

**FACULTY
OF MATHEMATICS
AND PHYSICS**
Charles University

MASTER THESIS

Bc. Mikuláš Zelinka

**Using reinforcement learning to learn
how to play text-based games**

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis: Mgr. Rudolf Kadlec, Ph.D.

Study programme: Informatics

Study branch: Artificial Intelligence

Prague 2017

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In date

signature of the author

Title: Using reinforcement learning to learn how to play text-based games

Author: Bc. Mikuláš Zelinka

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Rudolf Kadlec, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: The ability to learn optimal control policies in systems where action space is defined by sentences in natural language would allow many interesting real-world applications such as automatic optimisation of dialogue systems. Text-based games with multiple endings and rewards are a promising platform for this task, since their feedback allows us to employ reinforcement learning techniques to jointly learn text representations and control policies. We present a general text game playing agent, testing its generalisation and transfer learning performance and showing its ability to play multiple games at once. We also present pyfiction, an open-source library for universal access to different text games that could, together with our agent that implements its interface, serve as a baseline for future research.

Keywords: reinforcement learning, text games, neural networks

Firstly, I am very grateful to the Interactive Fiction community for their helpful and kind responses to my questions about IF games.

Secondly, I would like to thank my supervisor, Rudolf Kadlec, for his kindness and for broadening my horizons by offering inspirational and valuable insights.

Finally, my sincerest gratitude belongs to my friends (especially to the very nitpicky friend for his thorough comments), and to my whole family for being so incredibly supportive, patient and encouraging.

Contents

Introduction	3
1 Text-based games	5
1.1 Game structure	5
1.2 Genres and variants	6
1.3 Rewards	7
1.4 Game properties and their influence on task difficulty	8
1.4.1 Vocabulary size	8
1.4.2 Action input type	8
1.4.3 Game size	8
1.4.4 Cycles	9
1.4.5 Completeness and hidden game states	10
1.4.6 Deterministic and non-deterministic properties	11
1.5 Related problems and human performance	12
1.6 Unifying access to various text games	13
2 Background	14
2.1 Core definitions	14
2.1.1 Text games	14
2.1.2 Markov decision process	15
2.1.3 Learning task	16
2.2 Reinforcement learning	17
2.2.1 Updating the policy	17
2.2.2 Q-learning	18
2.3 Neural networks	20
2.3.1 Gradient descent	20
2.3.2 Recurrent networks	21
2.4 Language representation	23
2.4.1 Text preprocessing	23
2.4.2 Embeddings	23
2.5 Summary	24
3 Agent architecture	25
3.1 Related models	25
3.1.1 LSTM-DQN	25
3.1.2 DRRN	25
3.2 Motivation	26
3.3 SSAQN	27
3.3.1 Word embeddings	29
3.3.2 LSTM layer	29
3.3.3 Dense layer	29
3.3.4 Interaction function	30

3.3.5	Loss function and gradient descent	30
3.4	Action selection	31
3.5	Training loop	32
3.6	Technical details	33
3.6.1	Parameters	33
3.6.2	Scaling the Q-values	33
3.6.3	Estimating multiple Q-values effectively	33
3.7	Summary	34
4	Experiments	35
4.1	Setup	35
4.1.1	Games	35
4.1.2	Game simulator properties	37
4.1.3	Evaluation	37
4.1.4	Parameters	38
4.2	Individual games	38
4.2.1	Setting	39
4.2.2	Results	39
4.2.3	Discussion	41
4.3	Generalisation	42
4.3.1	Setting	42
4.3.2	Results	43
4.3.3	Discussion	44
4.4	Transfer learning	45
4.4.1	Setting	46
4.4.2	Results	46
4.4.3	Discussion	46
4.5	Playing multiple games at once	47
4.5.1	Setting	48
4.5.2	Results	48
4.5.3	Discussion	49
	Future work	51
	Conclusion	52
	Bibliography	53
	List of Figures	57
	List of Tables	57
	List of Abbreviations	58
	Appendices	59
	Appendix A Text games	60
	Appendix B pyfiction	64

Introduction

The process of learning to understand and reason in natural language has always been near the centre of attention in Artificial Intelligence (AI) research. One of the recently explored tasks in language understanding whose successful solving could have a big impact on learning to comprehend and respond in dialogue-like environments (Jurafsky and Martin [2009]) is the task of playing text-based games.

In text games, which are also known as Interactive Fiction (IF, Montfort [2005]), the player is given a description of the game state in natural language and then chooses one of the actions which are also given by textual descriptions. The executed action results in a change of the game state, producing new state description and waiting for the player’s input again. This process repeats until the game is over.

More formally speaking, text games are sequential decision making tasks with both state and action spaces given in natural language. In fact, a text game can be seen as a dialogue between the game and the player and the task is to find an optimal policy (a mapping from states to actions) that would maximise the player’s total reward. Consequently, the task of learning to play text games is very similar to learning how to correctly answer in a dialogue.

Usually, text games have multiple endings and countless paths that the player can take. It is often clear that some of the paths or endings are better than others and different rewards can be assigned to them. Availability of these feedback signals makes text games an interesting platform for using reinforcement learning (RL), where one can make use of the feedback provided by the game in order to try and infer the optimal strategy of responding in the given game, and potentially, in a dialogue.

Recently, there have been successful attempts (He et al. [2015], Narasimhan et al. [2015]) to play IF games using RL agents. However, the selected games were quite limited in terms of their difficulty and even more importantly, the resulting models had mostly not been tested on games that had not been seen during learning.

While being able to learn to play a text game is undoubtedly a success in itself, we should keep in mind that in order for the resulting model to be useful, it must generalise well to previously unseen data. In other words, we can merely hypothesise that a successful IF game agent can at least partly understand the underlying game state and potentially transfer the knowledge to other, previously unseen, games, or even natural dialogues. And for the most part, it remains to be seen how the RL agents presented in He et al. [2015] and Narasimhan et al. [2015] perform in terms of generalisation in the domain of IF games.

To summarise, IF games provide a very large and interesting platform for language research, especially when using reinforcement learning. Some attempts have been made at learning to play them using RL techniques, however, there has not been enough evidence that the resulting models do indeed understand the text and that they can generalise to new games or dialogues, which is what would make the models useful in real-life applications.

In this work, we mainly focus on the following:

- We present **pyfiction**¹, an open-source library that enables researchers to universally access different IF games and integrates into OpenAI Gym².
- We employ reinforcement learning algorithms to create a general agent capable of learning to play IF games. The agent is a part of the pyfiction library and can consequently be used as a baseline for future research.
- We explore what properties the resulting models have and how they perform in terms of generalisation.

The structure of the thesis is as follows.

In chapter 1, we describe the domain of text-based games and their variants, emphasising the influence of different game attributes on the learning task difficulty.

In chapter 2, we formally define the text-game learning task and review the methods commonly used for similar problems, including reinforcement learning, neural networks and techniques for language representation.

In chapter 3, we introduce our text-game playing agent with a general architecture and compare the agent to the ones presented in He et al. [2015] and Narasimhan et al. [2015].

In chapter 4, we test the agent on single-game and multiple-games learning tasks and conduct experiments focused on its transfer learning and generalisation abilities.

¹See appendix B or <https://github.com/MikulasZelinka/pyfiction>.

²A platform for evaluating RL agents: <https://gym.openai.com/>, Brockman et al. [2016].

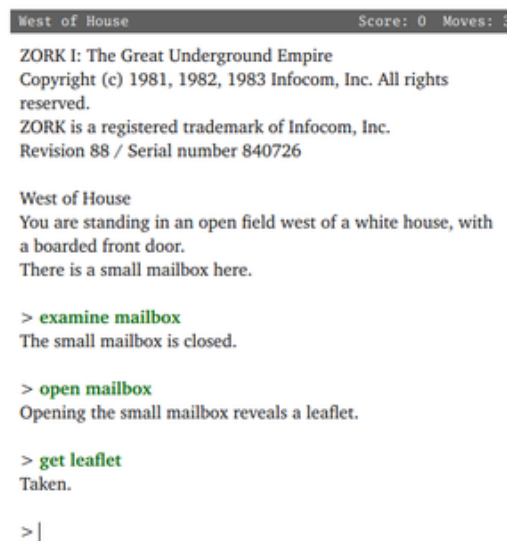
1. Text-based games

In this chapter, we introduce the domain of text-based games, commonly known as **interactive fiction** (IF). We describe the properties of IF games that influence the difficulty of the learning task.

We also present **pyfiction**, a library that allows convenient access to different IF games for research purposes.

1.1 Game structure

IF games typically use a basic input-output loop of providing a text description of the game state, waiting for player's action, updating the game state according to the chosen action and responding with a new text description of the new game state. This then continues until a final state is reached.



```
West of House                               Score: 0 Moves: 3

ZORK I: The Great Underground Empire
Copyright (c) 1981, 1982, 1983 Infocom, Inc. All rights
reserved.
ZORK is a registered trademark of Infocom, Inc.
Revision 88 / Serial number 840726

West of House
You are standing in an open field west of a white house, with
a boarded front door.
There is a small mailbox here.

> examine mailbox
The small mailbox is closed.

> open mailbox
Opening the small mailbox reveals a leaflet.

> get leaflet
Taken.

>|
```

Figure 1.1: *Zork I*, a classic IF game¹.

There are exceptions to the basic loop in several games such as visual-based puzzles or logical minigames. In this work, though, we focus on simpler games without similar elements.

Nevertheless, such games will likely present interesting challenges in the future.

Additionally, we simplify the task by ignoring the occasional extra resources such as images, sounds or other complementary game assets that enhance the player experience but are not necessary for the artificial agent's learning process.

¹Source: https://en.wikipedia.org/wiki/File:Zork_I_screenshot_video_game_Gargoyle_interpreter_on_Ubuntu_Linux.png.

1.2 Genres and variants

While the output of the game simulator is almost always a text description, the form of input that the game expects does vary. One of the most common criteria for classifying IF games is based on their accepted types of text input (for examples, see figure 1.2):

- *parser-based*, where the player types in any text input freely,
- *choice-based*, where multiple actions to choose from are typically available, *in addition* to the state description,
- *hypertext-based*, where multiple actions are present as clickable links *inside* the state description.



Figure 1.2: Parser, choice and hypertext-based games (He et al. [2015]).

This classification is mentioned mainly because it has significant implications for our task. There is a plethora of other criteria that can be used for classification of different types of IF games. The largest source of information on IF games and their variants is the Interactive Fiction Database² which features useful filters, tags and lists for finding specific kinds of games. For our purposes, though, the differences in game genre or storytelling elements do not largely matter.

Text games come in virtually all genres, often with different minigames or puzzles incorporated into them. As mentioned earlier, we will mostly deal with games without any additional special parts that do not fit into the general and simple input-output loop described above.

In this work, we only deal with *choice-based* and *hypertext-based* games but both variants are referred to as *choice-based*, as the hyperlinks are simply considered additional choices.

In fact, note that even all parser-based games with finite number of actions can be converted to choice-based games by simply enumerating all the actions accepted by the interpreter at different time steps. This trick was used by Narasimhan et al. [2015] where a subset of all possible action combinations was presented as the possible choices to the agent.

²IFDB: <http://ifdb.tads.org/>.

1.3 Rewards

IF games do often have multiple endings which are based on the player’s choices. These games can be found on the IFDB² under the `multiple endings` or `cyoa` (choose your own adventure) tags.

In some games, explicit numerical rewards are present to reflect the player’s performance. One example of such game is *Six*³ by Wade Clarke in which the player is rewarded after “tagging” one of the six children during the Tag playground game.

In other games, the feedback comes in purely textual form, saying how well the player did. The most extreme cases of distinct endings are usually the classic “*You live happily every after.*” and “*You died.*”. There are games with almost any number of endings, but importantly, if we define the ending as the last received state description, there are even games with thousands or more endings.

One example of such game is *Star Court*⁴ by Anna Anthropy, where in a specific ending, the player is sentenced to spend a variable — randomly generated — number of years in prison.

Similarly, in *Machine of Death*⁵ by Hulk Handsome, one ending includes a recapitulation of player’s recent choices, resulting in an exponential number of endings in the number of mentioned choices.

Assigning numerical rewards

In order to make use of the amount of feedback present in various text games, He et al. [2015] manually assigned numerical rewards to various endings of the games *Saving John* and *Machine of Death*, trying to quantify how well the player did to reach each specific ending.

We use the same rewards for these two games that are also used in our experiments. Additionally, we repeat the process of annotating game endings for different games (see appendix A for details).

Reward distribution

Note that we have only mentioned annotating different *endings*, and not states in general.

This is because in *Saving John*, *Machine of Death* as well as in other games we used, it corresponds to the nature of the game.

For all other non-terminating game states, the player is given a small negative rewards in order to encourage the agent to learn to play efficiently. Altogether, the rewards built into the games as a feedback enable us to use reinforcement learning effectively for this task.

In general, though, we hypothesise that games like *Six* with more densely distributed rewards should be easier to learn thanks to their presence throughout the whole game.

³Six: <http://ifdb.tads.org/viewgame?id=kbd990q5fp7pythi>.

⁴Star Court: <http://ifdb.tads.org/viewgame?id=u1v4q16f7gujdb2g>.

⁵Machine of Death: <http://ifdb.tads.org/viewgame?id=u212jed2a7ljg6hl>.

1.4 Game properties and their influence on task difficulty

IF games obviously differ immensely in terms of both their content and presentation. These differences are hard to capture and it is also unclear how they impact the learning process. Here, we focus on some of the quantifiable properties of IF games that contribute to the overall difficulty of the learning task.

1.4.1 Vocabulary size

By vocabulary size we mean the number of unique words — tokens — used throughout the game. In natural language processing (NLP) tasks, the tokens are commonly converted to word indices and consequently, a single sentence is represented as a vector of these indices. Vocabulary size then corresponds to the number of different word indices or to the maximum word index.

Unsurprisingly, larger vocabulary results in a more difficult task. In order to simplify the problem, techniques that reduce the vocabulary size such as *stemming* or *lemmatisation* are commonly used in NLP tasks.

The core idea behind these algorithms is to identify and unify semantically identical or similar words with different syntactic expressions, in particular synonyms.

In our case, the individual IF games are relatively small and usually only contain thousands of unique tokens at most. Since the amount of data does not present a problem, we only employ few simple rules to help reduce the vocabulary size (see section 2.4.1 for more details).

1.4.2 Action input type

The types of input accepted by IF games are described in section 1.2.

Their impact on the task difficulty is straightforward; parser-based games require the player to generate their input, resulting in unbounded action space.

On the other hand, choice-based and hypertext-based do not require any text generation and it is only necessary to evaluate the present actions to learn to play the games.

While most parser-based games accept only a finite number of sentences and consequently can be made much easier by enumerating all actions accepted by the game interpreter, their original version presents a great challenge for the future.

A different direction of research might be learning from IF manuals or experienced players, similar to how Branavan et al. [2014] learned to play the strategy game *Civilisation* using game manuals.

1.4.3 Game size

The sheer number of underlying game states the player can visit and the number of actions they can take has an increasing impact on the task difficulty.

