CS230

# First TextWorld Problems

**Renat Aksitov**
raksitov@stanford.edu

## Abstract

In this project I am considering a task of building a reinforcement learning agent for playing text-based games in the TextWorld framework. I am examining several existing architectures and introduce my own hybrid architecture. I then evaluate this architecture and its variations on a small set of games.

## 1 Introduction

Learning to play text games (also known as Interactive Fiction) recently became an important task in the language understanding domain. Text-based games are structured as a "dialog" between a game engine and the player. The engine provides a description of the the current game state in natural language, the player submits text command to execute an action from this state, and the game engine replies back with a feedback on the result of the command. In general in IF **the game state** is only partially observable, because of the inherent ambiguities of the text descriptions, **the action space** is both variable from state to state, and, potentially, unbounded, and **the rewards** are (very) sparse. The goal of the player in text-based games is to find an optimal policy that maximizes the total reward.

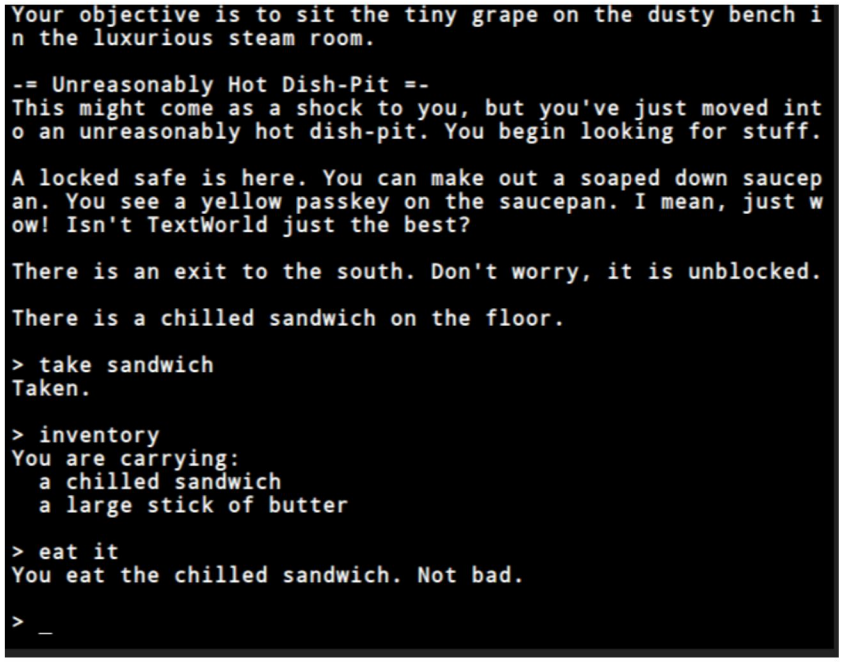


**Figure 1:** An example game from TextWorld, a house-based theme.

For AI agent to play IF games efficiently, it needs to master language understanding, dealing with a combinatorial actions space, efficient exploration, memory and sequential decision-making. As a result, most of the commercially available text-based games are beyond capabilities of the existing algorithmic approaches. To make the text games more approachable and to facilitate research in this area, Microsoft Research recently introduced TextWorld - a learning environment for the training and evaluation of Reinforcement Learning agents on text-based games (1). TextWorld provides a simple API for game interaction, similar to that of OpenAI's Gym, and allows users to setup and automatically generate new games with predefined properties. By using the generated games it is possible to restrict their scope, difficulty and language in a precisely controlled way, and to make them easier for the current learning algorithms than complex human-designed games. Additionally, the generated games could be used for studying generalization and transfer learning in this domain.

To provide more visibility for TextWorld framework in the community and to foster the research into generalization in text-based games, Microsoft Research launched a ML competition during NeurIPS 2018 - *First TextWorld Problems: A Reinforcement and Language Learning Challenge* (2). In this challenge an agent acts within the house, and tries to gather the ingredients in order to cook a delicious meal (Figure 1). The agent must figure out the ingredients from a recipe, explore the house to collect them,

and, once done, go to the kitchen to do the cooking. Additionally the agent will need to learn to open locked doors and to use tools, like knives or frying pans. The house layout, the specific ingredients and their locations vary from game to game, so it is not sufficient to simply memorize a procedure in order to succeed. As my project for the course I am exploring building of a Reinforcement Learning agent that could successfully play IF games from this competition.

