 6 Training Use Nature DQN

```
1:  Input: episodes num, b, p, $\epsilon$ = 1, $\epsilon$-decay, C
2:  Initialise experience memory $\mathcal{D}$
3:  Initialise the parameter $\theta$ in neural network randomly. We have two copies of neural network: current network $Q(s,a;\theta)$ and target network $Q'(s,a;\theta')$, with $\theta = \theta'$
4:  Initialise game simulators and load the vocabulary
5:  for i $\in$ [0,episodes num]  do
6:      Select action for current game state in current network $Q$ using the algorithm in 3.4, execute action and observe reward.
7:      Store tuple (state,action,reward,next_state,next_actions, is_terminal) into $\mathcal{D}$
8:      Sample b tuples $(s_j, a_j, r_j, s_{j+1}, a_{j+1})$ from $\mathcal{D}$ as mini batch, where a fraction of p have $r_j > 0$. If $s_{j+1}$ is the terminal state, restart the game.
9:      for $i, (s_j, a_j, r_j, s_{j+1}, actions\_next)$ in batch do
10:         Check if $s_{j+1}$ is terminal state.
11:         Calculate $y_i$ in target Q-network Q':
12: $y_i = \begin{cases} r_i, \text{ if } s_{i+1} \text{ is terminal state;} \\ r_i + \gamma\, max_{a_{i+1}} Q'(s_{i+1}, a_{i+1}; \theta'), elsewise \end{cases}$
13:     end for
14:     Loss function $\mathcal{L}(\theta) = (y_i - Q(s_i, a_i; \theta))^2$
15:     Perform gradient descent on loss $\mathcal{L}(\theta)$, update $\theta$
16:     if $i\%c == 1$ then
17:         Update target Q-network Q': $\theta' = \theta$
18:     end if
19:     $\epsilon = \epsilon \cdot \epsilon\_decay$
20: end for
```

To reduce the estimation bias caused by the calculation of *max Q* (or known as over estimation), we try to improve Nature DQN to double DQN (DDQN). Use the current Q network to select the actions and the target Q network to calculate the target Q. [4]

Calculate y_i,by finding the action corresponding to the maximum Q value in current Q-network first:

$$a^{max}(s_{i+1}; \theta) = argmax_{a_{i+1}} Q(s_{i+1}, a_{i+1}; \theta) \tag{7}$$

Then use this action $a^{max}(s_{i+1}; \theta)$ to find Q-value in target Q-network:

$$y_i = r_i + \gamma\, Q'(s_{i+1}, a^{max}(s_{i+1}; \theta); \theta') \tag{8}$$

Algorithm 7 Training Use Double DQN

```
When Calculate $y_i$ in target Q-network Q':
replace $y_i = \begin{cases} r_i, \text{ if } s_{i+1} \text{ is terminal state;} \\ r_i + \gamma\, max_{a_{i+1}} Q'(s_{i+1}, a_{i+1}; \theta'), elsewise \end{cases}$ in
algorithm 6 by
$y_i = \begin{cases} r_i, \text{ if } s_{i+1} \text{ is terminal state;} \\ r_i + \gamma\, Q'(s_{i+1}, argmax_{a_{i+1}} Q(s_{i+1}, a_{i+1}; \theta); \theta'), elsewise \end{cases}$
```

Table 2. Information of Each Game

	SJ	C	TRH	T
No. of Tokens	1119	364	155	575
No. of States	70	37	18	76
No. of Endings	5	8	7	10
Avg Words/Description	73.9	74.4	28.7	87.0
Opt Reward	73.9	74.4	28.7	87.0

4 Experiments And Analysis

We do learning experiments by apply *Random Algorithm*, *SSAQN*, *SSNDQN*, *SSDDQN* and *DDDQN* agents on four games (*Saving John (SJ), Cat Simulator 2016 (C), The Red Hair (TRH) and Transit (T)*). All the necessary information of each game is given in Table 2.

Table 3. Training Algorithm Parameters

Parameter	Value
Optimiser	RMSProp
Learning Rate	0.0001
Batch Size	256
γ	0.95
ε	1
ε-delay	0.99
Prioritised Fraction	0.25

We set our parameters as Table 3 and 4. Unless Specified otherwise, we just use the default values given by Keras and Tensorflow.

Table 4. SSAQN/SSNDQN/SSDDQN/DDDQN Layer Dimensions

Layer	Dimension
Embedding	32
LSTM	32
Dense	8

We would like to evaluate the performance of generalisation, which means how well the models performs in unseen games. So we train the agent using the leave-one-out method, in which 3 out of the 4 games are provided to the agent as training simulators and the agent's performance is simultaneously evaluated on the 4th game by transferring test. And we test 5 times per algorithm per game to obtain the average reward.

We set 2 algorithms (Random and SSAQN) as cross reference to analyze the performance of SSNDQN, SSDDQN and DDDQN.