In particular, the branching factor (the average number of actions resulting in different states) is a very important metric. Consider an agent that is randomly

exploring the state space. Now if the agent has thirty actions to take of which only one leads to a positive reward, it is highly unlikely that the exploration policy will find the reward. This problem can be further highlighted if these states with a high number of possible actions are themselves located further into the game, meaning even reaching the difficult decision state is unlikely.

Moreover, if rewards are only given to the player at the end of each game episode, as is the case in the games we utilise for training the agents, longer paths to the endings with more states result in much higher difficulty due to how sparse the rewards are.

1.4.4 Cycles

If we imagine an IF game as an oriented graph with nodes corresponding to the game states and actions representing the edges between the nodes (see figure 1.3), we can identify some interesting properties of the graph relevant to the task difficulty.

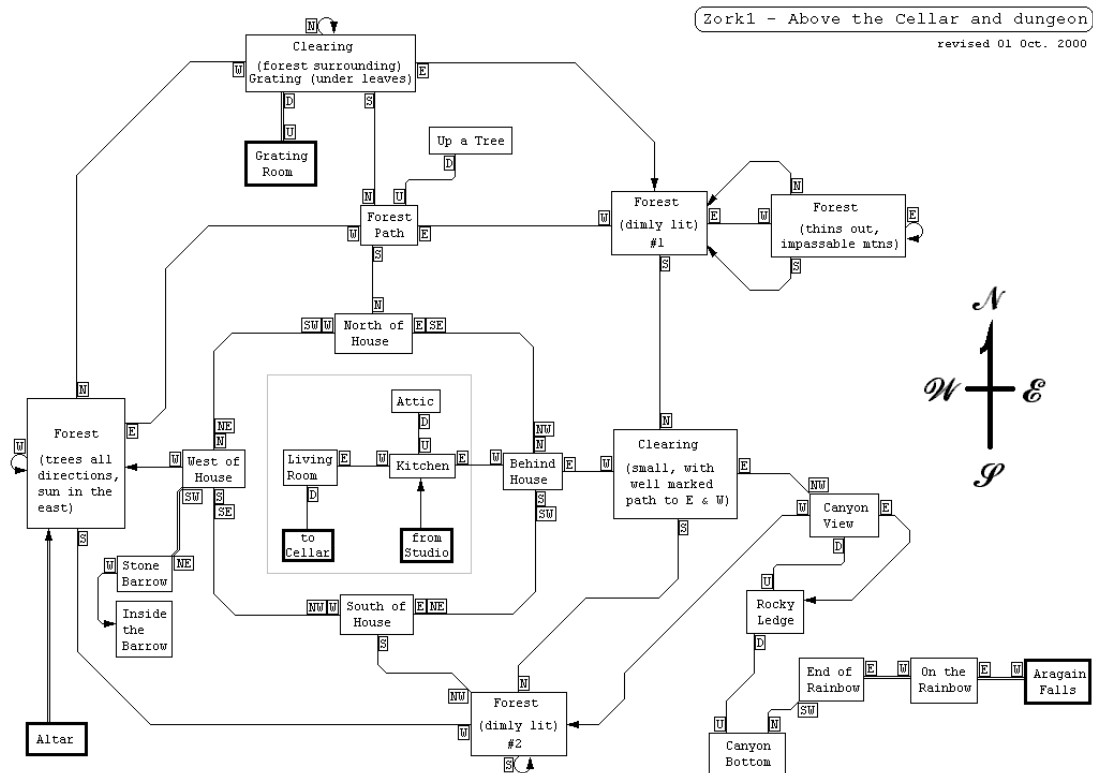


Figure 1.3: Part of *Zork I*'s game map⁶.

The most important feature is the presence or absence of cycles in the graph. In other words, if we have an agent with a deterministic policy, is it possible that it will never finish the game?

Interestingly, in most IF games, the answer to this question is yes, ergo there is a number of cycles in which a reflexive agent (an agent without memory) can get stuck. The most common example of such cycle is a transition between two

⁶Source: <http://www.lafn.org/webconnect/mentor/zork/zorkText.htm>.

rooms with a bidirectional connection, where the agent would choose `go north` and `go south` respectively, resulting in an infinite loop.

There are other important metrics, such as the average number of actions per state or perhaps the ratio of meaningful actions (actions that do further progress the story). These features do impact the task difficulty, although not as severely as cycle presence. While cycles can make it impossible for a simple agent to learn or at least completely explore the state space, the number of actions and the ratio of “good” and “bad” actions should only impact the speed of the learning process.

1.4.5 Completeness and hidden game states

In simple games, the text descriptions of states given to the player at any time contain all necessary information and thus provide complete game state to the player.

However, it is fairly common, especially in more complex games, that the game is making use of an underlying engine that produces descriptions that describe the state of the game world only partly. We call the underlying and complete game state the *hidden state* and the description the player can see is referred to as the *visible state*.

There are two common principles that result in the presence of hidden states.

Firstly, the game engine may be keeping track of important statistics such as player’s health and their inventory, or of the state and location of various game objects that the player can potentially interact with. In order to find out what is in the player’s inventory, for example, the player has to select an action resulting in inventory description that is completely separate from the world description received in the last step. We say that the game provides *incomplete* state descriptions.

To play these games successfully, an agent with internal memory or at least an agent that takes into account a history of the received game descriptions is required.

Secondly, games often make use of random text descriptions for the same underlying state. For example, in *Star Court*, the player is told that they took a specific job, but the job description varies based on few randomly selected options. For the game engine, however, all the jobs essentially represent the same game state and the point of the random descriptions is to add some flavour to the game and increase its replay value.

Implications of non-deterministic descriptions on task difficulty are discussed in section 1.4.6.

As we will see in the next chapter, the additional complexity coming from the incompleteness of state descriptions is rather significant, as it basically changes the underlying task from solving simple Markov decision processes (MDP, see section 2.1.2) to solving partially observable MDPs, which are typically intractable (Monahan [1982]).

1.4.6 Deterministic and non-deterministic properties

There is a varying degree of randomness in IF games. Some, like *Saving John*, have no random elements and are purely deterministic. These games can be solved by simple graph searching techniques such as breadth-first or depth-first search. In fact, even an agent that behaves randomly, only remembers its best path and iteratively plays the game will eventually find an optimal solution to such game.

Other games, for example *Machine of Death*, can be random, resulting in a much more difficult task. We identify two possible random elements to the IF games.

State and action descriptions

In some games, a single room, a single game state or a single action is always described using the exact same words; in others, the description can be generated randomly, resulting in different descriptions for the same inner states expressing the same meaning. For example, a state could be randomly described as either “*There is a glass on the table.*” or “*On the table, you notice an empty glass.*”

The most obvious and a very important implication of such functionality is that the agent cannot easily tell if two states are the same. As a consequence, these games can no longer be solved by simple graph search algorithms as one cannot determine if two nodes are really the same.

In particular, the agent now only observes visible states, different from the underlying hidden states. In section 1.4.5, we mentioned that hidden states, where the complete description of the game world consists of *multiple and separate* descriptions, present a challenge. However, in this case we only have a *single* hidden state expressed in different ways.

Consequently, in the case of random descriptions, no memorisation is explicitly required. Even an agent that does not account for these hidden states can theoretically learn to play games with random descriptions, as it can potentially either learn to handle a higher number of varying state descriptions (that represent a single hidden state capturing the whole game world) or even learn — from context — that a set of states really represents the same underlying state.

State-action transitions

Randomness can also lie in the way transitions from states using specific actions work. Selecting an action a in state s might either always result in the same state (deterministic behaviour); or in different states depending on the non-deterministic game engine (non-deterministic behaviour).

It appears that most games, or more precisely most state-actions tuples in most games, are deterministic. In other words, even in games where some transitions are random (e.g. *Machine of Death* or *Star Court*), most transitions are deterministic.

Non-deterministic behaviour of the modelled environment is implicitly accounted for by the theory of MDPs and does not significantly increase the theoretical learn-

ing task difficulty — or more precisely, the tractability of the problem. Nonetheless, the games we use for training our agents do differ in terms of the presence of random transitions and this additional complexity is very noticeable as the convergence of the learning algorithms is much slower on the non-deterministic games in practice (see section 4.2 for the results of learning individual games).

1.5 Related problems and human performance

Even though humans would typically find it easier to play text games than some of the classic visual-based games, the opposite is interestingly true for RL agents. Most notably, Mnih et al. [2013] and Mnih et al. [2015] reached human-like or even superhuman performance on classic, visual-based games from the Atari platform, whereas He et al. [2015] showed that humans significantly outperform their agent in both presented text games.

These results are not surprising as there has undoubtedly been less research towards RL for text-game playing agents than towards agents playing visual-based games.

However, we hypothesise that the main factor behind this gap is that the challenge in the majority of Atari games comes mostly from having to react fast, or more precisely, the lack of the ability to plan can be compensated for by very fast reactions in a lot of games, which is obviously very easy for artificial agents.

In text games, on the other hand, reaction speed is not a factor and the main challenge, besides planning in more complex IF games, lies in understanding natural language.

Additionally, in He et al. [2015], not only did human players outperform the virtual IF agents significantly but their sample efficiency was also incomparably higher. While the human players managed to learn to play the games in only few tries, the agents took at least thousands of iterations to reach acceptable performance.

1.6 Unifying access to various text games

There are different formats, engines and interpreters used to create and play IF games. To our knowledge, there is currently no universal interface to the various text-game types, making it difficult to try learning to play multiple games or conduct experiments related to generalisation that would require larger sets of games. Narasimhan et al. [2015] and He et al. [2015] also used different games based on different engines for their experiments, making it troublesome to compare the results.

It is important for natural language processing (NLP) tasks, including the task of learning to play IF games, to be able to work with large amounts of data and cover as much of the language space as possible due to the complexity and the unbounded nature of human language.

To help address this issue, we present **pyfiction**⁷, a Python library whose aim is to enable researchers to simplify access to IF games of various types or formats. Among others, the library currently supports games *Saving John* and *Machine of Death* used in He et al. [2015]. See appendix A for a detailed description of supported games used in this thesis and available for general purposes.

The library is easily extensible and also features support of any games written in **Inform 7**⁸. and runnable by the **Glulxe**⁹ interpreter by providing a Python interface to the interpreter.

Since most tools for creating IF games also support exporting to the web-based HTML format, pyfiction features a Python-based HTML simulator with several example games.

All IF game simulators provided by pyfiction share the same I/O interface, making it very simple to test an agent across a number of games based on different engines.

Additionally, in order to provide even more universal interface, we plan to integrate the games supported by pyfiction into OpenAI Gym (Brockman et al. [2016]), a general framework for evaluating RL agents on games of various genres.

For more details about pyfiction and its game interface, refer to appendix B or to the library website⁷.

⁷See <https://github.com/MikulasZelinka/pyfiction>.

⁸A system for designing IF games, see <http://inform7.com>.

⁹Interpreter of Z-machine game files, see <https://github.com/erkyrath/glulxe>.

2. Background

In this chapter, we define the task of playing text-based games and briefly review the methods and tools we utilise for building our agent.

2.1 Core definitions

Now that we have a clear idea about what properties text-based games have (see chapter 1 for their detailed description), we can define the concept and subsequently the task of solving the games more formally.

2.1.1 Text games

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

Definition 2.1.1 (text game).

Let us define a text game as a tuple $G = \langle H, H_t, S, A, \mathcal{D}, \mathcal{T}, \mathcal{R} \rangle$, where

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

Generally speaking, both the transition function \mathcal{T} and the description function \mathcal{D} may be stochastic and as mentioned in section 1.4, the properties of the game and its functions have a great impact on the task difficulty.

In particular, if both \mathcal{D} and \mathcal{T} are deterministic, the *whole game* is deterministic. Consequently, as we discussed in section 1.4.6, the problem is then reduced to a simple graph search problem that can be solved by graph search techniques and there is no need — from the perspective of finding optimal rewards — to employ reinforcement learning methods.

However, from the generalisation perspective, it still might be useful to attempt to learn simple games using RL techniques as the agents could potentially be able to generalise or transfer their knowledge to other, more difficult problems — which certainly does not apply to the graph search algorithms.

2.1.2 Markov decision process

We formalise the problem of playing text games as the task of solving a *Markov decision process* (MDP), where the parameters of the MDP correspond to the parameters of a text game as defined in 2.1.1.

Definition 2.1.2 (Markov decision process).

MDP is a stochastic process defined as a tuple $\mathcal{M} = \langle S, A, \mathcal{P}, \mathcal{R}, \gamma \rangle$, where

- S is a set of states,
- A is a set of actions,
- $\mathcal{P}(s_{t+1}|s_t, a_t)$ is the probability that choosing an action a_t in state s_t will result in state s_{t+1} ,
- $\mathcal{R}(s_t, a_t, s_{t+1})$ is the immediate reward function that assigns a numerical reward to a state transition,
- $\gamma \in [0, 1]$ is the discount factor determining preference of immediate or future rewards.

The transition function \mathcal{P} has the *Markov property*, meaning that the result of the state update only depends upon the present state and not upon the state history.

In other words, \mathcal{P} is a function of the last state only, as also seen in the definition.

Definition 2.1.3 (Markov property).

Function $\mathcal{P} : (S \times A) \rightarrow S$ is said to possess the Markov property if:

$$\mathcal{P}(s_{t+1}|s_t, a_t, s_{t-1}, \dots, s_0) = \mathcal{P}(s_{t+1}|s_t, a_t).$$

In our case, a text-game MDP is simply an MDP where the states of the MDP correspond to the game state descriptions and the same applies for actions and rewards.

The probability function $\mathcal{P}(s'|s, a)$ is then realised by the game transition function \mathcal{T} and crucially, it is unknown to the agent.

The terminal states of a text game can be implicitly encoded in its MDP as states with a deterministic transition to themselves and a zero reward.

The γ parameter is commonly called a *discount factor* and it determines the weights of rewards at different time steps. Lower values lead to preference of immediate rewards whereas higher values result in taking the future into account more. This parameter plays an important role in determining the optimal behaviour of the agent through influencing the definition of the cumulative reward as seen in equation 2.2.

Notice that not all IF games necessarily have the Markov property, i.e. in text games, there can be long-term dependencies. In fact, it is even difficult to determine whether a text game is or is not Markovian.

However, even problems with non-Markovian characteristics are commonly represented and modelled as MDPs while still giving good results (Sutton and Barto [1998]).

2.1.3 Learning task

We define the task of learning to play a text game as the process of finding an optimal policy for the respective text game MDP.

Policy is a function, usually realised by a set of decision-making rules, that takes a state as an input and produces an action as a result. We denote that applying a policy π in state s results in action a by writing:

$$a \leftarrow \pi(s). \quad (2.1)$$

The **discounted cumulative reward** that the agent receives is defined as

$$\sum_{t=0}^{\infty} \gamma^t \mathcal{R}(s_t, a_t, s_{t+1}) \quad (2.2)$$

and thus the **optimal policy** π^* maximises the following term:

$$\sum_{t=0}^{\infty} \gamma^t \mathcal{R}(s_t, a_t, s_{t+1}), \text{ where } a_t = \pi^*(s_t). \quad (2.3)$$

Note that in order to avoid infinite loops, we limit the maximum number of steps the agent can take. The game episodes are consequently always finite and the rewards do not need to be discounted.