## 2 Related work

Due to the difficulties outlined in the previous section, the area of text-based games is covered relatively sparcely in AI research. There are 2 foundational, widely sited papers which propose 2 different architectures for solving IF games.

The first paper (3) describes LSTM-DQN model, which uses an LSTM network for encoding the game state (i.e. the agent's observation) and Deep Q-Network for scoring actions. Authors assume that all the possible actions are known in advance, and have a form of (verb, object) pairs. Q-values for verbs and objects are computed separately, and the final Q-value is the average of the two.

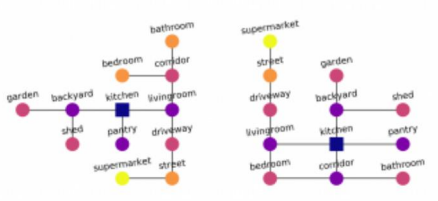


**Figure 2:** Example house layouts generated by TextWorld.

The second work (4) introduces deep reinforcement relevance network (DRRN). DRRN uses 2 separate embeddings for states and actions, then applies feed-forward networks with 1 or 2 hidden layers to the average of corresponding embeddings (bag-of-words representation) to obtain Q-values. The final Q-value is a dot product of states and actions Q-values (Figure 3). This paper employs softmax action-selection as an exploration strategy, instead of more common $\epsilon$-greedy. The key difference from LSTM-DQN is that DRRN could handle much larger action spaces and does not need to know the actions in advance.

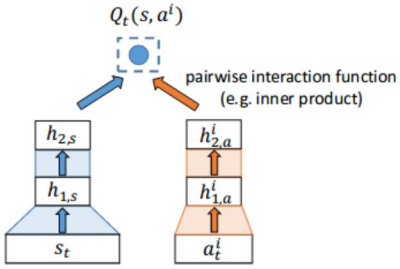


**Figure 3:** DRRN architecture with 2 hidden layers for both states and actions.

The paper (5) draws an inspiration from both LSTM-DQN and DRRN, and proposes Siamese State-Action Q-Network (SSAQN). Similarly to DRRN, this architecture also uses embeddings, but shared instead of separate. It employs a single LSTM as an encoder for both states and actions, and applies cosine similarity to produce a final Q-value.

While DRRN, at least theoretically, allows for an unbounded action space, the exploration over a large amount of possible actions presents a problem for Reinforcement Learning algorithms. In order to solve it, several ideas for reducing the size of action space in text-based games were proposed recently. (6) uses word embeddings, pretrained on Wikipedia, as a common knowledge database, to extract the affordance relations (e.g. $emb(object) - emb(action)$ will encode an affordance vector, that could then be used on the unseen objects to obtain relevant for them actions in the embedding space). As a different approach (7) introduces an action elimination network (AEN) that is trained jointly with DQN and predicts invalid ( redundant or irrelevant) actions, based on a feedback from the game engine.

## 3 Data

The training data is provided by the organizers of the competition and consists of 4440 different games (each game is a data file for the TextWorld framework with the full description of the world). The games in the training set have a different complexity, which is determined by the "skills" that the agent needs to learn in order to perform well in that game. These skills include the number of ingredients in the recipe, actions ("open", "cook", "cut" and "drop") and the number of locations in the game. Several sample games are available in (8).

The First TextWorld Problems allows to submit agents for evaluation on a hidden validation set (with the limit of 5 submissions per day and 100 submissions total). The evaluation games are new games drawn from a distribution similar to the one of the training set. As a result, they do not have any new actions/commands or styles of writing, but they could have new food items, new recipes and new house layouts. The official evaluation is performed by letting the agent play on a set of hidden games, for 10 episodes per game with a maximum of 100 steps per episode. The overall points accumulated during all the episodes of all the games are then adjusted by *handicap* multiplier and reported as an evaluation score.

What exactly is handicap? To make the learning task more approachable, the First TextWorld Problems allows an agent to request additional information for playing the games, besides the game's textual feedback (examples of various kinds of such information are presented in (9)). However, requesting additional information incurs a penalty which depends on the type of information requested (for example, using the output of the "look" and "inventory" commands has a penalty multiplier of 0.85, while using the list of "admissible commands" incurs the largest penalty and has a multiplier of 0.5). If agent requests multiple types of additional information, the highest handicap penalty of the requested set will be applied. All experiments in this project are performed under the condition of the highest handicap (i.e. all additional information is available for the agent).

# 4 Approach

In this section I describe my algorithm (DQN with the experience replay) in general as well as give some specifics of its application to the TextWorld games. I also outline my model structure (a hybrid of LSTM-DQN, DRRN and SSAQN), going over all the architecture choices that I have considered.

