Text-Room Adventure Engineering Design Document

Fei Wang fwang598@usc.edu
Zhongheng He hezhongh@usc.edu
Kainuo Feng kainuofe@usc.edu
Yongcheng Wang ycwang039@usc.edu

Abstract

In interactive fiction, players interact with unknown game worlds described in natural language. Most of the interactions rely on commonsense knowledge. Previous works have tried to use knowledge graphs to represent commonsense knowledge explicitly. However, none of existing works utilized rich commonsense knowledge outside the game world. In this paper, we incorporate pre-trained language models into the deep Q-learning framework to bring in rich commonsense knowledge existing in external text corpus, and combine history actions into decision making process. Further, we build a 2D simulator that can visualize the game process to implement case study and assist detailed performance analysis. Experimental results on the Jericho[1] platform show that our method performs better than several strong baseline methods.

1 Introduction

Interactive fiction (IF) is a software simulating environment where players use text commands to control characters and influence the environment (Ziegfeld, 1989) with a history of more than 60 years. In the common game setting of IF, the player starts the game at an unknown game world described in natural language and needs to interact with all possible objects around to explore the world and finally win the game.

IF games rely on the player's commonsense knowledge as a prior for how to interact with the game world (Hausknecht et al., 2020). For example, when encountering locked doors, human players intuitively understand that they need some keys. However, such commonsense knowledge is hard to learn from scratch directly through interaction with the game environments.

Besides, the player's history actions are continuous, and they are implicitly connected with each other. If we can encode the history actions as supplement information when making decisions, our model might perform better. Therefore, we combine history actions into the Q-learning computation since prior knowledge can be helpful in the continuous decision process, just like humans.

Previous works have tried to use knowledge graphs (KG) to represent commonsense knowledge explicitly (Ammanabrolu and Riedl, 2019a,b; Ammanabrolu and Hausknecht, 2019). These works constructed KG through interaction with the game environments. Although Ammanabrolu and Riedl (2019b) tried to transfer the commonsense knowledge learned from different games, none of existing works utilized rich commonsense knowledge outside the game world.

In this project, we incorporate pre-trained language models, such as BERT (Devlin et al., 2019), into the deep Q-learning framework (Mnih et al., 2013) to train AI agents for IF. Pre-trained language models are trained on large text corpus containing rich commonsense knowledge. Besides, we also combine history actions into the Q-learning computation since prior knowledge can be helpful in the continuous decision process, just like humans.

[1] https://github.com/microsoft/jericho

34th Conference on Neural Information Processing Systems (NeurIPS 2020), Vancouver, Canada.

Experimental results on Jericho platform (Hausknecht et al., 2020) show that our method performs better than several strong baseline methods. To further verify the effectiveness of our method, we build a 2D simulator to visualize the game environments and actions issued by models. Detailed analysis shows that our predictions are highly consist with gold actions.

In summary, our contribution is two-fold. First, we include both external commonsense knowledge and internal prior knowledge to the AI agents, by incorporating pre-trained language models and encoding history actions. Second, we build a 2D simulator that can visualize the game process to assist detailed performance analysis.

2 Overview of the text-adventure games

2.1 Interactive Fiction

The origin of interactive fiction can be followed to the 1960s and 1970s, when the simple natural language processing can be applied onto software programs and enables the program to process the uses' input and then response with human-like text messages. And it was also in the 1970s, Will Crowther, a programmer and an amateur caver, wrote the first text adventure game, ADVENT[2], which was further spread to the Internet and inspired many people to design and write their own text adventure games, like the Dog Star Adventure[3], the ZORK series[4], etc.

In the 1980s, IF became a standard product for many software companies and, the interactive fiction occurred outside the U.S. At that time, in Italy, IFs were published and distributed through various magazines in included tapes.

From the 1990s to this modern era, the market of IF is declining due to the growth of multi-media games, because the videos games seems much more amusing. However, we can still find the wisdom in IF.

Figure 1: Oldest Text Adventure Games: ADVENT, Dog Star Adventure, Zork

2.2 Game Environments and Platforms

From an interactive fiction to an interactive game, we need a platform to support the game environment and control the workflow. There are plenty of open source platforms able to create traditional parser-driven IF in which the user types commands — for example, go east, go downstairs, read the newspaper — to interact with the game. From the 1990s to present, we have many successful IF platform. And in the past decade, the most popular platforms might be the JerichoHausknecht et al. (2020) and TextWorldCôté et al. (2018) from Microsoft. And Facebook, Inc. also launched LIGHTJack Urbanek (2019) as its text interactive platform for the dialogue research.

[2]http://hdl.handle.net/2142/16406

[3]https://www.mobygames.com/game/dog-star-adventure

[4]https://adventuregamers.com/gameseries/view/1582

In this project, we select Jericho[5] as our platform for the text-adventure games. It is a lightweight python-based interface connecting learning agents with interactive fiction games. It runs on Linux-like systems and is easy to install or serve. Figure 3 is a game initialized in a MacOS terminal.