If the maximum number of steps is n , the agent's reward is then equal to $\sum_{t=0}^n \mathcal{R}(s_t, a_t, s_{t+1})$.

For finding the optimal policy in the text-game MDP, we employ reinforcement learning, whose solution techniques, as well as our approach to this specific instance of the problem, are described next.

2.2 Reinforcement learning

Reinforcement learning (RL) is an area of machine learning methods and problems based on the notion of learning from numerical rewards obtained by an agent through interacting with an environment Sutton and Barto [1998].

The RL agent does not have any supervisor and does not know the dynamics of its environment. Consequently, the agent has to both explore its environment and exploit its knowledge thereof.

In any given step, the agent observes a *state* of the environment and receives a *reward signal*. Based on the current state and the agent’s behaviour function — the *policy* — the agent chooses an *action* to take.

The action is then sent to the environment which is updated and the loop repeats. See figure 2.1 for illustration of the agent-environment interface.

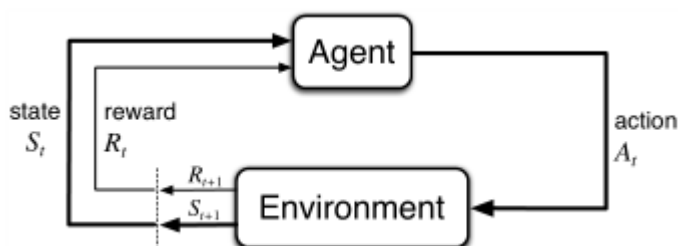


Figure 2.1: Interaction between the agent and the environment (Sutton and Barto [1998]).

Reinforcement learning is commonly employed for solving Markov decision process problems in which the dynamics of the environment are unknown to the agent.

The core idea behind solving an MDP problem using RL is iterative learning through combined exploration (finding out which states and actions lead to which rewards) and exploitation (updating the agent’s policy as to improve its expected long-time reward).

We follow the notation established in section 2.1.3 and denote the agent’s learned policy as $\pi(s)$.

2.2.1 Updating the policy

There are different approaches to learning the optimal policy. Some are based on learning the dynamics of the environment (model-based methods) while others solve the problem without explicitly modelling its transition function (model-free).

Here, we focus on model-free methods that have been shown to perform well on a variety of game-related tasks (Tesauro [1995], Mnih et al. [2015]).

In order to be able to update the policy, it is useful to work with the concept of *value functions*. A value function evaluates how good a given state under a given policy is, that is, what reward we can expect to obtain in the long run if we follow the policy from the specific state.

The **state-value function** $v_\pi(s)$ for policy π can be formally defined as (Sutton and Barto [1998])

$$v_\pi(s) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r_{t+1} \mid s \right], \quad (2.4)$$

where r_t corresponds to $\mathcal{R}(s_t, a_t, s_{t+1})$.

Similarly, an **action-value function** that determines the value of taking action a in state s using policy π is defined as

$$Q^\pi(s, a) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r_{t+1} \mid s, a \right]. \quad (2.5)$$

Now the optimal policy, denoted π^* , can be characterised by the optimal state-value function or the action-value function. The latter is defined as

$$Q^*(s, a) = \max_{\pi} Q^\pi(s, a). \quad (2.6)$$

In other words, if we know the optimal action-value function Q^* , we can obtain the optimal policy π^* by simply choosing the actions with maximum Q-values:

$$\pi^*(s) = \max_a Q^*(s, a). \quad (2.7)$$

Using this approach, we have now formally reduced the task of learning to play the text-based games to estimating an optimal action-value function in a text-game MDP. The Q-learning algorithm for solving the problem of optimal action-value function estimation is described next.

2.2.2 Q-learning

Q-learning (Watkins and Dayan [1992]) is an off-policy method for model-free control based on estimating the action-value function — the Q-function.

The Q-learning algorithm was shown to work well across a variety of different game-playing tasks (Mnih et al. [2015], Narasimhan et al. [2015], He et al. [2015]).

Q-learning is an off-policy algorithm, meaning the agent can follow an exploration policy while improving its estimates of the optimal policy by updating the Q-values.

Typically, the exploration policy is ϵ -greedy with relation to the Q-function, where $0 \leq \epsilon \leq 1$ is a parameter that corresponds to the probability of choosing a random action.

Q-learning attempts to find the optimal Q-values which obey the Bellman equation (Bellman [2013]):

$$Q^*(s_t, a_t) = \mathbb{E}_{s_{t+1}} \left[r_t + \gamma \cdot \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1}) \mid s_t, a_t \right]. \quad (2.8)$$

Consequently, the update rule for improving the estimate of the Q-function is as follows: (Sutton and Barto [1998]):

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha_t \cdot \left(r_{t+1} + \gamma \cdot \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right), \quad (2.9)$$

where $0 \leq \alpha_t \leq 1$ is the learning rate.

The Q-learning algorithm is guaranteed to converge towards the optimal solution (Sutton and Barto [1998]).

In simpler problems and by default, it is assumed that the Q-values for all state-action pairs are stored in a table.

This approach is, however, not feasible for more complex problems such as the Atari platform games task or our text-game task, where the state and action spaces are simply too large to store. In text games, for example, the spaces are infinite.

We deal with this problem by approximating the optimal Q-function by a function approximator in the form of a neural network. The Q-function is parametrised as

$$Q^*(s_t, a_t) \approx Q(s_t, a_t, \theta_t) = \theta_t(s_t, a_t), \quad (2.10)$$

where the θ function is realised by a neural network (see section 2.3 for an introduction and section 3.3 for architecture description).

The advantage of this non-tabular approach is that even in infinite spaces, the neural network can generalise to previously unobserved inputs and consequently cover the whole search space with reasonable accuracy.

In contrast to linear function approximators, though, non-linear approximators such as neural networks do not guarantee convergence in this context.

2.3 Neural networks

Artificial neural network (ANN or NN) is a computing system realised by interconnected groups of nodes inspired by biological neurons.

The original units, inspired by properties of biological neurons and capable of binary classification of linearly separable data, were called perceptrons (Rosenblatt [1958]).

A perceptron has a weight vector \mathbf{w} , a scalar bias value b and realises a simple binary function on its input vector \mathbf{x} :

$$f(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \\ 0 & \text{otherwise} \end{cases}, \quad (2.11)$$

where $\mathbf{w} \cdot \mathbf{x}$ is the dot product $\sum_i \mathbf{w}_i \mathbf{x}_i$.

First simple ANNs, called multilayer perceptrons (MLP), then consisted of organised layers of individual perceptron units connected together in a feed-forward fashion, forming a directed acyclic graph.

MLPs have one input and one output layer and an arbitrary number of hidden layers in between. Each layer consists of a set of perceptrons that additionally apply a non-linear *activation function* ϕ to their original output value:

$$f(\mathbf{x}) = \phi(\mathbf{w} \cdot \mathbf{x} + b). \quad (2.12)$$

The presence of multiple neurons in a single layer coupled with the nonlinear activations means that, in contrast to perceptrons, MLP classifiers — even those with only one hidden layer — are theoretically capable of classifying data that is not linearly separable.

In fact, a neural network with one hidden layer is a universal function approximator (Hornik [1991]).

MLPs are now commonly referred to as *feedforward neural networks* and the simple perceptron layers are called *fully-connected* or *dense* layers.

Neural networks are mostly employed in supervised learning classification problems across a variety of domains. In this context, the network is given training data $(\mathbf{x}_0, \mathbf{y}_0), \dots, (\mathbf{x}_n, \mathbf{y}_n)$, where \mathbf{x}_i are the *input vectors* and \mathbf{y}_i are the *target vectors* and the task is to adapt the weights of the network so that $\theta(\mathbf{x}_i) = \mathbf{y}_i$, where $\theta(\mathbf{x})$ is the network function.

2.3.1 Gradient descent

The task of learning in neural networks can be formalised as minimising a defined **loss function**.

Loss function of an input vector \mathbf{x} and a desired target vector \mathbf{y} is defined to quantify the error produced by the network.

One example of a loss function, denoted \mathcal{L} , is the mean squared error (MSE) function defined for a neural network θ and \mathbf{x}, \mathbf{y} of length n as

$$\text{MSE}(\mathbf{x}, \mathbf{y}, \theta) = \frac{1}{n} \sum_{i=1}^n (\theta(\mathbf{x})_i - \mathbf{y}_i)^2. \quad (2.13)$$

The optimisation problem then lies in minimising the given loss function.

There are different approaches to the optimisation problem in neural networks and here, we focus on the **backpropagation** algorithm (Werbos [1974], Hecht-Nielsen et al. [1988]) for gradient computation.

Backpropagation is based on first computing the error in the output layer and then propagating it backwards through the network.

More specifically, we compute the partial derivatives of the loss function with respect to the network weights and update the individual weights between neurons i and j in the direction of the gradient $\frac{\partial \mathcal{L}}{\partial \mathbf{w}_{ij}}$.

The error is then propagated to all weights in the previous layer and this is repeated for all layers up to the first hidden layer.

Using the backpropagated errors, the learning is commonly done using stochastic gradient descent (SGD):

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla \mathcal{L}(\mathbf{w}), \quad (2.14)$$

where α is the learning rate and \mathcal{L} is the loss function.

There are alternative optimisers for gradient descent in neural networks such as Adam (Kingma and Ba [2014]) or RMSProp (Tieleman and Hinton [2012]) based on the idea of adaptively changing the learning rate according to the gradient fluctuations that often give better results in practice.

Additionally, the weight updates are commonly done in batches, meaning a number of data samples is propagated through the network at once instead of separately for each sample, resulting in a faster training process and better convergence properties (Goodfellow et al. [2016]).

2.3.2 Recurrent networks

Recurrent neural networks (RNNs) represent a slightly different paradigm of computation, in which there are additional — recurrent — connections in the neural cells.

In this paradigm, instead of being presented at once, the input is supplied gradually in successive *timesteps*. And in addition to the “normal” input at a given timestep, a simple recurrent cell also receives its own last state as an input (see figure 2.2 for illustration).

Consequently, recurrent units, layers and networks provide additional computational power as they are able to maintain information about the past and, theoretically, capture long-term dependencies.

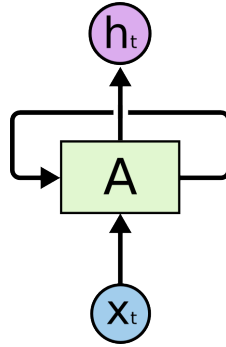


Figure 2.2: A simple RNN (Olah [2015]).
The cell **A** receives its last state as an additional input.

For learning in RNNs, a variant of backpropagation called backpropagation through time (BPTT) is used. BPTT is based on unfolding the network in time and gradually propagating the error to earlier timesteps (Werbos [1990]).

In practice, however, RNNs have great difficulty remembering the relevant information for long enough and tend to fail to learn important dependencies if there are too many timesteps between them (Hochreiter [1991], Bengio et al. [1994]).

Long short-term memory

To address the issue of long-term dependencies, Hochreiter and Schmidhuber [1997] introduced a special and a more complex variant of RNN cells, the long short-term memory (LSTM).

The basic idea of LSTMs is to work with multiple information pathways inside the cell.

On the main pathway, the information from the previous state flows unchanged and additional information can be added to the previous cell state through different gates realised by simple neural network layers with the sigmoid activation function.

For a thorough explanation, see Olah [2015].

In practice, LSTM-based networks have displayed incredible performance, obtaining state-of-the-art results in a number of disciplines including language modelling (Sundermeyer et al. [2012]) or speech recognition (Graves et al. [2013]).

Thanks to the emphasis that is put on preserving long-term relations in LSTMs, they are suited particularly well for NLP-related tasks, where long-term dependencies in the input data often form important and useful features.

2.4 Language representation

One of the critical aspects of NLP tasks is learning a good representation of the input data through creating meaningful features that capture the word semantics.

In this section, we first briefly talk about text preprocessing as a tool to reduce the vocabulary size and consequently make the task somewhat easier.

Second, we describe the process of converting the input textual data into a powerful numerical vector representation.

2.4.1 Text preprocessing

As we mentioned in section 1.4.1 and as should be clear from the description of word embeddings (see section 2.4.2), the difficulty of the representation task as well as the text-game task grows with the increasing number of unique words.

In our task, we currently only deal with vocabulary sizes of thousands of words, which is why we employ only few basic text preprocessing rules based on the syntactic content and not on semantics.

In other words, we are not interested in matching synonyms or more generally, sets of words that would increase the vocabulary size linearly. Instead, we primarily deal with strings whose number can potentially explode exponentially, i.e. with tokens that do bear a meaning by themselves but are usually not separated from other tokens by whitespace characters.

For preprocessing, we follow this procedure:

1. convert the text to lower case,
2. remove all special characters (only preserve alphanumerical and whitespace characters, quotes and hyphens),
3. split numbers into digits,
4. insert a space between X 's expressions, i.e. $X's$ becomes $X 's$,
5. expand unambiguous contracted expressions such as 'll or 've.

Finally, split the text data by white spaces and assign a unique **token** to each non-empty unique string.

2.4.2 Embeddings

In order to be able to work with high-dimensional complex data such as text in neural networks, we have to first convert the text data into numerical vectors.

Generally, this process is called embedding and in our case, **word embeddings** are the real-valued vectors corresponding to the actual words in the input.

NLP techniques enable us to both map complex textual data into vectors of numbers and, importantly, to also find a relevant mapping function that results in a powerful useful representation.

The process of creating word embeddings from text is described next.

The first step is to identify possible unique tokens (see section 2.4.1 for an example in our context). This is usually done by scanning for all possible words in a dataset, however, it is possible to theoretically add new tokens on the fly at the cost of computational expenses. In text games, for example, it is convenient to parse the source code or sample the game randomly for thousands of episodes to obtain a good estimate of its complete vocabulary.

After obtaining the vocabulary of tokens, they are ordered and an index is assigned to each token, meaning each sentence is now represented as a vector of token indices.

Finally, each token has a set of parameters to it; this is the actual word embedding. Word embeddings are high-dimensional vectors (the dimension is typically 50 or 100) that — we hope and hypothesise — represent semantic features of their corresponding words.

There are different approaches to learning good and useful word embeddings.

Recently, RNNs and RNNs with LSTM units have been successfully applied to a number of related tasks in representation generation (Mikolov et al. [2013], Palangi et al. [2016]).

Specifically, we follow the approach used in Narasimhan et al. [2015], where the representation generator module was trained as a part of a neural network for estimating a Q-value, learning the representation and the control policy jointly.

2.5 Summary

We briefly reviewed the basic methods of reinforcement learning, neural networks and language representation that we utilise for building our agent. Below, we summarise their usage in the context of the text game learning task.

We utilise reinforcement learning to define the agent’s objective and consequently learn the control policy. We use the model-free Q-learning control algorithm that estimates the agent’s action-value function (Q-function).

The Q-function is approximated using supervised learning in the form of a neural network. The network makes use of embedding and LSTM layers for representation generation and of dense layers followed by an interaction function between states and actions to finally estimate the Q-value.