## 4.1 Game representation

Let's consider sequential decision making problem, where at each time step $t$ the agent receives 1 or more strings of text with the description $s_t$ of the current hidden state $h_t$ and several strings $a_t^i$ with all possible actions that could be taken from $h_t$. The agent then chooses one of these actions $a_t^{i_0}$ based on the probability $\pi(a_t|s_t)$ (also called policy) with the goal of maximizing future rewards. After that the environment updates it's internal state $h_{t+1} = h'$ based on a probability of a transition $p(h'|h_t, a_t^{i_0})$ and returns a reward $r_t$ at this time step to the agent.

Note, that in the case of the TextWorld games in my dataset, state transitions are deterministic, and a state description $s_t$ could be a union of the actual observation from the environment and all of the additional information requested. In particular, I'm adding outputs of *look* and *inventory* commands, as well as *extra.recipe* to the state. Also, in this setup we do not know all possible actions in advance, so LSTM-DQN can not be applied directly.

## 4.2 Deep Q-Learning

To calculate the full reward (i.e. what the agent needs to maximize) from a time $t$ a standard approach is to use a *discounted* reward with discount $\gamma$:

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... \tag{1}$$

Let's now define the Q-function $Q^\pi(s, a)$ as the expected reward starting from $s$ and taking the action $a$, and after that following a policy $\pi(a|s)$:

$$Q^\pi(s, a) = \mathbb{E}[R_t|s_t = s, a_t = a] \tag{2}$$

To find the optimal Q-function and the corresponding (optimal) policy we can use the Q-learning algorithm with some learning rate $\eta_t$:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \eta_t \cdot (r_t + \gamma \cdot \max_a Q(s_{t+1}, a) - Q(s_t, a_t)) \tag{3}$$

As is, without any generalization, the Q-learning is not very practical for games with large states and actions spaces. The common approach to fix this is to employ *a function approximator* $Q(s, a, \theta) \approx Q(s, a)$. (10) proposes to use a deep neural network as a function approximator and calls such an approximator a Deep Q-Network (DQN). Parameters $\theta$ in this case are the weights of the deep network.

## 4.3 Parameter learning

To learn the parameters $\theta$ we can use stochastic gradient descent. The mini-batches for it are sampled from the "experience replay" buffer, which is a strategy proposed in (10) to avoid issues with strong correlation of the most recent transitions. The buffer is filled during the course of some exploration strategy which agent employs for interacting with the environment. The standard choice is $\epsilon$-greedy, though (4) uses softmax action selection policy. The simplest strategy for sampling from the experience replay is random uniform, but it is not the only choice. For example, (3) proposes to prioritize rare valuable transitions with positive reward, while (11) suggests to prioritize the most recent transitions. The full learning procedure is shown in algorithm 1.

**Algorithm 1** DQN algorithm for TextWorld

Initialize experience replay $\mathcal{D}$
Initialize network parameters $\theta$ with small random weights

1: **for** ($episode = 1$ to $M$) **do**
2: Start the game and obtain a raw text for an initial state and list of admissible actions
3: Process raw state and actions text and convert them into representations $s_1$ and list $[a_1^i]$
4: **for** ($t = 1$ to $T$) **do**
5: Compute list $[Q(s_t, a_t^i, \theta)]$ for the list of actions
6: Select an action $a_t$ from the list based on the chosen exploration policy and $[Q(s_t, a_t^i, \theta)]$
7: Execute the selected action $a_t$ in the environment
8: Observe reward $r_t$ and obtain the new representations $s_{t+1}$ and $[a_{t+1}^i]$
9: Add transition $(s_t, a_t, r_t, s_{t+1}, [a_{t+1}^i])$ to the buffer $\mathcal{D}$
10: Sample $minibatch$ of transitions $[(s_j, a_j, r_j, s_{j+1}, [a_{j+1}^i])]$ from $\mathcal{D}$ according to the chosen sampling strategy
11: Set $y_j = \begin{cases} r_j, & \text{if } s_{j+1} \text{ is terminal,} \\ r_j + \gamma \max_i Q(s_{j+1}, a_{j+1}^i, \theta), & \text{otherwise.} \end{cases}$
12: Perform a gradient descent step on the loss $\mathcal{L}(\theta) = (y_j - Q(s_j, a_j, \theta))^2$
13: **end for**
14: **end for**