Figure 2: Jericho Framework

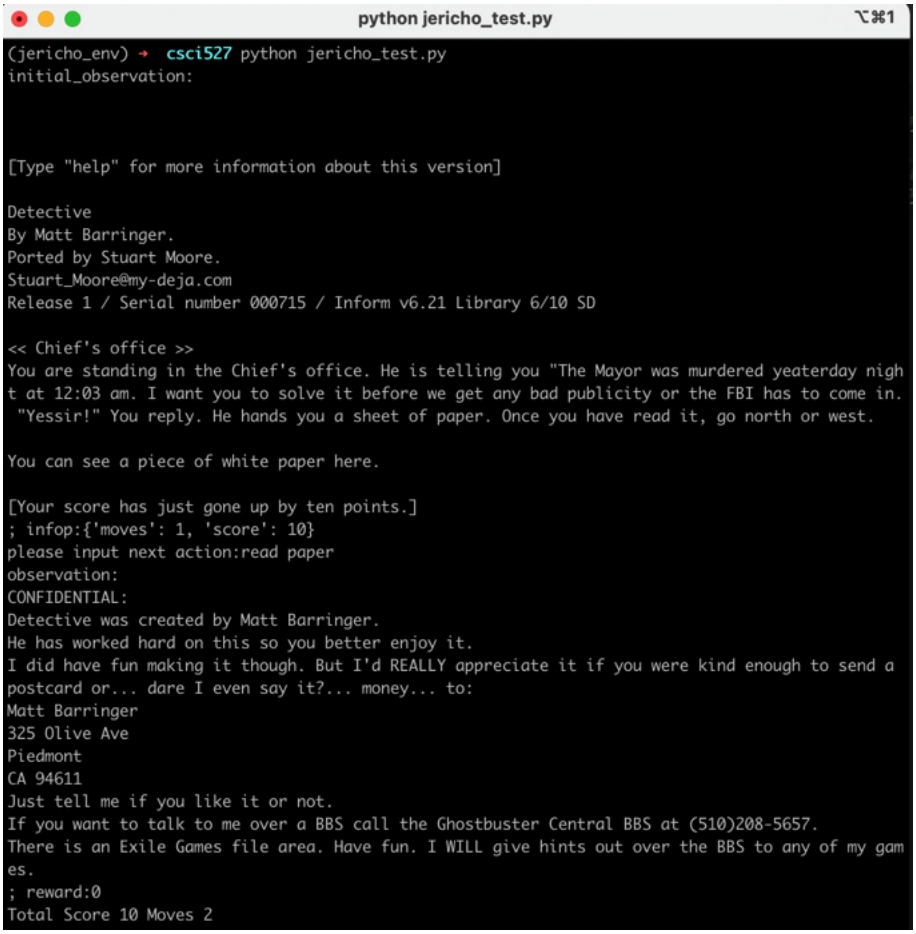

Figure 3: Initialization of a text adventure game

2.3 Game Scenario

We choose a single player text-adventure game whose name is "Detective" as an example to show you he basic scenario of a text-adventure game: the player character is a famous detective. At the beginning of the game, the player is standing in the police chief's office and is asked to investigate the mayor's murder case. The player should search the neighbor area for clues and evidence. Some events and clues are embedded in room descriptions.

The entire game is based on text and we only need a terminal with python environment to run it. Look at the Figure 4, the game process is formed by three elements: observation, action and reward. Each

[5]https://jericho-py.readthedocs.io/en/latest/index.html

turn, the player gets the observation, and then take action according to this information. Then the game engine will tell the player how many scores or reward he gets by taking this action. Just as

The Figure 5 is a detailed example of our game. Through the observation, the player finds a gun on the floor. And he just typed “take gun” in the terminal as an action. Then he got 10 scores as a reward because this is a clue for the murder case. For some useless action, you can also see that the player gets zero reward.

The ultimate goal of the player is to get as much reward as possible in a limited number of actions.

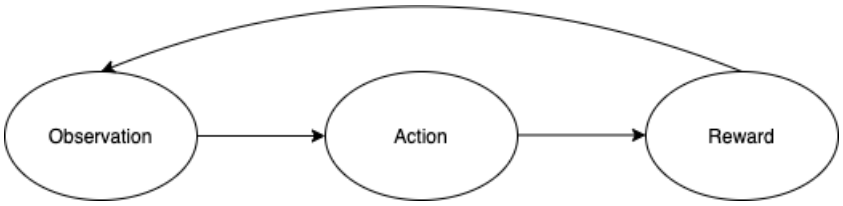

Figure 4: Agent Work Process

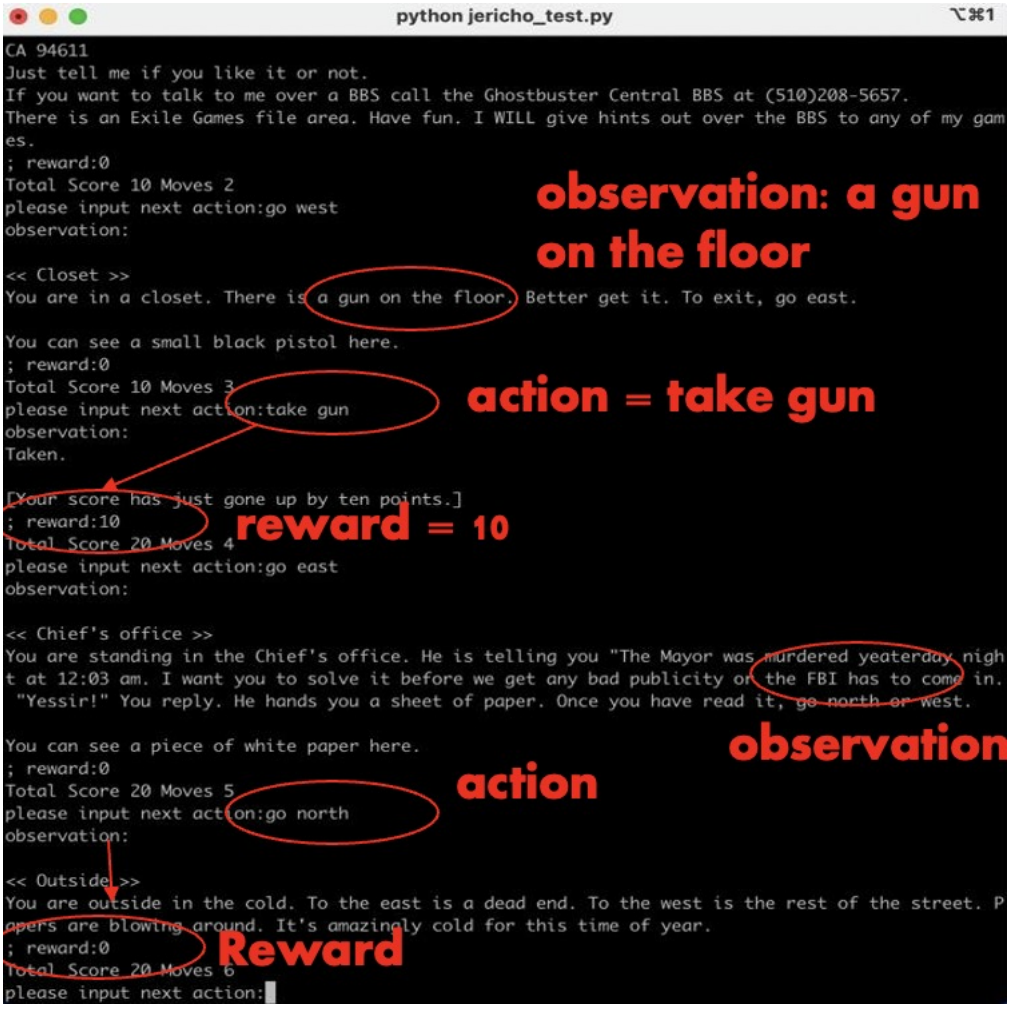

Figure 5: Demo of Observation-Action-Reward

3 Related work

3.1 Prior research on text-adventure games

IF Game Agent has been well explored and applied on IF game agents, focusing on text understanding and interaction between state information and action. Narasimhan et al. proposed LSTM-DQN, which utilized LSTM as a representation generator to jointly learn state representations and action policies using game rewards as feedback. Hausknecht et al. extend LSTM-DQN as Template-DQN for template-based action generation, by using the output of LSTM-DQN to predict the placeholds in the template. Deep Reinforcement Relevance Network (DRRN) (He et al., 2016) was introduced for choice-based action selection. It represented action and state spaces with separate embedding, which are then combined with an interaction function to approximate the Q-function in reinforcement learning.