The learning targets that determine the value of the network’s loss function are then based on the basic Q-learning update rule (see equation 2.9).

The network is trained in an end-to-end manner, implicitly building important feature representations of the input text in each layer.

The resulting architecture and its parameters are described in more detail in chapter 3.

3. Agent architecture

We have now introduced the domain of IF games and the methods commonly used for solving similar sequential decision-making tasks.

In this chapter, we present the architecture of our agent, denoted SSAQN, capable of playing choice-based text-games using reinforcement learning.

After presenting recent relevant models and explaining our motivation behind choosing a slightly different architecture, the structure of the chapter closely follows agent’s data flow — we start with complete text descriptions of states and actions and gradually work our way towards a condensed vector representation used for determining the compatibility of the original state-action pairs.

3.1 Related models

We briefly describe two architectures that were recently used to play text games.

3.1.1 LSTM-DQN

Narasimhan et al. [2015] presented an LSTM-DQN agent that used an LSTM network for representation generation, followed by a variant of a Deep Q-Network (DQN, Mnih et al. [2015]) used for scoring the generated state and action vectors.

The underlying task of playing parser-based games was effectively reduced to playing choice-based games by presenting a subset of all possible verb-object tuples as available actions to the agent.

In the framework, a single action consists of a verb and an object (e.g. *eat apple*) and the model computes Q-values for objects — $Q(s, o)$ — and actions — $Q(s, a)$ — separately.

The final Q-value is obtained by averaging these two values.

3.1.2 DRRN

He et al. [2015] introduced a Deep Reinforcement Relevance Network (DRRN) for playing hypertext-based games. The agent learned to play *Saving John* and *Machine of Death* and used a simple bag-of-words (BOW) representation of the input texts.

The learning algorithm is also a variant of DQN; DRRN refers to the network that estimates the Q-value, using separate embeddings for states and actions. DRRN then uses a variable number of hidden layers and makes use of softmax action-selection.

The final Q-value is obtained by computing an inner product of the inner representation vectors of states and actions.

3.2 Motivation

Our goal is to introduce a minimal architecture serving as a proof of concept, with the ability to capture important sentence-level features and ideally capable of reasonable generalisation to previously unseen data.

First, we highlight some of the aspects of the LSTM-DQN and DRRN models that could be improved upon in terms of these requirements.

The main drawback of DRRN is its use of BOW for representation generation. Consequently, the model is incapable of properly handling state aliasing and differentiating simple, yet important, nuances in the input, such as in “*There is a treasure chest to your left and a dragon to your right.*” and “*There is a treasure chest to your right and a dragon to your left.*”.

Moreover, He et al. [2015] claim that separate embeddings for state and action spaces lead to faster convergence and to a better solution. However, since both state and action spaces really contain the same data — at least in most games and especially in hypertext games where actions are a subset of states — we aim to employ a joint embedding representation of states and actions.

We also believe that a joint representation of states and actions should eventually lead to stronger generalisation capabilities of the model, since such model should be able to transfer knowledge between state and action descriptions as their representation would be shared.

The LSTM-DQN agent, on the other hand, utilises an LSTM network that can theoretically capture more complex sentence properties such as the word order. However, its architecture only accepts actions consisting of two words.

Additionally, the two action Q-values are finally averaged, which would arguably be problematic if the verbs and objects were not independent. For example, the value of the verb “*drink*” varies highly based on the object; consider the difference between the values of “*drink*” when followed by either “*water*” or “*poison*” objects.

We thus aim to utilise a minimalistic architecture that should:

- be able to capture dependencies on sentence level such as word order,
- accept text input of any length for both states and action descriptions,
- accept and evaluate any number of actions,
- use a powerful interaction function between states and actions.

Next, we present our chosen model.

3.3 SSAQN

Our neural network model is inspired by both LSTM-DQN and DRRN. For the sake of clarity, it is referred to as **SSAQN** (Siamese State-Action Q-Network).

Similarly to Narasimhan et al. [2015] and He et al. [2015], we employ a variant of DQN (Mnih et al. [2015]) with experience replay and prioritised sampling which uses a neural network (SSAQN) for estimating the DQN’s Q-function.

SSAQN uses a siamese network architecture (Bromley et al. [1993]), where two branches — a state branch and an action branch — share most of the layers that effectively generate useful representation features.

This is best illustrated by visualising the network’s computational graph; see figure 3.1 on the next page.

As we are using a twin architecture, the weights of the embedding and LSTM layers are shared between state and action data passing through.

States and actions are only differentiated in the dense layers whose outputs are then fed into the similarity interaction function.

The most important differences to LSTM-DQN and DRRN are:

- the network accepts two text descriptions (state and action) of arbitrary length as an input,
- the embedding and LSTM layers are the same for states and action, i.e. their weights are shared,
- the interaction function of inner vectors of states and actions is realised by a normalised dot product, commonly called cosine similarity (see sec. 3.3.4).

The output of the SSAQN is the estimated Q-value for the given state-action pair, i.e. the network with parameters θ realises a function:

$$\theta(s, a) = Q(s, a, \theta) \approx Q^*(s, a), \quad (2.10)$$

where the input variables s and a contain preprocessed text as described in section 2.4.1.

To compute the $Q(s, a^i)$ for different i — for multiple actions — we simply run the forward pass multiple times.

Next, the SSAQN architecture is described layer-by-layer in more detail. Additional technical details including the network parameters are discussed in section 3.6.

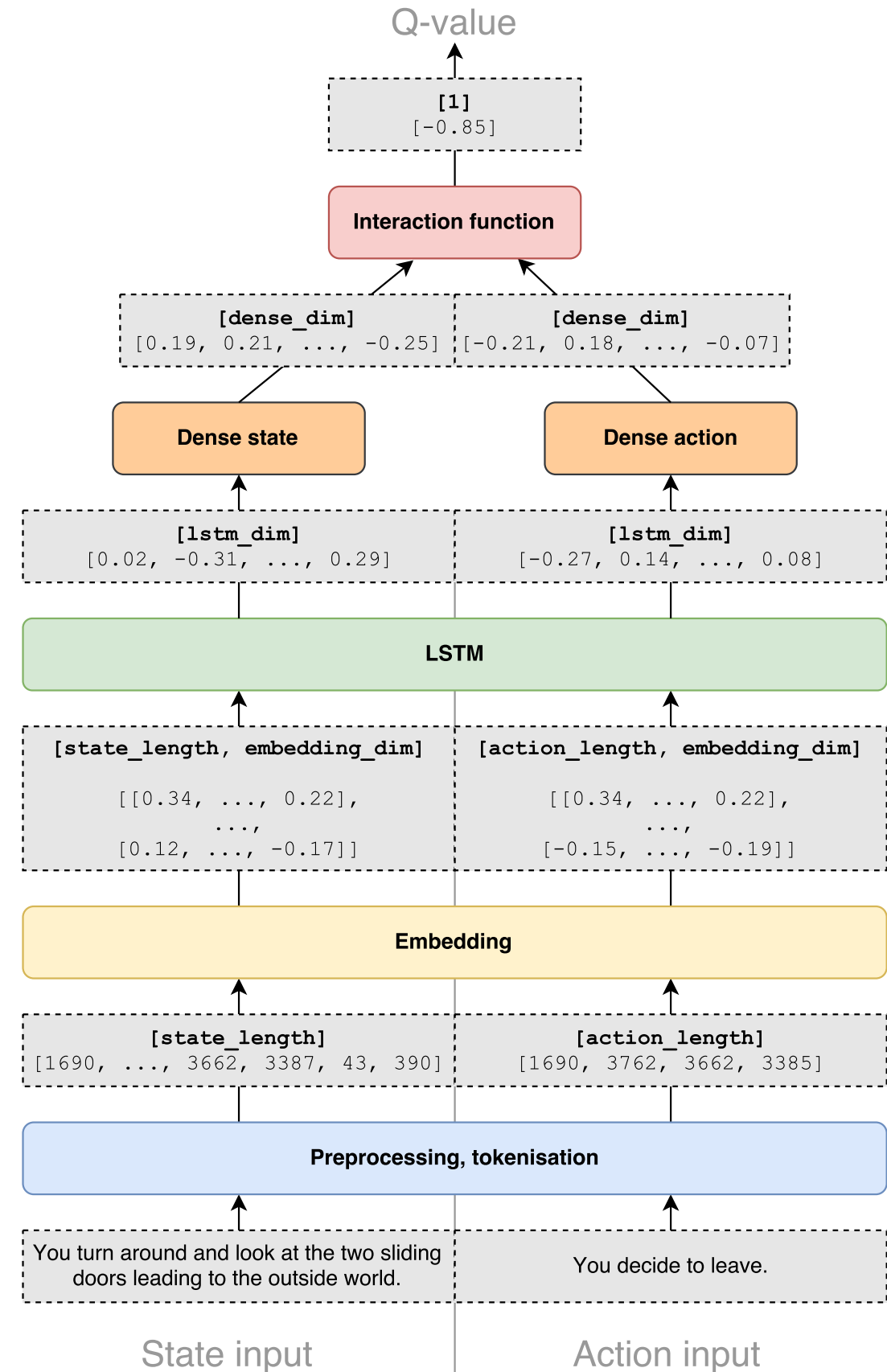


Figure 3.1: Architecture of the SSAQN model and its data flow. Grey boxes represent data values in different layers; the bold text corresponds to the shape of the data tensors.

3.3.1 Word embeddings

After preprocessing the text (see section 2.4.1), we convert the words to their vector representation using word embeddings (see section 2.4.2).

Since our dataset is comparatively small, we use a relatively low dimensionality for the word representations and work with `embedding_dim` of 16.

The weights of the word embeddings are initialised to small random values and trained as a part of the gradient descent algorithm.

Naturally, the weights of the state and action branches are shared, meaning that a word appearing in a state description and in a action descriptions is converted to the same vector in both branches.

Importantly, when training, instead of updating single state-action tuples, we perform the updates in batches. Consequently, we pad the input sequences of words for each batch so that they are all aligned to the same length.

The padding is realised by prepending a special token with an index of 0 to the padded sentence.

In principle, it is not necessary to pad the sequences or perform the updates in batches, but this leads to a faster training process. Still, note that the length of the output of the embedding layer is variable in length.

3.3.2 LSTM layer

The inputs of the LSTM layer (explained in section 2.3.2) are the word embedding vectors of variable length.

Similarly to embeddings, the weights of the LSTM units are also initialised to small random values and their weights are shared between states and actions.

The role of the LSTM layer is to successively look at the input words, changing its inner state in the process and thus detecting time-based features in its input.

It is also at this layer that we go from having a data shape of arbitrary length to having a fixed-length output vector.

The output size is equal to the number of LSTM units in the layer and in our experiments, we use `lstm_dim` of 32.

3.3.3 Dense layer

Following the shared LSTM layer, we now have two dense layers (also commonly called fully-connected), one for states and one for actions.

Again, we initialise the weights randomly and we also apply the hyperbolic tangent activation function:

$$\tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}. \quad (3.1)$$

As the dense layers for states and actions are the only layers to not necessarily share weights between the two network branches, they do play an important role in building differentiated representations for state and action input data.

Note that as the interaction layer uses a dot product, we require both outputs of the dense layers to be of the same dimension and we set `dense_dim` of both branches to 8.

However, theoretically, it would be interesting to use different layer dimensions for states and actions at this level, as usually, the original state text descriptions carry more information than action descriptions in IF games.

Thus, a possible extension of the network would be to use two or more hidden dense layers in the state branch of the network and to only reduce the dimension to the action dimension in the last hidden dense state layer.

3.3.4 Interaction function

Lastly, we apply the cosine similarity interaction function to the state and action dense activations, resulting in the final Q-value.

If the input are two vectors \mathbf{x} and \mathbf{y} of `dense_dim` = n elements, we define their cosine similarity as a dot product of their L2-normalised (and consequently unit-sized) equivalents:

$$\text{cs}(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\|_2 \|\mathbf{y}\|_2} = \frac{\sum_{i=1}^n \mathbf{x}_i \mathbf{y}_i}{\sqrt{\sum_{i=1}^n \mathbf{x}_i^2} \sqrt{\sum_{i=1}^n \mathbf{y}_i^2}}, \quad (3.2)$$

which corresponds to the cosine of the angle between the input vectors.

Cosine similarity is commonly used for determining document similarity (Huang [2008]).

Here, we apply it to the two hidden vectors of dense layer values that should meaningfully represent the condensed information that was originally received as a text input by the network and we interpret the resulting value as an indicator of mutual compatibility of the original state-action pair.

Obviously, the range of values of the cos function is $[-1, 1]$, whereas the original rewards that we aim to estimate have arbitrary values. Therefore, we need to scale the approximated Q-values as discussed in section 3.6.2.

3.3.5 Loss function and gradient descent

Recall the Q-learning rule (see equation 2.9). We define the loss function at time t as

$$\mathcal{L}_t = \left(r_t + \gamma \cdot \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right)^2, \quad (3.3)$$

which is simply a mean squared error (see equation 2.13) of the last estimated Q-value and the target Q-value.

For gradient descent, we make use of the RMSProp optimiser (Tieleman and Hinton [2012]) that has been shown to perform well in numerous practical applications, especially when applied in LSTM networks (Dauphin et al. [2015]).

3.4 Action selection

Given an SSAQN θ , where $Q(s, a) \leftarrow \theta(s, a)$, the agent selects an action by following the ϵ -greedy policy $\pi_\epsilon(s)$ (Sutton and Barto [1998]) realised by the following algorithm:

Algorithm 1 Action selection

ϵ .. probability of choosing a random action
 $h(s, a)$.. number of times the agent selected a in s in the current run

- 1: **function** ACT($s, actions, \epsilon, h$)
- 2: **if** $random() < \epsilon$ **then return** random action
- 3: **end if**
- 4: $q_values = \theta(s, a_i)$ **for** a_i in $actions$
- 5: $q_values = (q_values + 1)/2$ \triangleright scale Q-values from $[-1, 1]$ to $[0, 1]$
- 6: $q_i = q_i^{h(s, a_i)+1}$ **for** q_i in q_values \triangleright apply the history function
- 7: $q_values = (q_values \cdot 2) - 1$ \triangleright scale Q-values from $[0, 1]$ to $[-1, 1]$
- 8: **return** a_i with $\max q_i$ \triangleright ¹
- 9: **end function**

The ϵ is the exploration parameter as described in section 2.2.2.

For sampling in training phase (see algorithm 2), ϵ is gradually decayed from the starting value of 1, i.e. at first, the agent’s policy is completely random.

In testing phase, 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.

History function

The only important difference between the standard ϵ -greedy control algorithm and our action selection policy is that we additionally employ a *history function*, $h(s, a)$.

The scope of the history function is a single run of the agent on a single game, i.e. it is reset every time a game ends.

$h(s, a)$ returns a value equal to the number of times the agent selected action a in state s in the current run. That is, if the agent never selects an action twice in the same state during a run, the history function has no impact on action selection.

To be precise, the history function penalises the already visited state-action pairs, as seen on line 6 of algorithm 1.