### 4.4 Network architecture

Following papers (3), (4) and (5), the architecture of my deep network consists of word embeddings layer, state and actions encoder layers and the final interaction function. There are multiple choices to explore within this generic architecture:

- embeddings could be trainable or not (trainable work much better for me), pre-trained or randomly initialized (the latter outperforms heavily in my Textworld experiments),
- each encoder could be either a feed-forward network (DRRN) or recurrent (LSTM-DQN),
- both embeddings and encoders could be either shared (SSAQN) or separate (DRRN),
- the interaction function could be inner product (DRRN) or cosine similarity (SSAQN) (the performance is about the same for me),
- we can apply mean-pooling to the outputs of the recurrent network (LSTM-DQN) or just treat the last output as an encoding (mean pooling does work better in my experiments).

The full architecture search and my final architecture are presented in the next section.

## 5 Results

### 5.1 Experiments setup

Originally, I was planning to use an official evaluation mechanism described in the Data section, for benchmarking my models, but it, unfortunately, ended up being infeasible. It turned out, that the speed of training for the family of models I'm looking into, is too slow for me to be able to train on multiple games at a time, let alone on some meaningful subset of 4000+ training games. Specifically, my original implementation was training for more than an hour even on the easiest games (those, that could be occasionally won by a random agent), and it would have taken more than a day for the games of moderate difficulty. I have implemented multiple modifications for speeding up the training (discussed in the next subsection), but, even with all the improvements, the training on just a single game still easily takes couple hours. As a result, I have abandoned the idea to train on the whole training set of the competition and to measure generalization on unseen games. Instead, for my project I am limiting all the experiments to a small subset of games in the training data.

### 5.2 Reducing training time

Arguably, the (very slow) speed of training was the main issue that I encountered in my project. I think it is for the most part caused by the structure of the DQN-algorithm, i.e. the necessity to constantly run the inferences in the graph to calculate Q-values (line 11 in algorithm 1). Overall, there were multiple things that I was able to implement in order to speed up training. I have started from eliminating the most obvious inefficiencies, like moving all the broadcasting into the graph and caching already seen states / actions (to avoid tokenization/conversion to ids of the same string multiple times). Then I implemented batching of all Q-values in the minibatch into the one computation, this part had the biggest impact on performance. Another change that provided large boost was switching to experience replay that prioritizes both positive rewards and recent transitions. Interestingly, the most direct solution of increasing the batch size for training, did not produce good results for me: the time of training stayed

about the same, while the quality of the trained agent dropped. At the end, with all the modifications I made I was able to speed up my training more than 10x.

### 5.3 Hyper parameters search

It is crucial in Reinforcement Learning not to overfit to a single environment with hyper parameters. To make sure this is the case, I did most of my tuning on the easy games and then just used the best parameters for the more difficult ones. One essential idea, that allowed me to have the same hparams for all games was using an "adaptive epsilon" in the $\epsilon$-greedy: the epsilon is changed based on the mean rewards achieved recently and as a result the exploration automatically adjusts to the difficulty of the specific game. The results of my hparam search are summarized in table 1. Two of the most important parameters here were $\gamma$, which performs significantly worth for larger values, and the size of the replay buffer (larger buffers cause divergence much more easily).

| Hyper Parameter | Value | Search Space |
|---|---|---|
| Optimizer | Adam | { Adam; RMSProp } |
| Learning rate | 0.001 | { 0.1 .. 0.00001 } |
| Dropout | 0.9 | { 0.5 .. 1.0 } |
| Gamma | 0.5 | { 0.4 .. 1.0 } |
| Replay buffer | 10,000 | { 1,000 .. 100,000 } |
| RNN encoder | GRU | { GRU; LSTM } |
| RNN hidden size | 128 | { 32 .. 256 } |
| Grad norm | 5.0 | { None; 1.0 .. 10.0 } |

**Table 1:** Final hyper parameters

### 5.4 Architecture search

I did very intensive search in the architecture space, which I will try to summarize in this subsection. I have started with implementing DRRN model as close to the presented in the paper as possible (DRRN-BASIC model). In it I used 2 feed forward layers for the context encoder and 1 for the action encoder. Then I replaced the basic replay buffer with prioritized by rewards and recency buffer (DRRN-PRIOR). The next model that I've tried was replacing feed forward encoders in DRRN-PRIOR with RNN encoders, followed by a dense layer (DRRN-GRU-BOTH). I also tried to share embeddings and / or encoders weights, as SSAQN paper advises, but sharing did not perform well in my experiments. The most successfull model of this form was DRRN-PRIOR with shared embeddings (DRRN-SHARED-EMB). Finally, the best performing model that I found is DRRN-PRIOR with only state encoder replaced by RNN encoder and action encoder left as feed forward (DRRN-GRU-STATE).