More recently, novel combinations of text game and other tasks (i.e., machine reading comprehension) has been proposed, which further improves the performance of the agents. Guo, Xiaoxiao, et al. used a reading comprehension model with Bidirectional Attention Flow (BiDAF) to encode state and action information, by treating the observation as a context and action as a query like a question and answering task. Moreover, they retrieve past observations in previous steps to determine long-term effects of action, aiming to tackle the partial observability of text games. Yao, Shunyu, et al. used a pre-train language model (i.e., GPT-2) to generate candidate actions, showing the potential of direct action-generation. Ammanabrolu et al. proposed a graph-based deep reinforcement learning. They represented the game state as a knowledge graph and updated it during the exploration. And when the agent needs to select the actions, we turn the information of the knowledge graph to a single vector and use it in a neural network. This work was further extended to KG-A2C (Ammanabrolu et al, 2020) by exploring and generating actions using a template-based action space and utilizing Advantage Actor Critic training method.

3.2 Pre-trained language model

Recently, language models pre-trained on large scale of corpus (BERT, gpt, elmo, xlnet, erine etc) demonstrate surprising power in encoding context and semantic information and bring natural language processing to a new era, among which the BERT obtained the state-of-the-art results on eleven natural language processing tasks when it was proposed in 2018. Nowadays, the BERT has become the backbone and foundation of many other works related to natural language processing. (span bert, albert, roberta)

In detail, BERT is a transformer-based(Vaswani et al., 2017) architecture pre trained with two tasks: Masked Language Model (MLM) and Next Sentence Prediction (NSP). In MLM, the model needs to predict the masked tokens in the original sentence, and in this way, it learns to capture the bidirectional contextual information. In NSP, the model needs to predict whether the given two sentences are consecutive, aiming to capture the relationship between different sentences. After the pretraining, BERT can be fine-tuned in many downstream tasks (i.e., Question Answering).

In our work, we use BERT to encode the text description and explore the relationship between state and actions.

Figure 6: Logo of Google BERT

4 Schedules

4.1 Pre-Midterm Schedule

In week 2 and 3, we played different text adventure games and made a background research of the interactive fictions. We tried different games on different platforms, and finally selected Jericho as our game platform.

In week 4, we researched relative work and designed our basic model framework. Besides, we made plans on building a visual simulator to support our research. In week 5 and 6, we successfully reproduced our baseline on the DRRN framework, and added BERT onto it. Besides, we successfully built our visual simulator on the Unity 2D engine.

In week 7 and 8, we tried another two components, the Two-tower State-action Matching and Cross-attention State-action Matching on our DRRN_BERT model, and successfully beat several strong baselines. We made case study with the help of our visual simulator and found some new directions to improve our model.

In addition, we joined the guest lecture - LIGHT: TRAINING AGENTS THAT CAN ACT AND SPEAK WITH OTHER MODELS AND HUMANS IN A RICH TEXT ADVENTURE GAME WORLD7, raised by the USC information science institute[6] and learned what Facebook is doing with their text adventure games and how they collect lots of useful data for similar research.

This is the contribution of each member in our group. And Fei will talk about our model in detail.

[6]https://www.isi.edu/events/calendar/13633/

LIGHT: TRAINING AGENTS THAT CAN ACT AND SPEAK WITH OTHER MODELS AND HUMANS IN A RICH TEXT ADVENTURE GAME WORLD

When: Thursday, March 4, 2021, 11:00 am - 12:00 pm PDT ICAL

Where: This talk will be live streamed only, it will Not Be recorded.

This event is open to the public.

Type: NL Seminar

Speaker: Jason Weston (FAIR/NYU)

Video Recording: NA

Description: **Abstract:** LIGHT is a rich fantasy text adventure game environment featuring dialogue and actions between agents in the world, which consist of both models and humans. I will summarize work on building this research platform, including crowdsourcing and machine learning to build the rich world environment, training agents to speak and act within it, and deploying the game for lifelong learning of agents by interacting with humans.
See https://parl.ai/projects/light/

Light - Parl
LIGHT Learning in Interactive Games with Humans and Text. The LIGHT project is a large-scale fantasy text adventure game research platform for training agents that can both talk and act, interacting either with other models or with humans.
parl.ai

(and the talk!) for more.

Figure 7: Guest lecture of LIGHT

4.2 Post-Midterm Schedule

In week 9 to 11, we developed our history-actions encoder, and added it onto our DRRN_BERT model, we called it DRRN_BERT_MEMO model. At this stage, we use concat method to build the history-actions encoder, and tried it on the Two-tower State-action Matching and Cross-attention State-action Matching architectures separately. It worked better on the Cross-attention State-action Matching architecture.

In week 11 to 12, we further developed a LSTM encoder for the history actions. Although it improved our performance a little bit, it could not beat the previous model with concat encoder. Meanwhile ,through case study, we found that sometimes our model might stick to a local optimal action and ignore the global optimal decision. This may further create a local endless loop.

In week 13 to 14, we further added constraints to avoid the endless loop and lead our model to the global optimization choice, which improved our model score by 2 points in the complex Deephome text game.

5 Method

In this section, we first introduce the deep reinforcement learning framework for text adventure games, and then describe two types of state-action matching models.

5.1 Overview of the Learning Framework

Following previous works He et al. (2016); Hausknecht et al. (2020), we solve the sequential decision making problem for text adventure games with deep Q-learning Mnih et al. (2013).