We would like to find out how each model performs range from the number of the episodes of each training. For this reason, we train the same time as test. For each episode, we train for one time and save the training data for the later episodes, and we test for five times and save the average rewards.

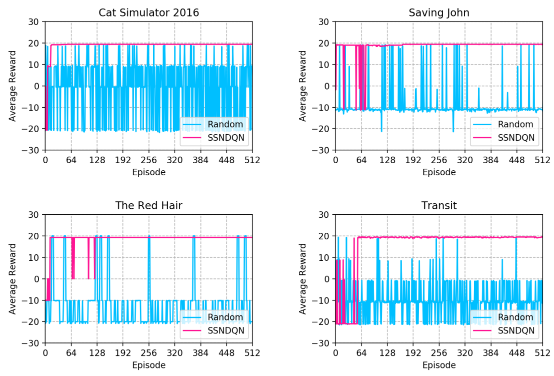


Fig. 5. Comparison of Random Algorithm and SSNDQN

4.1 Analysis on SSNDQN

Fig.5 compares Random Algorithm with SSNDQN on the 4 games. It indicates that Random Algorithm is not able to converge to optimal rewards while SSNDQN can converge to optimal rewards after some episodes.

Fig.6 compares SSAQN with SSNDQN on the 4 games. It indicates that both algorithms is able to converge to optimal rewards, but intuitively, SSNDQN performs better.

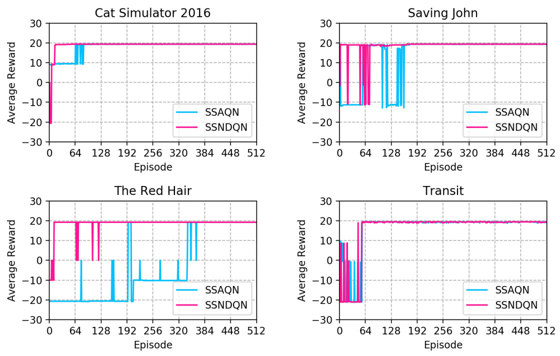


Fig. 6. Comparison of SSAQN and SSNDQN

4.2 Analysis on SSDDQN

Fig.7 compares Random Algorithm with SSDDQN on the 4 games. It indicates that Random Algorithm is not able to converge to optimal rewards while SSDDQN can converge to optimal rewards after some episodes.

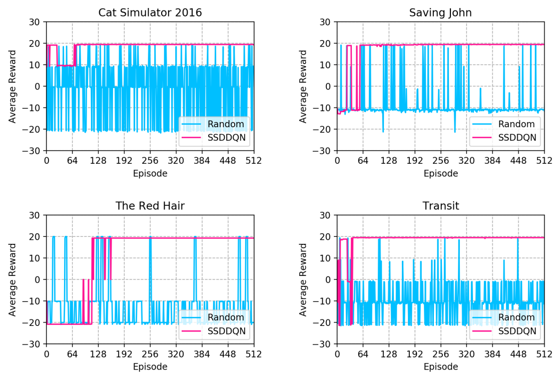


Fig. 7. Comparison of Random Algorithm and SSDDQN

Fig.8 compares SSAQN with SSDDQN on the 4 games. It indicates that both algorithms is able to converge to optimal rewards, but intuitively, SSDDQN performs better.

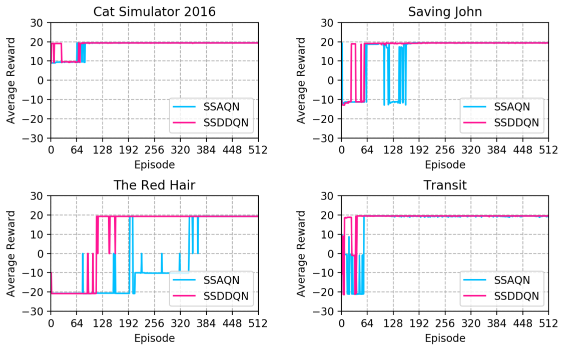


Fig. 8. Comparison of SSAQN and SSDDQN

4.3 Analysis on SSDDQN

Fig.9 compares Random Algorithm with DDDQN on the 4 games. It indicates that Random Algorithm is not able to converge to optimal rewards while DDDQN can converge to optimal rewards after some episodes except for *The Red Hair*. Maybe it is able to converge after more episodes, but the performance on *The Red Hair* with DDDQN is not so good as the former two algorithms.

Fig.10 compares SSAQN with DDDQN on the 4 games. It indicates that both algorithms is able to converge to optimal rewards, and intuitively, DDDQN performs better in *Cat Simulator 2016* and *Saving John* while SSAQN does better in *The Red Hair* and *Transit*.