### 5.5 Evaluation

For evaluation, I've chosen 3 games of varying difficulty and trained several architectures on each game for at most 1000 episodes (10 parallel games per episode). After convergence or completing 1000 episodes, I'm evaluating the model on the same game and report the mean score over 10 episodes. The results are presented in table 2. Each game column shows the max possible reward in the first row. All the models that were able to converge on a specific game will have the same cumulative reward, so it is more meaningful to compare them by looking into the next two columns: number of episodes till convergence and the training time till convergence (these are computed on the easy game, where all the models converge)

| Model architecture | Easy (3) | Moderate (6) | Hard (10) | Episodes | Time, sec |
|---|---|---|---|---|---|
| Random baseline | 1.2 | 0.8 | 0.3 | n/a | n/a |
| DRRN-BASIC | 3.0 | 2.5 | 1.5 | 120 | 362 |
| DRRN-PRIOR | 3.0 | 5.2 | 2.5 | 52 | 156 |
| DRRN-GRU-BOTH | 3.0 | 4.5 | 4.2 | 15 | 297 |
| DRRN-SHARED-EMB | 3.0 | 6.0 | 3.1 | 27 | 80 |
| DRRN-GRU-STATE | 3.0 | 6.0 | 8.0 | 8 | 245 |

**Table 2:** Performance results and comparisons

## 6 Conclusions

During my project, I explored various architectures for playing text-based games in the TextWorld framework. I have started with the architectures available in the literature (LSTM-DQN, DRRN and SSAQN) and proceeded to combine pieces from them into the hybrid model, that could successfully solve many single games from the First TextWorld Problems competition. I also performed extensive architecture search, in order to come up with my hybrid model.

I had several ideas, that I did not have the time to implement/explore, that might be worse trying in the future. First one is, of course, generalization. This requires either more significant computational resources or, maybe, a different RL algorithm. Two simpler things to try is using target network in DQN to improve convergence and exploring various NLP ideas to be able to drop admissible commands and reduce handicap. Finally, my success with using adaptive epsilon makes me wonder whether it could be feasible to have auxiliary network that will be predicting level of exploration required at the moment.

*The code of the project is available in (12)*

## References


[1] Côté, M.A., Kádár, Á., Yuan, X., Kybartas, B., Barnes, T., Fine, E., Moore, J., Hausknecht, M., Asri, L.E., Adada, M. and Tay, W., 2018. **Textworld: A learning environment for text-based games**. *arXiv preprint arXiv:1806.11532*.

[2] https://competitions.codalab.org/competitions/20865#learn_the_details

[3] Narasimhan, K., Kulkarni, T. and Barzilay, R., 2015. **Language understanding for text-based games using deep reinforcement learning**. *arXiv preprint arXiv:1506.08941*.

[4] He, J., Chen, J., He, X., Gao, J., Li, L., Deng, L. and Ostendorf, M., 2015. **Deep reinforcement learning with an unbounded action space**. *arXiv preprint arXiv:1511.04636, 5*.

[5] Zelinka, M., 2018. **Using reinforcement learning to learn how to play text-based games**. *arXiv preprint arXiv:1801.01999*.

[6] Fulda, N., Ricks, D., Murdoch, B. and Wingate, D., 2017. **What can you do with a rock? affordance extraction via word embeddings.** *arXiv preprint arXiv:1703.03429.*

[7] Zahavy, T., Haroush, M., Merlis, N., Mankowitz, D.J. and Mannor, S., 2018. **Learn what not to learn: Action elimination with deep reinforcement learning.** *In Advances in Neural Information Processing Systems (pp. 3566-3577).*

[8] https://github.com/raksitov/TextWorld/tree/master/sample_games

[9] https://github.com/raksitov/TextWorld/blob/master/data_exploration.ipynb

[10] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. and Riedmiller, M., 2013. **Playing atari with deep reinforcement learning.** *arXiv preprint arXiv:1312.5602.*

[11] Zhang, S. and Sutton, R.S., 2017. **A deeper look at experience replay.** *arXiv preprint arXiv:1712.01275.*

[12] https://github.com/raksitov/TextWorld/models