The game starts with description of the game background s_0. At each time step t, the agent issues a textual action $a_t \in A_t$ based on the environment description s_t. Then the agent receives a reward r_t according to the game score earned by a_t. The environment description is updated to s_{t+1}. The game stops when the agent reach some special states or the maximum steps.

We define the Q-function as the expected reward of taking the action a_t under the state s_t

$$Q(s_t, a_t) = E[r + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) | s_t, a_t],$$

where γ is a discount factor. For inference, we will choose the action with the maximum Q-value

$$\pi(s_t) = \arg\max_{a_t \in A_t} Q(s_t, a_t).$$

In our framework, the Q-function is fitted by deep text matching networks.

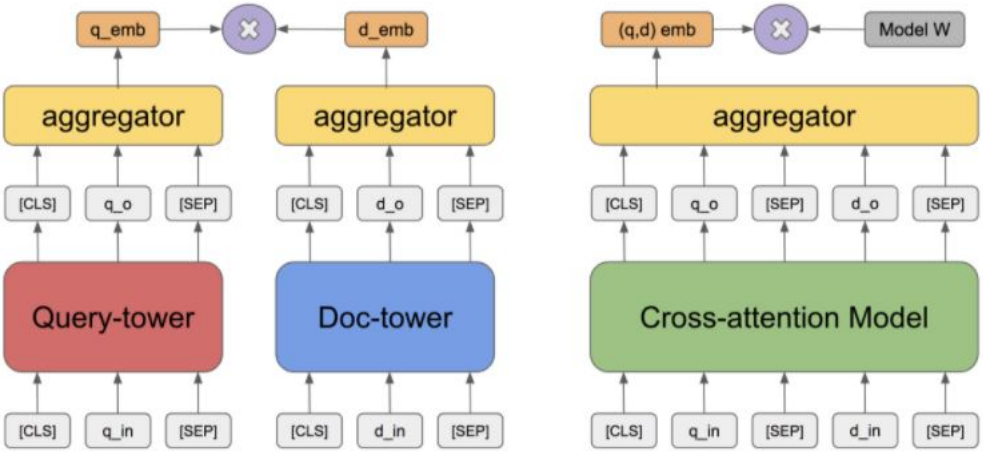

Figure 8: Two-tower model (left) and cross-attention model (right). Figure is copied from Chang et al. (2019).

5.2 Two-tower State-action Matching

Two-tower matching models have been used in various of text matching tasks Huang et al. (2013); He et al. (2016); Chang et al. (2019). The query and candidate embeddings are from two independent text encoders and then aggregated to calculate the matching score.

We apply BERT Devlin et al. (2019) as text encoders for both actions and states. The textual descriptions of candidate actions and states are encoded by different BERT models. The encoded representations are concatenated and then fed to a multi-layer perceptron (MLP).

5.3 Cross-attention State-action Matching

Cross-attention matching models is another type of matching models. Previous work Guo et al. (2020) have shown that cross-attention models usually produce better results but cost longer computation time in comparison with two-tower models.

We apply BERT as the joint encoder of actions and states. Following Devlin et al. (2019), we combine the text sequence of action and state and distinguish them by adding different segment tokens. The final representation of the first token `[CLS]` is then sent to a MLP to calculate the score.

5.4 BERT Incorporation

We use the BERT-base model released by Google[7]. For the two-tower model, we set the max sequence length of actions as 16 and that of states as 256. For the cross-attention model, we set the max sequence length as 512. The model is trained asynchronously on 8 parallel instances of the game environment for 20k steps with a batch size of 64. Following previous work Hausknecht et al. (2020), episodes are terminated after 100 valid steps or game over/victory. The learning rate is set to 1e-5 as suggested by Devlin et al. (2019). γ is set to 0.9. We implement our method based on Pytorch Paszke et al. (2019) and Transformers Wolf et al. (2019). Listing 1 and 2 shows some code snippets.

Listing 1: Example code snippet for using the Transformers package.

```
from transformers import AutoTokenizer, AutoModel

tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")
model = AutoModel.from_pretrained("bert-base-uncased")

inputs = tokenizer("Hello␣world!", return_tensors="pt")
outputs = model(**inputs)
```

[7]https://github.com/google-research/bert

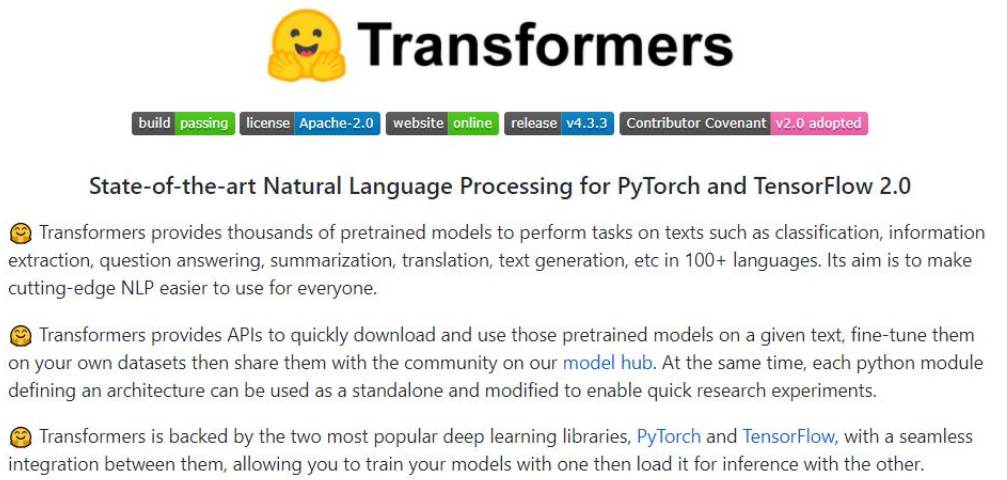

Figure 9: Description of the Transformers package.