¹Lines 4 to 8 can be formally written as:
return $\operatorname{argmax}_{a_i \in actions} (((\theta(s, a_i) + 1)/2)^{h(s, a_i)} \cdot 2) - 1$

The history function serves as a very simple form of intrinsic motivation (Singh et al. [2004], Singh et al. [2010]). It is similar to optimistic initialisation (Sutton and Barto [1998]) in that it leads the agent to select previously unexplored state-action tuples.

Additionally, note that the history function is not Markovian in the sense that it takes the whole game episode into account. In practice, the history function greatly helps the agent to avoid infinite loops, since for many games, it is likely to get stuck in an infinite loop when following a random deterministic Markovian policy (see section 1.4.4).

3.5 Training loop

Putting together all the parts introduced above, we can now formally describe the agent’s learning algorithm.

We use a variant of DQN (Mnih et al. [2015]) with experience replay and prioritised sampling of experience tuples with positive rewards (Schaul et al. [2015]).

Note that the agent supports playing — and learning on — multiple games at once.

Algorithm 2 Training algorithm (a variant of DQN)

episodes .. number of episodes, *b* .. batch size, *p* .. prioritised fraction
 ϵ .. exploration parameter, ϵ_decay .. rate at which ϵ decreases

```

1: function TRAIN(episodes, b, p,  $\epsilon = 1$ ,  $\epsilon\_decay$ )
2:   Initialise experience replay memory  $\mathcal{D}$ 
3:   Initialise the neural network  $\theta$  with small random weights
4:   Initialise all game simulators and load the vocabulary
5:   for  $e \in 0, \dots, episodes - 1$  do
6:     Sample each game once using  $\pi_\epsilon$ , store experiences into  $\mathcal{D}$ 
7:     batch  $\leftarrow b$  tuples  $(s_t, a_t, r_t, s_{t+1}, a_{t+1})$  from  $\mathcal{D}$ , where
           a fraction of p have  $r_t > 0$ 
8:     for  $i, (s_t^i, a_t^i, r_t^i, s_{t+1}^i, a_{t+1}^i)$  in batch do
9:       targeti  $\leftarrow r_t^i$ 
10:      if  $a_{t+1}^i$  then  $\triangleright s_{t+1}^i$  is not terminal
11:        targeti  $\leftarrow \gamma \cdot \max_{a_{t+1}^i} \theta(s_{t+1}^i, a_{t+1}^i)$   $\triangleright Q(s, a, \theta) = \theta(s, a)$ 
12:      end if
13:    end for
14:    Define loss as  $\mathcal{L}_e(\theta) \leftarrow (target_i - \theta(s_t^i, a_t^i))^2$   $\triangleright$  see equation 3.3
15:    Perform gradient descent on  $\mathcal{L}_e(\theta)$ 
16:     $\epsilon \leftarrow \epsilon \cdot \epsilon\_decay$   $\triangleright$  decrease the exploration parameter (algorithm 1)
17:  end for
18: end function

```

3.6 Technical details

The architecture described above as well as its optimisation phase was implemented in **Keras** (version 2.0.4, Chollet et al. [2015]), using the **Tensorflow** backend (version 1.1.0, Abadi et al. [2015]).

For complete reference, the source code of the agent is freely available as part of **pyfiction** (see appendix B). Additionally, visualisation of the implemented architecture as defined in figure 3.1 is available on page 64.

3.6.1 Parameters

Both the training algorithm and the neural network use a set of parameters that can dramatically change the course of the learning process.

The employed dimensions of the neural network layers as defined in figure 3.1 are described above, in sections 3.3.1, 3.3.2 and 3.3.3, as they were identical in all our experiments.

All parameters of the training algorithm are summarised in section 4.1.4 or given in the sections of the individual experiments.

3.6.2 Scaling the Q-values

In text games, the player can be given rewards of arbitrary values.

However, note that the Q-value as approximated by SSAQN is a result of the cosine similarity operation (see equation 3.2), thus the approximated value is in $[-1, 1]$.

Since we are trying to approximate the actual game reward, we use this to our advantage and scale the Q-value — with the knowledge of maximum possible reward in game environment — to $[-r_{\max}, r_{\max}]$, where r_{\max} is the largest possible cumulative reward, ergo r_{\max} is a parameter of the game environment.

This approach has the additional benefit of the Q-values getting normalised between games with different reward scales.

3.6.3 Estimating multiple Q-values effectively

In section 3.3, we said that in order to determine Q-values in a single state for different actions, multiple forward passes of the network function $\theta(s, a)$ are needed.

Obviously, there is overhead associated with such approach, as we are evaluating the state branch repeatedly using the same state input text.

To compute multiple $Q(s, a_i)$ values effectively, we instead perform a single partial state forward pass and cache the value of the state dense layer activations.

Then, we perform a partial action forward pass for each action and compute the respective Q-value using the cached values of the state dense layer, resulting in up to twice as effective computation when compared to the naive approach.

3.7 Summary

We presented SSAQN, a minimalistic neural network with siamese architecture for efficient estimation of Q-values in the context of the text-game learning task.

SSAQN is powerful in the sense that it accepts and handles input of arbitrary length and is, at least theoretically, capable of capturing long-term language dependencies thanks to the LSTM layer that operates on word embeddings.

Moreover, the employed similarity interaction function applied on inner state and action representations should be able to give good results even for mutually semantically dependent state-action pairs.

To see how the model performs in practice, we conduct experiments on several text games in chapter 4.

4. Experiments

In this chapter, we conduct learning experiments using the SSAQN agent described in chapter 3.

In each experiment, we first explain our motivation, then describe the methods and employed parameters and after presenting the results, we discuss the experiment’s implications.

We evaluate the agent on a combination of six IF games, namely *Saving John*, *Machine of Death*, *Cat Simulator 2016*, *Star Court*, *The Red Hair* and *Transit* (see section 4.1.1 for their description).

For more details about the selected games including relevant statistics and ending annotations, refer to appendix A which also includes a table summarising all game parameters as well as the experiment results (see page 63).

4.1 Setup

In this section, we briefly introduce the IF games we used for the learning tasks and describe the experiment evaluation process.

We also give an overview of methods and parameters common for all experiments.

4.1.1 Games

When selecting IF games for the learning task, we searched for hypertext-based or choice-based games with multiple endings, preferably with a high rating count on the IFDB.

The game simulators for the selected games, *Saving John* (SJ), *Machine of Death* (MoD), *Cat Simulator 2016* (CS), *Star Court* (SC), *The Red Hair* (TRH) and *Transit* (TT) are all implemented in pyfiction (see appendix B) and share the same agent-environment interface.

For *Saving John* and *Machine of Death*, we wrapped the functionality of game simulators as presented in He et al. [2015].

For all other games, we obtained their publicly available web-based versions and their simulators internally use a web browser as an interface.

Table 4.1 summarises game statistics relevant to the learning task difficulty as discussed in section 1.4, including the presence of non-deterministic transitions and descriptions.

	SJ	MoD	CS	SC	TRH	TT
# tokens	1119	2055	364	3929	155	575
# states	70	≥ 200	37	≥ 420	18	76
# endings	5	≥ 14	8	≥ 17	7	10
Avg. words/description	73.9	71.9	74.4	66.7	28.7	87.0
Deterministic transitions	Yes	No	Yes	No	Yes	Yes
Deterministic descriptions	Yes	Yes	Yes	No	Yes	Yes
Optimal reward	19.4	≈ 21.4	19.4	?	19.3	19.5

Table 4.1: Summary of game statistics.

Next, we discuss the consequences of the specific non-deterministic properties of the games on the learning task difficulty.

Machine of Death

In *Machine of Death*, the player is randomly given a prophecy of one of three possible fates very early on in the story when interacting with the Machine of Death.

After that, there are three completely independent stories based on the randomly selected fate, meaning *Machine of Death* essentially contains three different games.

The game thus does have a few non-deterministic transitions but all its descriptions are deterministic.

As mentioned in section 1.3, an additional complexity in *Machine of Death* comes from the fact that one of the endings enumerates a number of player’s recent actions, meaning there are effectively hundreds of possible final states.

To be able to test generalisation properties of their model, He et al. [2015] introduced a version of *Machine of Death* with paraphrased action descriptions which we also utilise in our generalisation experiment.

Star Court

Star Court is a story set in distant future where the player is on trial for a crime they likely did not commit and where they have to defend themselves to prove their innocence.

The game is highly random and as even the number of years the player has to spend in prison — should they be found guilty — is random, there are effectively thousands of different endings and possible final reward values.

Both the game transitions and game descriptions vary highly and it is very difficult to determine if a good ending can always be reached without relying on luck¹.

Saving John, Cat Simulator 2016, The Red Hair, Transit

All four remaining games — *Saving John*, *Cat Simulator 2016*, *The Red Hair* and *Transit* — are deterministic both in their transitions and in their descriptions.

¹We were unable to determine whether this actually is the case. Anyway, it is *likely* that it is not possible to consistently reach a good ending with a positive reward.

Consequently, as discussed in section 1.4, these games should be similarly difficult to play.

Difficulty and optimal rewards

By the criteria established in section 1.4, all games but *Machine of Death* and *Star Court* are *simple*, i.e. both their transitions and descriptions are deterministic and can thus be solved by simple search algorithms.

Therefore, the optimal cumulative reward is always reachable in any instance of a simple game as seen in table 4.1.

The same applies for each branch of *Machine of Death*, i.e. a deterministic policy exists that always finds the maximum possible reward in all three stories. However, the value of the game’s average optimal reward depends on the hidden transition probabilities and we estimate it at about 21.4.

In *Star Court*, it is yet unknown if an optimal or a positive reward can always be reached¹.

4.1.2 Game simulator properties

The agent interacts with *game simulators* implemented in *pyfiction*, objects that serve as interfaces between the agent and the game.

There are slight differences between the underlying games and the output of their simulators for the purposes of the learning task.

First of all, the game simulators remove all special game elements such as images or sounds.

Secondly, in IF games, available actions are usually presented in the same way and in particular, in the same order. Consequently, the game simulator randomly shuffles the game actions in each step. This is to force the agent to consider the contents of the actions and to make it impossible to only learn to act based on the action order.

Lastly, we impose a limit on the maximum number of steps the agent takes in the game in order to break otherwise infinite cycles. Its value is set to 500 for *Machine of Death* and *Star Court* and to 100 for all other games.

4.1.3 Evaluation

To evaluate an agent’s performance on a game, we simply let the agent play the game using the action selection mechanism as described in section 3.4 with ϵ set to zero, ergo the agent uses the greedy policy $\pi_0(s)$ and is deterministic.

The reward of a single game episode is simply a sum of the rewards obtained in all timesteps of the episode.

More precisely, the agent is tested at the end of every n th episode of the training process, i.e. right after the episode’s weight update (see algorithm 2).

In deterministic games, the agent is only evaluated once in each n th episode as its behaviour is also deterministic.

In *Machine of Death* and *Star Court*, we let the agent play the game five times in each n th episode and record the average of the obtained rewards and their standard deviation.

We set n to 1 when playing deterministic games individually and to either 8 or 16 in all other cases to speed the training process up.

4.1.4 Parameters

See table 4.2 for a summary of used parameter values as defined in chapter 3.

Unless specified otherwise, we use the default values for built-in parameters of both Keras and Tensorflow.

Layer	Dimension	Parameter	Value
Embedding	16	Optimiser	RMSProp
LSTM	32	Learning rate	<i>variable</i>
Dense	8	Batch size	256
		γ	0.95
		ϵ	1
		ϵ -decay	<i>variable</i>
		Prioritised fraction	0.25

(a) SSAQN layer dimensions

(b) Training algorithm parameters

Table 4.2: Parameters used in the learning tasks.

4.2 Individual games

Firstly, we train the agent on individual games and evaluate its performance on the same data, i.e. train datasets are equal to test datasets.

The results are compared to the random agent baseline (labelled `random`) and additionally, for *Saving John* and *Machine of Death*, we are able to compare the results to the DRRN model from He et al. [2015].

Note that for all following experiments, we additionally use the result of this individual game task (labelled `test`) as a baseline.

4.2.1 Setting

We run a separate experiment for each game using the learning parameters as seen in table 4.3.

Recall that the evaluation is done using the agent’s greedy test policy realised by the SSAQN, i.e. $\epsilon = 0$.

	Learning rate	ϵ -decay
Saving John	0.00010	0.990
Machine of Death	0.00001	0.999
Cat Simulator 2016	0.00010	0.990
Star Court	0.00001	0.999
The Red Hair	0.00010	0.990
Transit	0.00100	0.990

Table 4.3: Learning parameters for the individual games task.

4.2.2 Results

Results are depicted in figure 4.2.1 on page 40.

In all fully deterministic games, the agent converges to an optimal policy² in under 250 episodes.

In *Machine of Death* (figure 4.2.1b), the agent did not converge in the given number of episodes as indicated by the high standard deviations of the average reward. Nevertheless, the resulting policy performs significantly better than the random baseline.

In *Star Court* (figure 4.2.1d), the agent performs better than the random baseline on average but it is difficult to judge the optimality of the resulting policy (see section 4.1.1).

²Recall that a policy is optimal if it maximises the cumulative reward (see section 2.2.1). Consequently, the agent converged to the best possible total reward.

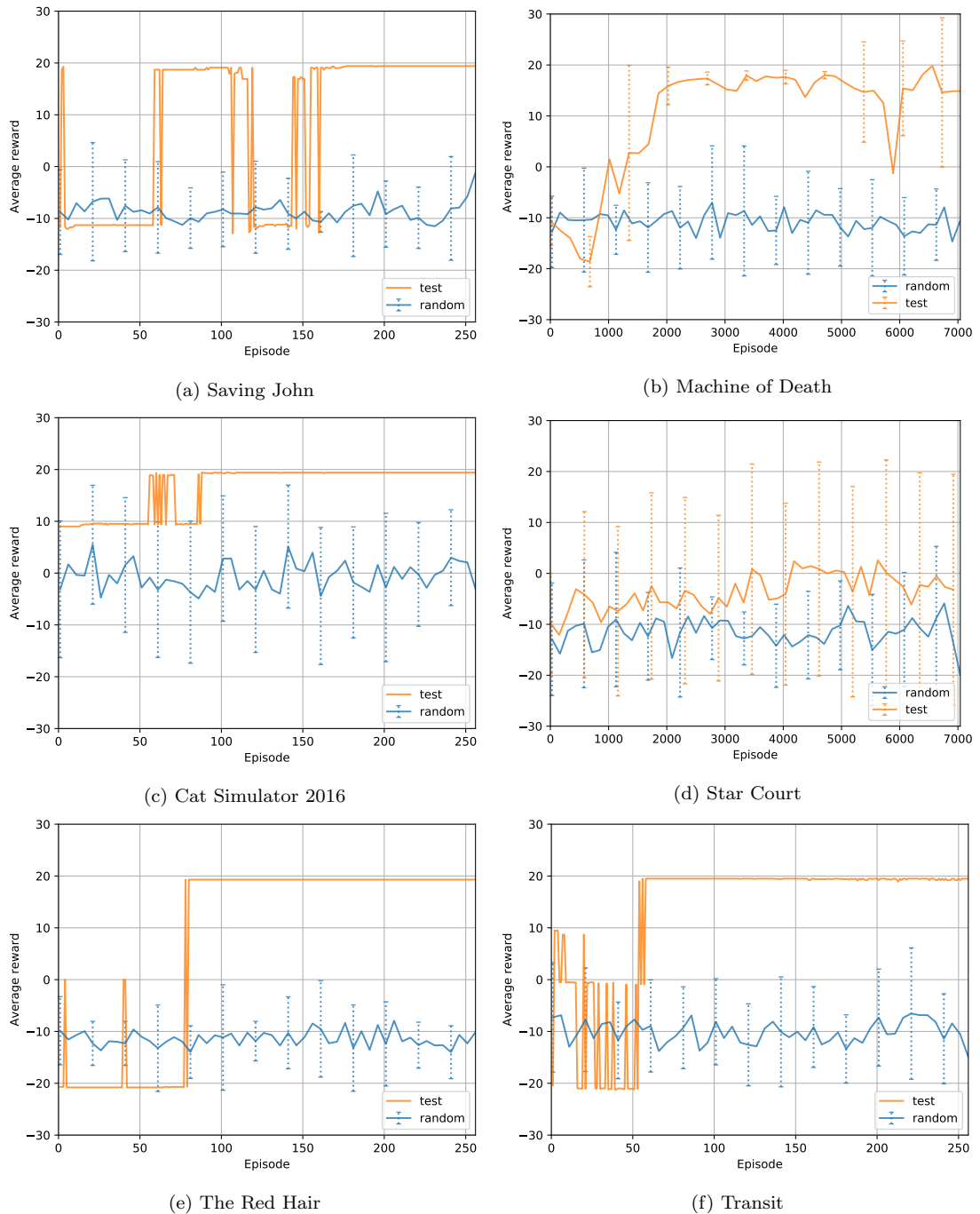


Figure 4.2.1: Performance of the SSAQN agent in individual games.