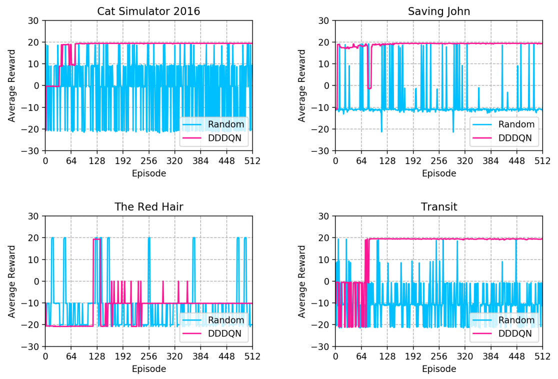


Fig. 9. Comparison of Random Algorithm and DDDQN

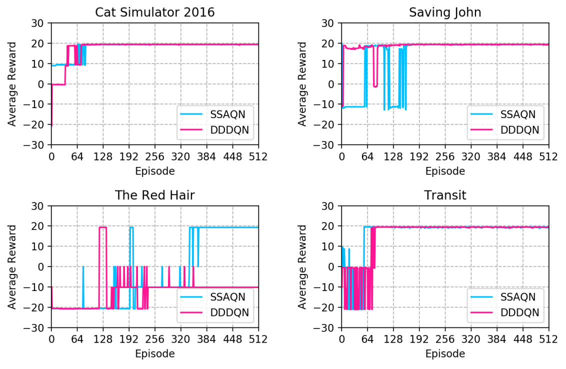


Fig. 10. Comparison of SSAQN and DDDQN

4.4 Comparison of All The Algorithms

Fig.11 compares all the algorithms on the 4 games. Intuitively, it is obvious that SSNDQN and SSDDQN performs better than SSAQN and DDDQN. But it is hard to say whether SSNDQN is better or SSDDQN is better.

It seems that all the models performs well in generalisation except DDDQN on *The Red Hair*. In other words, based on the current set of parameters, DDDQN model is overfitting.

4.5 Conclusion of The Experiment

In conclusion, we have finished the initiatory evaluation on our models as shown in former sections. Due to the limitation of time, we should have trained all the models for more episodes and for each model, we should have run for more times, such as 100 times, to obtain the average rewards which can really minimize special cases.

5 Conclusion

In this project, we develop three deep reinforcement relevance networks, SSNDQN, SSDDQN and DDDQN models for handling states and actions described by natural languages in text games. We find that SSNDQN and SSDDQN converges better than DDDQN and the original model,

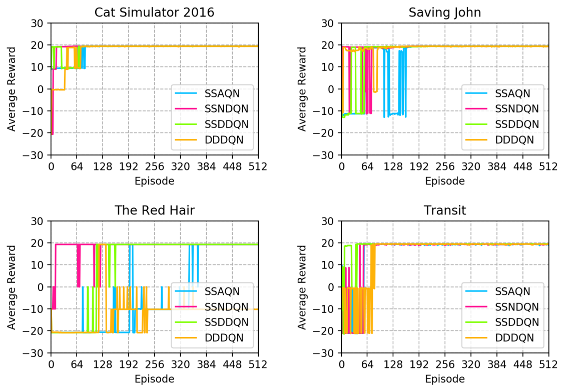


Fig. 11. Comparison of All The Algorithms

SSAQN, because the improvements on neural network architecture and the training model. However, we should have trained all the models for more episodes, and for each model we should have run for more times to obtain the average rewards which can really minimize the impact of uncertainty. Furthurmore, our current models only fit for choice-based games, so we may generalize our models for other forms of text games.

References

[1] Zelinka, M., 2018. "Using reinforcement learning to learn how to play text-based games". *MASTER THESIS*.

[2] Ji He, Jianshu Chen, X. H. J. G. L. L. L. D., and Ostendorf, M., 2016. "Deep reinforcement learning with an unbounded action space". *CoRR*.

[3] Narasimhan, K., Kulkarni, T. D., and Barzilay, R., 2015. "Language understanding for text-based games using deep reinforcement learning". *CoRR*.

[4] Hado van Hasselt, Arthur Guez, D. S., 2016. "Deep reinforcement learning with double q-learning". *AAAI*.

[5] Sutton, R. S., and Barto, A. G., 1998. "Reinforcement learning : an introduction". *MIT Press*.

[6] Tesauro, G., and Center, I. T. J. W. R., 1995. "Temporal difference learning and td-gammon". *Acm*.

[7] Bellman, R., 2013. "Dynamic programming". *Courier Corporation*.

[8] Watkins, C., 1989. "Learning from delayed rewards". *PhD thesis, University of Cambridge, Cambridge, England*.

[9] Sutton, R. S., and Barto, A. G., 1998. *Reinforcement learning : an introduction*. MIT Press,.

[10] Wiering, M., 1999. "Explorations in efficient reinforcement learning". *PhD thesis, University of Amserdam, Amsterdam*.

[11] V. Mnih, K. Kavukcuoglu, D. S. A. A. R. J. V. M. G. B. A. G. M. R. A. K. F. G. O. S. P. C. B. A. S. I. A. H. K. D. K. D. S. L., and Hassabis., D., 2015. "Human-level control through deep reinforcement learning.". *Nature*.