Listing 2: Code snippet for encoding states and actions using BERT.

```
def bert_encode(self, actions, state, act_sizes):
    # action input
    act_input_ids = torch.cat([x['input_ids']
                    for x in actions], 0).cuda()
    act_attention_masks = torch.cat([x['attention_mask']
                    for x in actions], 0).cuda()
    act_segment_ids = torch.zeros_like(act_input_ids)

    # state input
    obs_input_ids = torch.cat([x['input_ids']
                    for x in state.obs], 0).cuda()
    obs_attention_masks = torch.cat([x['attention_mask']
                    for x in state.obs], 0).cuda()
    look_input_ids = torch.cat([x['input_ids']
                    for x in state.description], 0).cuda()
    look_attention_masks = torch.cat([x['attention_mask']
                    for x in state.description], 0).cuda()
    inv_input_ids = torch.cat([x['input_ids']
                    for x in state.inventory], 0).cuda()
    inv_attention_masks = torch.cat([x['attention_mask']
                    for x in state.inventory], 0).cuda()
    state_input_ids = torch.cat((obs_input_ids,
                    look_input_ids, inv_input_ids), dim=1)
    state_attention_masks = torch.cat((obs_attention_masks,
                    look_attention_masks, inv_attention_masks), dim=1)

    # Expand the state to match the batches of actions
    state_input_ids = torch.cat([state_input_ids[i].repeat(j, 1)
                    for i, j in enumerate(act_sizes)], dim=0)
    state_attention_masks = torch.cat([state_attention_masks[i].repeat(j, 1)
                    for i, j in enumerate(act_sizes)], dim=0)
    state_segment_ids = torch.ones_like(state_input_ids)

    # Concat along hidden_dim
    input_ids = torch.cat((act_input_ids, state_input_ids), dim=1)
    attention_masks = torch.cat((act_attention_masks,
```

```
                    state_attention_masks), dim=1)
segment_ids = torch.cat((act_segment_ids, state_segment_ids), dim=1)

outputs = self.bert_encoder(input_ids=input_ids,
                    attention_mask=attention_masks,
                    token_type_ids=segment_ids)['pooler_output']

return outputs
```

5.5 Encode History Actions

The history states and actions are also useful for predicting the next action, because the action sequences are continuous and highly related. We combine the current and history states and actions as model input. And implemented two types of history encoders. The first version is simply concatenated the history states and actions to the current state and action sequence. The second version is applying long short-term memory network as memory module. The model architecture is shown in Figure 10.

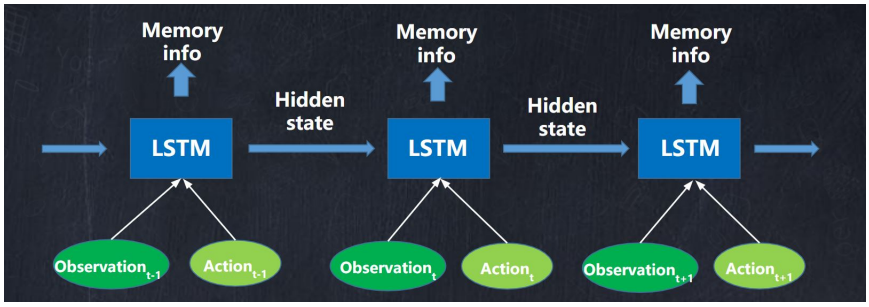

Figure 10: LSTM based memory module.

5.6 Break Circular States

According to the case study in Figure 11, we found that our agents sometimes are trapped by circular states. The agent may repeatedly visit the same states. To address this problem, we design a count-and-switch operator to break the circular states. The algorithm is shown in Algorithm 1.

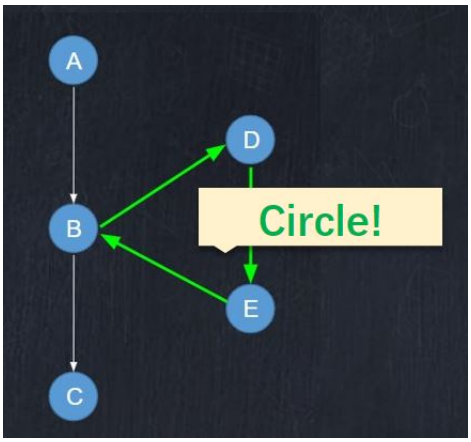

Figure 11: Cirular states. The agents may visit B, D and E repeatedly.

Algorithm 1 Breaking Circular States

```
Initialize a dict T to record the number of times each state is visited;
while game is not finished do
    Observe the state s;
    Update T;
    Sort valid actions A;
    Issue an action A[T[s]];
end while
```

6 Pre-midterm Experiments

6.1 Main results

Table 1: Scores for different games

Game	detective	deephome
best steps	50	350
RAND	113.7	1
NAIL	136.9	13.3
TDQN	169	1
DRRN	197.8	1
DRRN_BERT	200	14
DRRN_BERT_MEMO	**290**	**27**
MaxScore	360	300

Table 2 shows the performance of our model and baselines on detective game. DRRN performs the best among all baselines. Our two-tower model DRRN_BERT is better than DRRN. The improvement shows that text adventure games can benefit from pre-trained language models.

Figure 12 shows the achieved scores during training. It is obvious that DRRN_BERT converges faster and the learning process is more stable. We suppose that it is because we start from the pre-trained language models instead of learning from scratch.

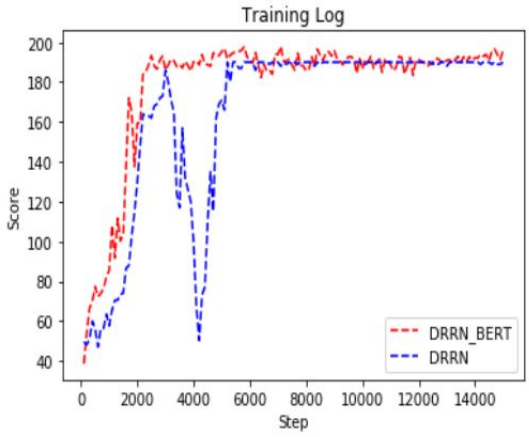

Figure 12: Training process of different methods.

6.2 Visualization

The Figure 13 is an abstract 2D map of our game. You can see that there are tens of rooms and buildings. And in some rooms, there are also lots properties and evidence which are crucial to the murder case. Just like the gun the previous slide, the player needs to explore the map and collect these properties.