Comparison to DRRN

In figure 4.2.2, we compare the performance of SSAQN and DRRN on *Saving John* and *Machine of Death*.

Note that the speed of convergence is incomparable as both algorithms perform slightly different operations in a single episode.

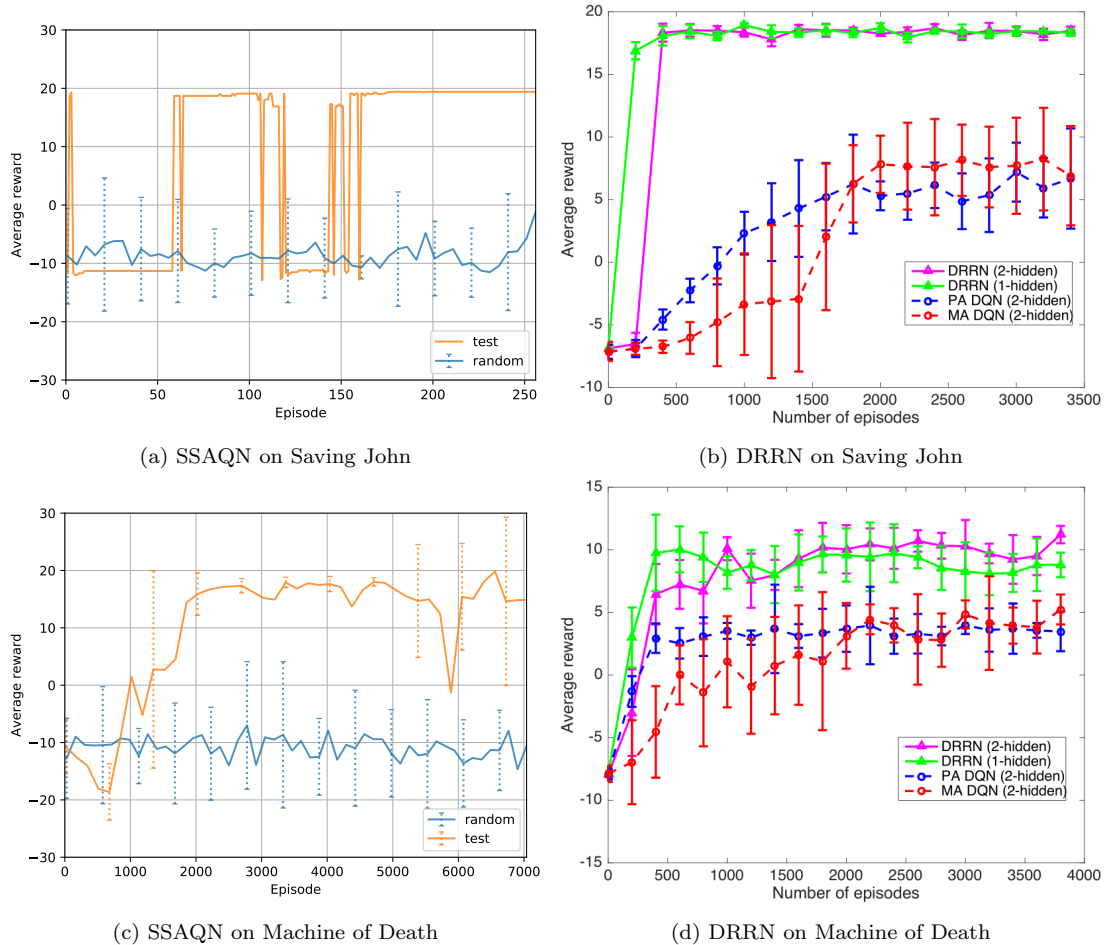


Figure 4.2.2: Comparison of SSAQN and DRRN on Saving John and Machine of Death.

In *Saving John*, the SSAQN agent converged to the optimal reward of 19.4, whereas the DRRN agent converged to a nearly-optimal reward of 18.7 (He et al. [2015]).

The fluctuations in DRRN’s performance are likely caused by its employment of softmax selection.

In *Machine of Death*, the DRRN agent converged to a local optimum of 11.2. The SSAQN agent did not converge in this number of iterations and the performance deviations are much higher. However, the average reward is noticeably higher than in DRRN, at about 15.4.

When looking at the raw performance data of SSAQN on *Machine of Death*, we see that the high standard deviations are caused by its volatility in one of the three game branches where the reward oscillates highly.

In the other two game branches, SSAQN converged to the optimal rewards of 28.5 and 18.4 respectively.

4.2.3 Discussion

The agent was able to learn to play all games well barring *Star Court*, in which it is difficult to define the optimal solution and thus judge the agent’s performance.

It will be more interesting to see how different tasks influence agent’s perfor-

mance in *Star Court*.

In *Machine of Death*, a higher number of iterations or employment of other techniques such as learning rate decay would likely be needed to converge to a local optimum on the third game branch.

Still, the resulting policy was better than that of the DRRN agent on average.

We hypothesise that this is thanks to the SSAQN’s ability to extract a higher number of features from the same amount of data, as it is able to capture sentence-level word relations and to distinguish between a number of states and actions represented identically in DRRN.

This difference is consequently more pronounced on *Machine of Death*, where there are significantly more states and actions than in *Saving John* in which SSAQN only performs slightly better than DRRN.

Learning instabilities

An interesting property of the learning curves (fig. 4.2.2a being the prime example) of the simple games is their high instability.

We believe these instabilities exist because in a single episode, there is a comparatively low number of decisions to make. Additionally, IF games can often be extremely sensitive and volatile (or, in the IF terms, “unforgiving”) and a single wrong action can result in falling into a game branch with very low rewards without being able to correct the mistake.

Taken together, this means that even a very slight change in the policy affecting only a single decision can often significantly impact the final reward, which is what we see in the results.

Compare this to the Atari games platform where in each episode, the agent performs thousands of updates on average. As a result, the impact of a single decision is much less significant than in IF games.

4.3 Generalisation

Next, we test how the agent performs in terms of generalisation.

The ability to generalise, or, in our context, the ability to perform well in previously unseen games, is a crucial factor determining the power and the usefulness of the model.

4.3.1 Setting

For this purpose, we train the agent using the leave-one-out method, in which five out of the six games are provided to the agent as training simulators and the agent’s performance is simultaneously evaluated on the sixth game, unseen during training.

The vocabulary provided to the agent is a unification of the vocabularies of all six individual games consisting of 5567 tokens.

In this section, referring to an experiment by a game name means that this specific game was left out during training phase and only used for testing.

We used the same parameters for all games; a learning rate of 0.00001 and an ϵ -decay of 0.999.

4.3.2 Results

Results are shown in figure 4.3.1.

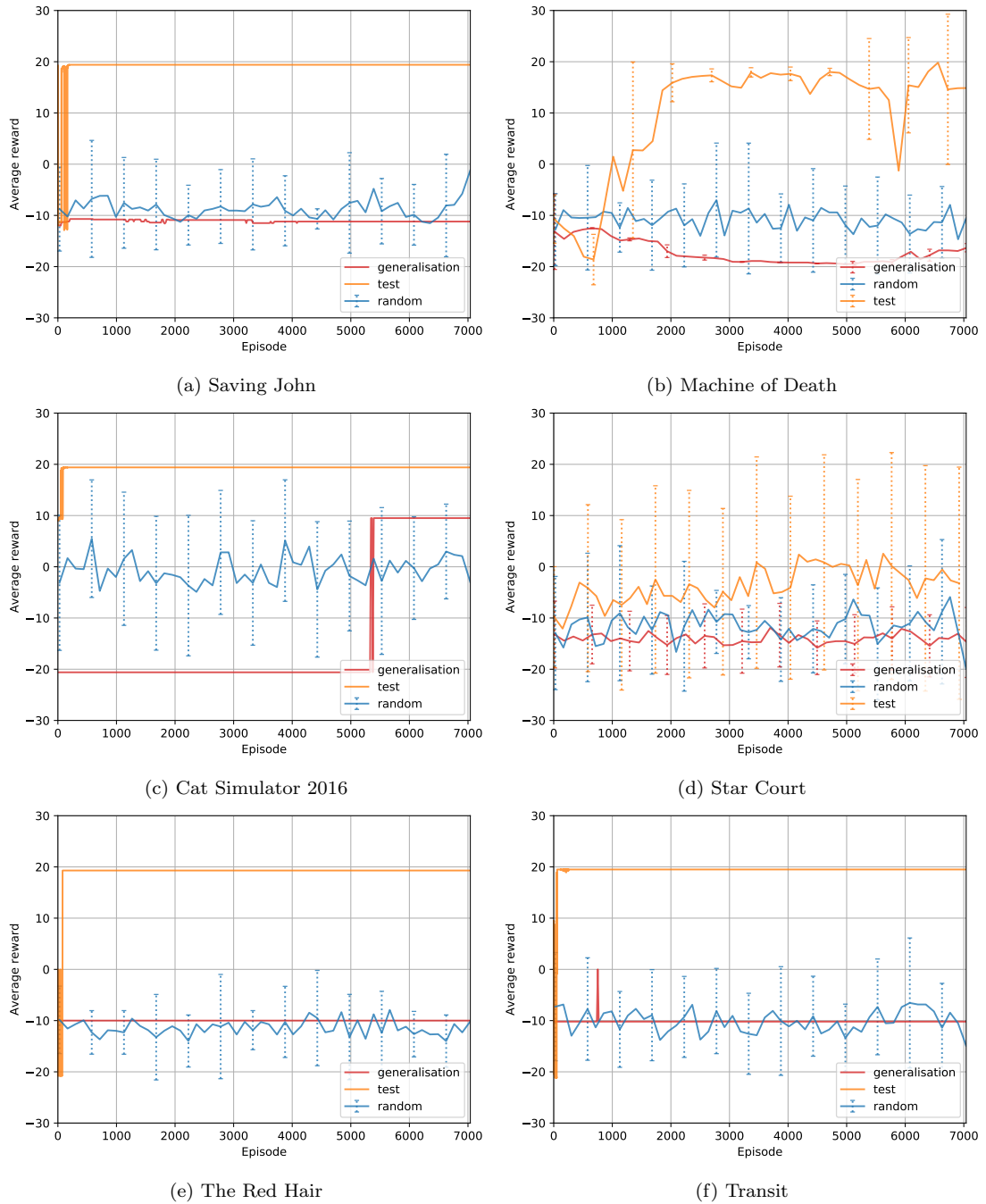


Figure 4.3.1: Generalisation performance.

4.3.3 Discussion

Unfortunately, the agent cannot generalise on the given dataset.

The result is not surprising, and as we can also see from the flat learning curves, the process of training on different games has hardly any impact on the performance in the unseen game.

This is partly due to the incomplete coverage of the state and action natural language spaces of the train and test domains, i.e. some words of the tested game do not appear at all in the training games. Or, even if they are present in at least one training game (see table 4.4), their appearance therein might be minimal or insignificant and consequently, without much impact on the specific word embedding.

	SJ	MoD	CS	SC	TRH	TT
% tokens present in other games	68.4	56.0	79.4	33.7	92.3	72.7

Table 4.4: Shared vocabulary statistics. A token is shared between a game and other games if it is present in at least one other game.

Most importantly, though, generalisation is hard in IF games because the individual games do not necessarily share objectives and actions that lead to good rewards in one game are often very bad choices in another.

IF games are works of art created by humans and as such, there is no universal correct decision-making policy valid in different games and different contexts, as it obviously is very common for humans to have different opinions on what action is the right choice in a given context.

We hypothesise that even if a set of games shared the natural language state and action spaces and we tested a well-trained model on a previously unseen game, the resulting performance would — on average — not be significantly better than that of a random agent.

As discussed in section 4.2.3, another factor that supports this hypothesis is the fact that there are comparatively few decisions to make in a single game run and only failing once is often sufficient to fail completely.

Thus, the impact of individual decisions is very high and it is very likely that, when generalising, at least one “correct” decision in the new game will not be selected by the policy learned on different games.

Paraphrased actions

He et al. [2015] used a modified version of *Machine of Death* for testing generalisation abilities of their model.

In the alternative version, all actions descriptions are paraphrased in natural language and the state descriptions remain unchanged.

The DRRN agent trained on the original actions was shown to generalise well to the paraphrased dataset, reaching a reward of 10.5 as compared to the original reward of 11.2.

We were unable to reproduce this behaviour with our model, or, more precisely, we were unable to reach this performance *consistently*.

In reality, different instances of our model pre-trained on the original actions that reached almost identical performance in the original game, performed very differently in the paraphrased version of the game. Some were on par with the original model, reaching better generalisation results than DRRN, but some performed significantly worse.

This leads us to believe that the evaluation of such experiments should be done in a continuous manner, similarly to how we evaluate our generalisation experiments, i.e. by continuously monitoring the test performance on the paraphrased dataset while training on the original dataset and thus averaging the results.

The performance of our agent while following exactly this approach can be seen in figure 4.3.2:

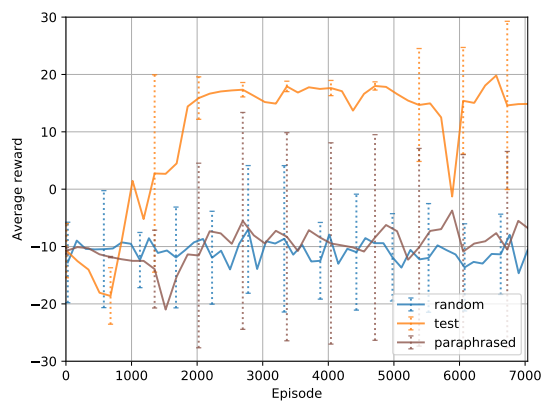


Figure 4.3.2: Generalisation on paraphrased actions in *Machine of Death*.