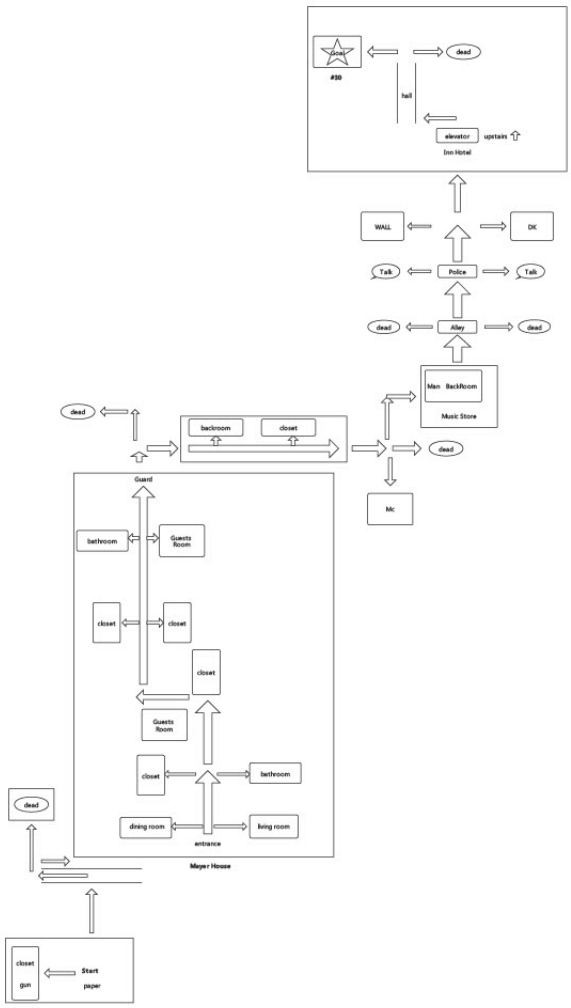

Figure 13: Abstract 2D map of the game

While our game is purely based on natural language, sometimes it's hard to catch the decision process of our model throughout the whole game. Therefore, we plan to build a 2D map on Unity and use it as a simulator to better demonstrate the continuous actions of our model. This can help us analyze our model and improve it at a high level.

The Figure 14 is the structure of the whole game world in unity 2D. We can walk on the street, pick a random building, and lead to different result. It may have some clues to get you a reward, may be a common place with nothing important, or may cause you to die.

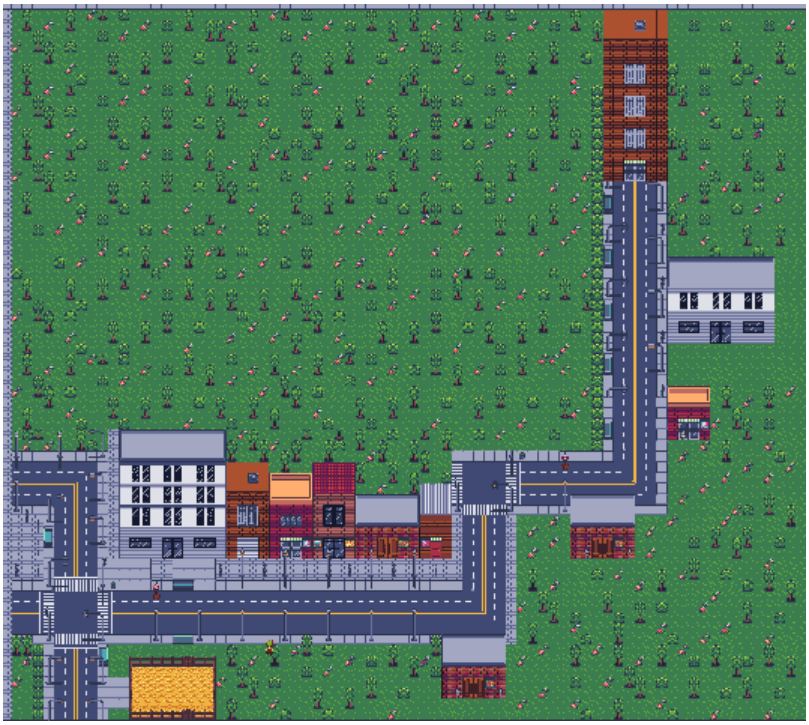

Figure 14: Unity 2D map of the game world

The Figure 15 is the inside of a building. When we are at the front door, depending on the action the player chooses, we may go to different rooms and meet different events.