Unfortunately, the agent was able to generalise somewhat successfully in only one of the three game branches.

Generally speaking, though, it is very important to evaluate generalisation experiments carefully. Especially in the context of IF games that are often volatile (see section 4.2.3), it is common to simply “get lucky” and find an instance of a policy resulting in a high reward despite not actually being able to understand the underlying text.

In fact, we run a random agent on *Machine of Death* and observe that it receives a positive cumulative reward in 8.12% of games across 10000 simulation episodes.

4.4 Transfer learning

Following the generalisation experiment, we now test the agent’s transfer learning ability, i.e. the ability to make use of the knowledge obtained on a different dataset when being confronted with a new domain.

Even if the agent was not able to generalise, it is still possible that its obtained knowledge will eventually lead to better results when facing a different environment.

4.4.1 Setting

In this experiment, the weights of the model are not initialised to random values.

Instead, the agent is given the model trained on five different games in the generalisation experiment (see section 4.3) and we start training it on the sixth, previously unseen game.

We also use the same unified vocabulary as in the generalisation experiment (see section 4.3.1).

We used the same parameters as in the generalisation experiment, i.e. a learning rate of 0.00001 and ϵ -decay of 0.999.

4.4.2 Results

For results, see figure 4.4.1.

The performance is, in most cases, slightly worse than in the task of playing individual games.

4.4.3 Discussion

For the most part, transfer learning on the simple games leads to a slower learning process with identical final results (the agent still converges to an optimal policy). This is likely because instead of starting with small random weights, the model is now trained on five different games and, instead of making use of the pre-trained knowledge, the agent first has to “forget” and override its information about different games, making the task even more difficult.

The only exception is *The Red Hair* (fig. 4.4.1e), in which an optimal policy is found immediately in transfer learning. We hypothesise that this is the case thanks to its almost complete vocabulary coverage as seen in table 4.4.

In both *Machine of Death* and *Star Court*, the performance is significantly worse than in the individual task, both due to their sheer size (resulting in the pre-trained model overfitting on the simple games) and the specificity of their vocabularies.

To summarise, the agent only displays little ability to transfer previously obtained knowledge to a similar but previously unseen dataset.

However, on a single game with a high percentage of shared tokens, transfer learning does yield better results than individual learning, although further testing on different datasets would be needed to confirm that this is the expected behaviour.

Lastly, while not necessarily benefiting the learning process, transfer learning at the very least does not render the agent unable to learn the simpler games at all. Instead, the process is only slowed due to overfitting on previous data.

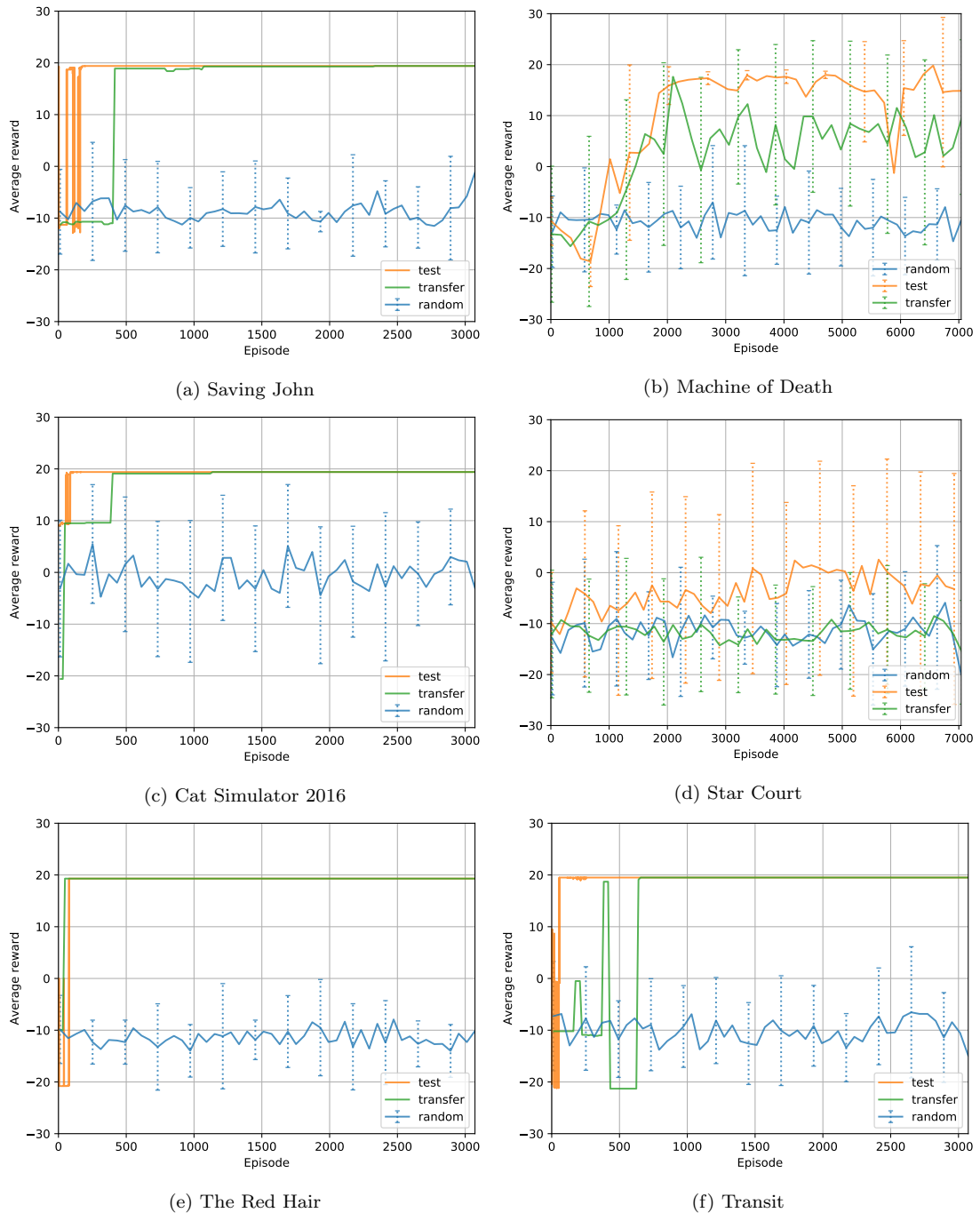


Figure 4.4.1: Transfer learning performance.

4.5 Playing multiple games at once

Finally, we let the agent play all six games simultaneously, meaning both training and testing phases are executed on all games at the same time. More precisely, a *single instance* of the agent is used for this purpose

This is a challenging task, since it requires the agent to integrate and combine the knowledge obtained from playing different text games into its limited model.

The first interesting question is whether the information from all six games can be condensed into such a small model, i.e. whether the agent will be able

learn to play the games simultaneously at all.

Now if that would be the case, we would then be interested in the differences between learning curves of individual games when compared to the simple one game learning task.

Additionally, if the agent was not able to learn a game in an individual, per-game setting, will the universal task perhaps enhance its performance on the previously too difficult game?

4.5.1 Setting

Similarly to the generalisation experiment (section 4.3), we use a learning rate of 0.00001 and ϵ -decay of 0.999.

The agent also uses the unified vocabulary of all games (see section 4.3.1).

When following the algorithm 2, the agent samples each game once in each episode when following the exploration policy and storing experience data.

Consequently, the number of experience tuples of different games in the replay memory \mathcal{D} should be proportionate to their respective average number of steps.

This is convenient, as the more complex games should then be sampled more often for learning.

4.5.2 Results

See figure 4.5.1 for a comparison of all games on this task only and refer to figure 4.5.2 for the usual format where we visualise multiple experiments for each game separately.

The `universal` label corresponds to the multiple-games experiment.

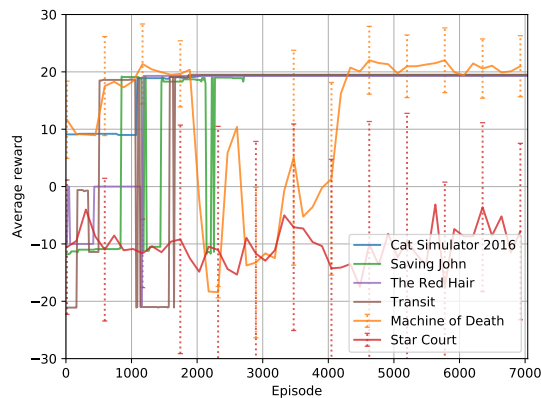


Figure 4.5.1: Comparison of performance in all games in the multiple-games learning task.

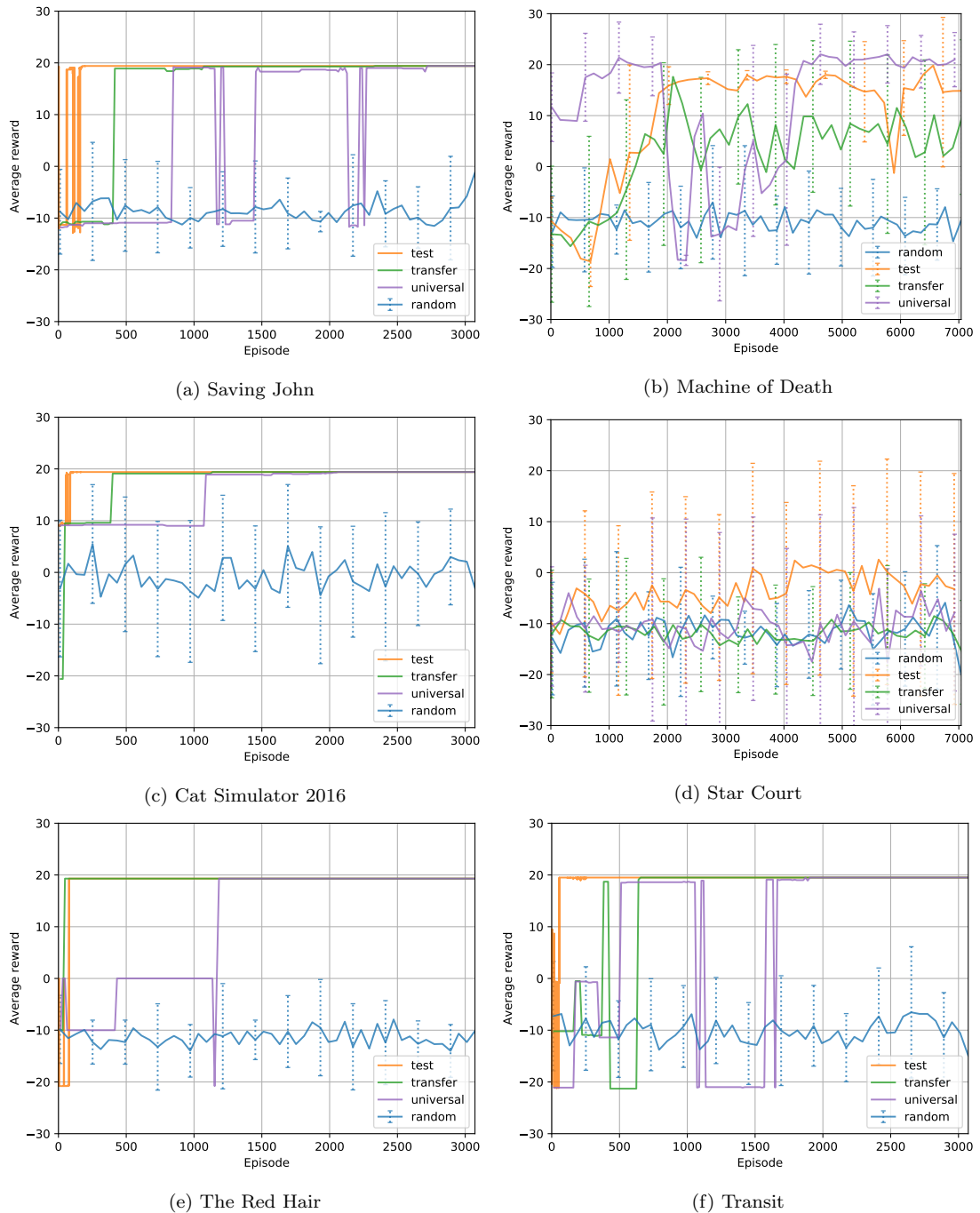


Figure 4.5.2: Performance on the universal task — playing multiple games at once.

4.5.3 Discussion

First of all, the agent was able to learn five games similarly to the individual task (see section 4.2).

The convergence rate on the simple games was considerably slower than in the individual and transfer learning settings, suggesting that at first, the agent struggled to compress the knowledge required to play multiple games.

Ultimately, the agent still found a single policy that is optimal in all four simple games, which is a very good result.

For *Star Court*, training on multiple games unfortunately did not result in better performance; the opposite was in fact the case. This is to be expected, though, as not only was the individual *Star Court* task difficult in itself, but we could also see in section 4.4 that learning from other games does not help if the majority of vocabulary is not shared between the game input data.

The most interesting result, though, is that the agent was now able to converge on *Machine of Death*, even to a nearly optimal reward (see figure 4.5.2b).

The optimal reward is equal to about 21.4 (see table 4.1 and section 4.1.1) and the agent converged to a reward of 21.0. For comparison, the best version of the DRRN agent converged to 11.2 in the individual-game task.

We hypothesise that the improvement in the multiple-game setting over the individual-game setting exists because the simpler games act as regularisers for the neural network.

It is possible that instead of overfitting on two of the three game branches of *Machine of Death*, as was likely the case in section 4.2, the network is now forced to condense its information to the point that information from other games positively influence the agent’s decisions in *Machine of Death*.

Moreover, as suggested by the very slow convergence on the four simple games, it is likely that the higher rate of information compression results in better feature distillation and consequently, to a better knowledge transfer from the simpler games to *Machine of Death*.

At any rate, similarly to all our experiments, further testing with more data is needed to verify our hypotheses.

We can still safely conclude that the SSAQN agent is capable of playing multiple games at the same time whilst only using a minimal architecture with shared representation layers for states and actions. More precisely, the universal agent is able to find a policy that is optimal in several different games.

Additionally, the agent performs better on a complex, non-deterministic game in the multiple-games setting than when only learning it individually, even converging to a nearly optimal average reward³.

This suggests that the agent is capable of transfer learning, although it seems that a high level of knowledge compression is required for the agent to display this ability.

³The final average reward is better than that of human players in *Machine of Death*, as He et al. [2015] recorded a performance of 16.0 for experienced players.

Future work

We did not systematically search for optimal hyper-parameters for our agent model and it is likely that a better performance could be reached by their tuning.

However, achieving the best possible results on the presented environments is not the primary goal. Instead, the most interesting and relevant topic in the domain of IF games is the agent’s ability to generalise and it is very likely that in order for the artificial IF agents to display good generalisation properties, a much larger volume of IF games would be needed.