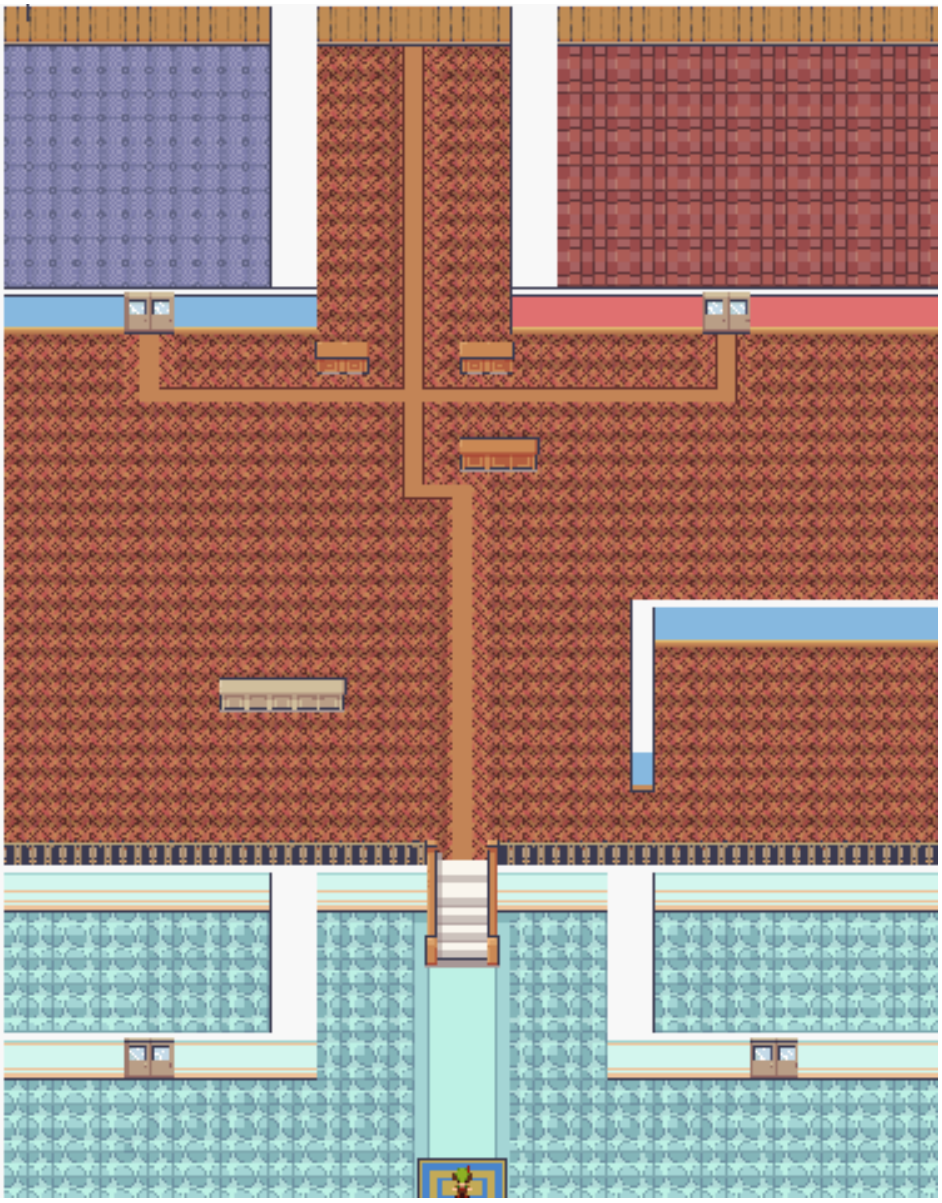

Figure 15: Unity 2D map of the inside of a building

The scenes are properly ordered in unity2D and the main character has been put into every scene. Our player is able to move flexibly in the scene with the camera following all the time. The Figure 16 is the three-view image of our main character.

In unity, we build three folders for the images of the main character facing every direction. We use a class called PlayerMovement to control the movement of the player. In PlayerMovement class, to deal with the position of our main character, we choose to use fixed update function to make sure it doesn't depend on frame rate. We also need an animator to manage the animation of our main character. We have built three animations: walk up, walk down and walk side for different directions the player is heading. We only need walk right image and walk right animation, because for walk left case, we can just flip our character.

There are warp points at the entrance and exit of every scene. When we reach a warp point, the main player will be automatically transported to a specific place in another scene according to the index of warp point and current scene. By now we have finished part of the warp points, we can use these to travel between scenes.

Figure 16: Three-view image of our main character

The interact points are set on the scenes. In this game, when the player get in a specific area, he will get an observation of the new environment around him. When he interact with an object, he will get the information of this object. By now we have finished part of the interact points. When the player get near a interact point, a dialogue box will pop out and tell you what you have observed. The Figure 17 is an example of a dialogue box when we enter a room.

In Unity, we use a class called ObjectDialogue to store the form of an observation, which is observation name and the content of the observation. For each area with an observation, we need a DialogueTrigger class to trigger the dialogue box. And for each dialogue trigger, we need to give a value to the ObjectDialogue. For example, when our player get into the living room and see a battered piece of wood, we should set ObjectDialogue.name as "living room" and ObjectDialogue.content as "You see a battered piece of wood." We also need an animator to manage the in and out animation of the dialogue box. When the dialogue box is closed, it goes below the canvas, and when the dialogue box is open, it pops up.

Our ultimate goal of the visualization is that creating an engine which can read the player actions from the model output and automatically simulate the movements and player-environment interaction in our map.

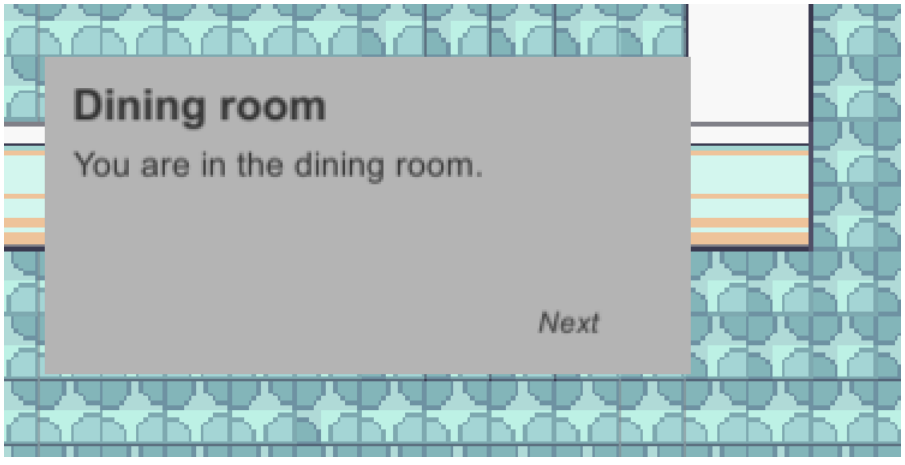

Figure 17: A dialogue box example

6.3 Pre-midterm Case Study

Figure 18 shows the difference between the result of our work and the standard human-played walk-through offered by the game developer. There are two walk-through paths in the figure, the green arrows represent the model output while the red arrows represent the standard human strategy.

Our model manages to avoid all the dangerous events and successfully arrives at the exit point. The two routes overlap to a large extent. In this specific game, Detective, our model is able to behave and take actions like a human. It utilizes the previous information and take actions under the reward policy of this game.

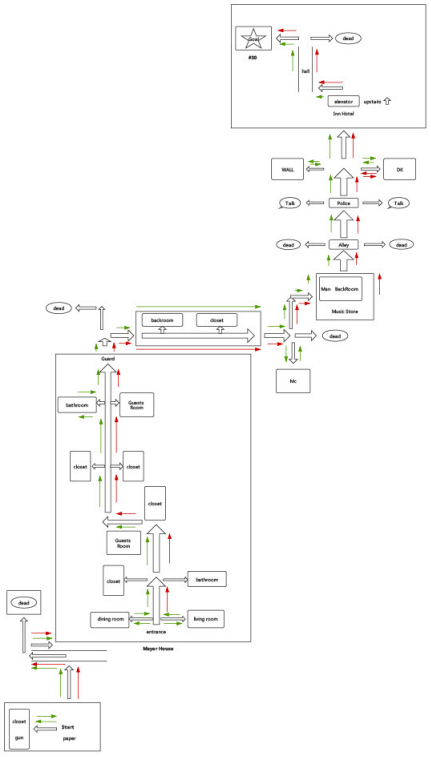

Figure 18: The results of our model output (green) and the standard human player walk-through (red).

6.4 Pre-midterm Limitations

Although we arrived at a milestone of several games, we still meet with several limitations. First, the games we selected in the prior-midterm session is simple and the total number of actions we need to win a game is less than one hundred. The experiments we took yet can not verify the universal ability of our model in different games setting especially in super complex games. Second, the reward policy of the games we selected is quite stable. For example, in the Detective game, each valid action normally get 10 to 20 scores of reward. We need to further verify the ability of our model in games where the rewards vary a lot. The model needs to trade-off when facing with a candy and a set of delicious meal.

7 Post-midterm Experiments

7.1 Outlook at the midterm

In the prior-midterm session, we incorporate pre-trained language models into the deep Q-learning framework to bring in rich commonsense knowledge existing in large text corpus.We successfully

designed and built our DRRN_BERT model and beats the previous state-of-the-art method. Our model performs well in several simple text-adventure games and achieves scores close to human performance.

In the post-midterm sessions, we plan to upgrade our model and further test its ability in complex text games with a super large action space, in which human players could not get a relatively high score. To achieve this goal, we want to add two modules into our model. The first module is called memory. Action selection is not independent between different steps, so the latest observation is not always sufficient for us and we need previous observation and selected actions to help the agent to build a complete trail. For instance, we can store our observations and actions as dense vectors during the exploration step. When the agent needs to select an action, it can combine the historical information to make a better choice. The second module is the world graph module, which can be implemented as a knowledge graph. Knowledge graph is a perfect tool to manage the objects that the agent has met and known and help the agent to represent the world that it has explored. We can update this map during the exploration and use the graph embedding model to convert the graph information into vectors and combine it in the model.

7.2 Post-midterm Experiments and results

In the pre-midterm session, the games we selected in the prior-midterm session is simple and the total number of actions we need to win a game is less than one hundred. And the reward policy of the games we selected is quite stable. For example, in the Detective game, each valid action normally gets 10 to 20 scores of reward. Therefore, we plan to further launch experiments on large games. Deephome is a super complex text-adventure game, which costs nearly 300 steps to walk through.

*** to be updated by Fei ***

Table 2: Scores for different games

Game	best steps	RAND	NAIL	TDQN	DRRN	DRRN_BERT	DRRN_BERT_MEMO	MaxScore
detective	50	113.7	136.9	169	197.8	200	**290**	360
deephome	350	1	13.3	1	1	14	**27**	300

7.3 Visualization

The games Deephome has hundreds of steps, which will lead to hundreds of observations and actions. It's hard to manipulate all the observations, actions and rewards in unity. So we decided to focus on a part of the environment, the mining part, and show all the difference between our model's walk-through and human walk-through within this specific place.

The Figure 19 is the structure of part of the game world in unity 2D. We can walk on the street and search some buildings or mines. But not all places can get you a reward. For example, in this map, searching ore mine and city generator will be meaningless action with 0 reward.

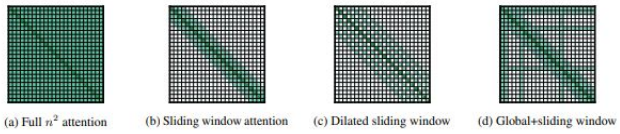

Figure 21: Longformer.

Figure 19: Unity 2D map of the mining part of Deephome game world

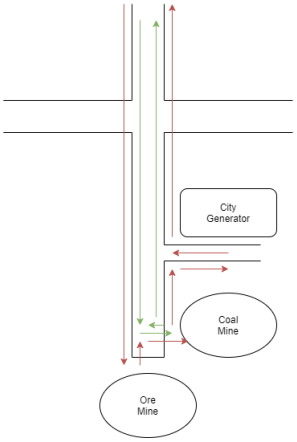

Figure 20: The final Deephome game partial routes of our model output (green) and the standard human player walk-through (red).

7.4 Post-midterm Case Study

Figure 20 shows the partial walk-through routes of our model(green) and the human player(red). Exploring the Ore Mine and City Generator has zero reward. While human player explores every place without reward evaluation, our model ignores these two places and only explore the Coal Mine which has reward. This means our model is able to get evaluate each possible action, then give up the useless action and only choose the reward-able action. This behavior saves lots of time and steps.

References

Prithviraj Ammanabrolu and Matthew Hausknecht. 2019. Graph constrained reinforcement learning for natural language action spaces. In *International Conference on Learning Representations*.

Prithviraj Ammanabrolu and Mark Riedl. 2019a. Playing text-adventure games with graph-based deep reinforcement learning. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 3557–3565.

Prithviraj Ammanabrolu and Mark O Riedl. 2019b. Transfer in deep reinforcement learning using knowledge graphs. *EMNLP-IJCNLP 2019*, page 1.

Wei-Cheng Chang, X Yu Felix, Yin-Wen Chang, Yiming Yang, and Sanjiv Kumar. 2019. Pre-training tasks for embedding-based large-scale retrieval. In *International Conference on Learning Representations*.

Marc-Alexandre Côté, Ákos Kádár, Xingdi Yuan, Ben Kybartas, Tavian Barnes, Emery Fine, James Moore, Ruo Yu Tao, Matthew Hausknecht, Layla El Asri, Mahmoud Adada, Wendy Tay, and Adam Trischler. 2018. Textworld: A learning environment for text-based games. *CoRR*, abs/1806.11532.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186.

Weiwei Guo, Xiaowei Liu, Sida Wang, Huiji Gao, Ananth Sankar, Zimeng Yang, Qi Guo, Liang Zhang, Bo Long, Bee-Chung Chen, et al. 2020. Detext: A deep text ranking framework with bert. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, pages 2509–2516.

Matthew Hausknecht, Prithviraj Ammanabrolu, Marc-Alexandre Côté, and Xingdi Yuan. 2020. Interactive fiction games: A colossal adventure. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 7903–7910.

Ji He, Jianshu Chen, Xiaodong He, Jianfeng Gao, Lihong Li, Li Deng, and Mari Ostendorf. 2016. Deep reinforcement learning with a natural language action space. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1621–1630.

Po-Sen Huang, Xiaodong He, Jianfeng Gao, Li Deng, Alex Acero, and Larry Heck. 2013. Learning deep structured semantic models for web search using clickthrough data. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, pages 2333–2338.

Siddharth Karamcheti Saachi Jain Samuel Humeau Emily Dinan Tim Rocktäschel Douwe Kiela Arthur Szlam Jason Weston Jack Urbanek, Angela Fan. 2019. Learning to speak and act in a fantasy text adventure game.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems*, 32:8026–8037.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2019. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*.

Richard Ziegfeld. 1989. Interactive fiction: A new literary genre? *New Literary History*, 20(2):341–372.