We thus believe that the major challenge lies in expanding the set of available IF games by adding more environments, resulting in a bigger overlap between the natural language space of the individual IF games as well as between the domains of IF games and natural dialogues.

This would likely lead to both better generalisation and transfer learning capabilities of models similar to SSAQN (Bajgar et al. [2016]).

Another interesting direction of future research might include learning from IF game manuals or perhaps even from game transcripts of human players, similarly to Branavan et al. [2014] or Williams et al. [2017].

For a lot of popular IF games, transcripts are available on the *ClubFloyd*⁴ website, where players meet regularly to play IF games together. The game traces are archived and publicly available⁵ and could consequently be used as a basic source of guidance data.

⁴ClubFloyd information and history: <http://www.ifwiki.org/index.php/ClubFloyd>.

⁵Transcripts are published on http://www.allthingsjacq.com/interactive_fiction.html#clubfloyd.

Conclusion

In this thesis, we introduced the task of learning control policies for playing text-based games, in which state and action descriptions are given in natural language.

The text-game learning task is rather challenging and the ability to act optimally in complex text games could potentially have a positive impact on solving interesting real-world tasks such as comprehending and answering in natural dialogues.

Similarly to the work presented in Narasimhan et al. [2015] and He et al. [2015], we employed deep reinforcement learning techniques based on the DQN algorithm (Mnih et al. [2015]) to build an agent theoretically capable of capturing high-level features of the game descriptions.

Our goal was to design a general architecture with shared representations for both states and actions, serving as a minimalistic proof of concept.

The resulting agent model only employs one recurrent and one dense layer in its underlying neural network that realises feature extraction from text descriptions and it performs similarly to or better than the DRRN model from He et al. [2015] on two games presented therein.

Importantly, we note that the main challenge in similar language-based sequential decision-making tasks does not lie in maximising performance on a single game, but rather in being able to generalise to previously unseen data.

For this purpose, we present `pyfiction`, an open-source library enabling universal access to different text games.

The library includes several different games on which we evaluated our agent mainly in terms of its generalisation and transfer learning capabilities.

We show that it is very difficult to generalise in text games, especially on a relatively small number of games. Even more importantly, it is difficult to correctly assess generalisation properties of a model in this context as text games are very sensitive to subtle changes of the agent’s policy.

Finally, we test our agent on the universal task of playing multiple text games at once. We observe that a single instance of the agent is capable of simultaneously playing multiple games optimally and that this task leads to an even better performance on a non-deterministic game when compared to learning it individually, suggesting the agent’s ability to transfer knowledge.

Additionally, the source code of our agent model is publicly available as a part of `pyfiction` and we hope that both the model and the text-game interface that `pyfiction` introduces could serve as a baseline for future research.

Bibliography

- Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL <http://tensorflow.org/>. Software available from tensorflow.org. [Cited on pages 33 and 64]
- Ondrej Bajgar, Rudolf Kadlec, and Jan Kleindienst. Embracing data abundance: Booktest dataset for reading comprehension. *CoRR*, abs/1610.00956, 2016. URL <http://arxiv.org/abs/1610.00956>. [Cited on page 51]
- Richard Bellman. *Dynamic programming*. Courier Corporation, 2013. [Cited on page 18]
- Yoshua Bengio, Patrice Y. Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Trans. Neural Networks*, 5(2):157–166, 1994. doi: 10.1109/72.279181. URL <https://doi.org/10.1109/72.279181>. [Cited on page 22]
- S. R. K. Branavan, David Silver, and Regina Barzilay. Learning to Win by Reading Manuals in a Monte-Carlo Framework. *CoRR*, abs/1401.5390, 2014. URL <http://arxiv.org/abs/1401.5390>. [Cited on pages 8 and 51]
- Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. *CoRR*, abs/1606.01540, 2016. URL <http://arxiv.org/abs/1606.01540>. [Cited on pages 4 and 13]
- Jane Bromley, Isabelle Guyon, Yann LeCun, Eduard Säckinger, and Roopak Shah. Signature verification using a siamese time delay neural network. In Jack D. Cowan, Gerald Tesauro, and Joshua Alspecter, editors, *Advances in Neural Information Processing Systems 6, [7th NIPS Conference, Denver, Colorado, USA, 1993]*, pages 737–744. Morgan Kaufmann, 1993. URL <http://papers.nips.cc/paper/769-signature-verification-using-a-siamese-time-delay-neural-network>. [Cited on page 27]
- François Chollet et al. Keras. <https://github.com/fchollet/keras>, 2015. [Cited on pages 33 and 64]
- Yann N. Dauphin, Harm de Vries, Junyoung Chung, and Yoshua Bengio. RMSPprop and equilibrated adaptive learning rates for non-convex optimization. *CoRR*, abs/1502.04390, 2015. URL <http://arxiv.org/abs/1502.04390>. [Cited on page 31]

- Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. <http://www.deeplearningbook.org>. [Cited on page 21]
- Alex Graves, Abdel-rahman Mohamed, and Geoffrey E. Hinton. Speech recognition with deep recurrent neural networks. *CoRR*, abs/1303.5778, 2013. URL <http://arxiv.org/abs/1303.5778>. [Cited on page 22]
- Ji He, Jianshu Chen, Xiaodong He, Jianfeng Gao, Lihong Li, Li Deng, and Mari Ostendorf. Deep Reinforcement Learning with an Unbounded Action Space. *CoRR*, abs/1511.04636, 2015. URL <http://arxiv.org/abs/1511.04636>. [Cited on pages 3, 4, 6, 7, 12, 13, 18, 25, 26, 27, 35, 36, 38, 41, 44, 50, 52, 57, 58, and 60]
- Robert Hecht-Nielsen et al. Theory of the backpropagation neural network. *Neural Networks*, 1(Supplement-1):445–448, 1988. [Cited on page 21]
- Sepp Hochreiter. Untersuchungen zu dynamischen neuronalen netzen. *Diploma, Technische Universität München*, 91, 1991. [Cited on page 22]
- Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 1997. doi: 10.1162/neco.1997.9.8.1735. URL <https://doi.org/10.1162/neco.1997.9.8.1735>. [Cited on pages 22 and 58]
- Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251–257, 1991. doi: 10.1016/0893-6080(91)90009-T. URL [https://doi.org/10.1016/0893-6080\(91\)90009-T](https://doi.org/10.1016/0893-6080(91)90009-T). [Cited on page 20]
- Anna Huang. Similarity measures for text document clustering. In *Proceedings of the sixth new zealand computer science research student conference (NZCSRSC2008)*, Christchurch, New Zealand, pages 49–56, 2008. [Cited on page 30]
- Dan Jurafsky and James H. Martin. *Speech and language processing: an introduction to natural language processing, computational linguistics, and speech recognition, 2nd Edition*. Prentice Hall series in artificial intelligence. Prentice Hall, Pearson Education International, 2009. ISBN 9780135041963. URL <http://www.worldcat.org/oclc/315913020>. [Cited on page 3]
- Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014. URL <http://arxiv.org/abs/1412.6980>. [Cited on page 21]
- Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781, 2013. URL <http://arxiv.org/abs/1301.3781>. [Cited on page 24]
- Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing Atari with Deep Reinforcement Learning. *CoRR*, abs/1312.5602, 2013. URL <http://arxiv.org/abs/1312.5602>. [Cited on page 12]
- Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis

- Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015. doi: 10.1038/nature14236. URL <https://doi.org/10.1038/nature14236>. [Cited on pages 12, 17, 18, 25, 27, 32, 52, and 58]
- George E Monahan. State of the art—a survey of partially observable markov decision processes: Theory, models, and algorithms. *Management Science*, 28(1):1–16, 1982. [Cited on page 10]
- Nick Montfort. *Twisty Little Passages: An Approach to Interactive Fiction*. MIT Press, Cambridge, MA, USA, 2005. ISBN 0262633183. [Cited on page 3]
- Karthik Narasimhan, Tejas D. Kulkarni, and Regina Barzilay. Language Understanding for Text-based Games Using Deep Reinforcement Learning. *CoRR*, abs/1506.08941, 2015. URL <http://arxiv.org/abs/1506.08941>. [Cited on pages 3, 4, 6, 13, 18, 24, 25, 27, and 52]
- Christopher Olah. Understanding LSTM networks, 2015. URL <http://colah.github.io/posts/2015-08-Understanding-LSTMs/>. [Cited on pages 22 and 57]
- Hamid Palangi, Li Deng, Yelong Shen, Jianfeng Gao, Xiaodong He, Jianshu Chen, Xinying Song, and Rabab K. Ward. Deep sentence embedding using long short-term memory networks: Analysis and application to information retrieval. *IEEE/ACM Trans. Audio, Speech & Language Processing*, 24(4):694–707, 2016. doi: 10.1109/TASLP.2016.2520371. URL <https://doi.org/10.1109/TASLP.2016.2520371>. [Cited on page 24]
- Frank Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, pages 65–386, 1958. [Cited on page 20]
- Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized Experience Replay. *CoRR*, abs/1511.05952, 2015. URL <http://arxiv.org/abs/1511.05952>. [Cited on page 32]
- Satinder P. Singh, Andrew G. Barto, and Nuttapon Chentanez. Intrinsically motivated reinforcement learning. In *Advances in Neural Information Processing Systems 17 [Neural Information Processing Systems, NIPS 2004, December 13-18, 2004, Vancouver, British Columbia, Canada]*, pages 1281–1288, 2004. URL <http://papers.nips.cc/paper/2552-intrinsically-motivated-reinforcement-learning>. [Cited on page 32]
- Satinder P. Singh, Richard L. Lewis, Andrew G. Barto, and Jonathan Sorg. Intrinsically motivated reinforcement learning: An evolutionary perspective. *IEEE Trans. Autonomous Mental Development*, 2(2):70–82, 2010. doi: 10.1109/TAMD.2010.2051031. URL <https://doi.org/10.1109/TAMD.2010.2051031>. [Cited on page 32]
- Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. LSTM neural networks for language modeling. In *INTERSPEECH 2012, 13th Annual Conference of the International Speech Communication Association, Portland, Oregon, USA, September 9-13, 2012*, pages 194–197. ISCA, 2012. URL <http://>

- www.isca-speech.org/archive/interspeech_2012/i12_0194.html. [Cited on page 22]
- Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning - An Introduction*. Adaptive computation and machine learning. MIT Press, 1998. ISBN 0262193981. URL <http://www.worldcat.org/oclc/37293240>. [Cited on pages 15, 17, 18, 19, 31, 32, and 57]
- Gerald Tesauro. Temporal difference learning and td-gammon. *Commun. ACM*, 38(3):58–68, 1995. doi: 10.1145/203330.203343. URL <http://doi.acm.org/10.1145/203330.203343>. [Cited on page 17]
- Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012. [Cited on pages 21 and 31]
- Christopher J. C. H. Watkins and Peter Dayan. Technical Note Q-Learning. *Machine Learning*, 8:279–292, 1992. doi: 10.1007/BF00992698. URL <https://doi.org/10.1007/BF00992698>. [Cited on page 18]
- Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990. [Cited on page 22]
- Paul John Werbos. Beyond regression: New tools for prediction and analysis in the behavioral sciences. *Doctoral Dissertation, Applied Mathematics, Harvard University, MA*, 1974. [Cited on page 21]
- Jason D. Williams, Kavosh Asadi, and Geoffrey Zweig. Hybrid code networks: practical and efficient end-to-end dialog control with supervised and reinforcement learning. *CoRR*, abs/1702.03274, 2017. URL <http://arxiv.org/abs/1702.03274>. [Cited on page 51]

List of Figures

1.1	<i>Zork I</i> , a classic IF game ⁶	5
1.2	Parser, choice and hypertext-based games (He et al. [2015]).	6
1.3	Part of <i>Zork I</i> 's game map ⁷	9
2.1	Interaction between the agent and the environment (Sutton and Barto [1998]).	17
2.2	A simple RNN (Olah [2015]).	22
3.1	Architecture of the SSAQN model and its data flow.	28
4.2.1	Performance of the SSAQN agent in individual games.	40
4.2.2	Comparison of SSAQN and DRRN on Saving John and Machine of Death. . .	41
4.3.1	Generalisation performance.	43
4.3.2	Generalisation on paraphrased actions in <i>Machine of Death</i>	45
4.4.1	Transfer learning performance.	47
4.5.1	Comparison of performance in all games in the multiple-games learning task.	48
4.5.2	Performance on the universal task — playing multiple games at once.	49
B.1	Keras implementation of the SSAQN model.	64

List of Tables

4.1	Summary of game statistics.	36
4.2	Parameters used in the learning tasks.	38
4.3	Learning parameters for the individual games task.	39
4.4	Shared vocabulary statistics.	44
A.1	Cat Simulator 2016 rewards.	60
A.2	Star Court rewards.	61
A.3	The Red Hair rewards.	62
A.4	Transit rewards.	62
A.5	Game properties and agent performance summary.	63

List of Abbreviations

IF	interactive fiction
IFDB	Interactive Fiction Database, http://ifdb.tads.org/
CYOA	choose your own adventure

MDP	Markov decision process
AI	artificial intelligence
RL	reinforcement learning
(A)NN	(artificial) neural network
MLP	multilayer perceptron
MSE	mean squared error
SGD	stochastic gradient descent
RNN	recurrent neural network
LSTM	long short-term memory (a form of a RNN, Hochreiter and Schmidhuber [1997])
BPTT	backpropagation through time
NLP	natural language processing
BOW	bag of words
DQN	Deep Q-Network (Mnih et al. [2015])
DRRN	Deep Reinforcement Relevance Network (He et al. [2015])
SSAQN	Siamese State-Action Q-Network (chapter 3)

SJ	Saving John
MoD	Machine of Death
CS	Cat Simulator 2016
SC	Star Court
TRH	The Red Hair
TT	Transit

Appendices

A. Text games

In this appendix, we

(i) present a table on page 63 that summarises the majority of relevant properties of the different games as well as the results of the SSAQN agent in these environments;

(ii) provide complete information about the rewards assigned to different IF games we used for evaluating our agent model.

To recapitulate, we utilised the following IF games: *Saving John*¹, *Machine of Death*², *Cat Simulator 2016*³, *Star Court*⁴, *The Red Hair*⁵ and *Transit*⁶.

For numerical rewards associated with the different game endings of *Saving John* and *Machine of Death*, see He et al. [2015] whose values we use and whose text game simulator is wrapped by pyfiction (see appendix B), resulting in an unified interface for all employed games.

Rewards assigned to different endings

For determining the ending type, we usually only detect and present substrings of the final states.

The rules are evaluated in the order given by the table.

In pyfiction’s source code, the reasoning behind the reward value is usually given for the corresponding reward state in each game simulator.

Cat Simulator 2016

The rewards are based on whether the cat was able to get food and whether it found a good place to sleep.

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

Table A.1: Cat Simulator 2016 rewards.

¹SJ: <http://ifdb.tads.org/viewgame?id=1vzv5y27vixinm22>

²MoD: <http://ifdb.tads.org/viewgame?id=u212jed2a71jg6h1>

³CS: <http://ifdb.tads.org/viewgame?id=79f1ic623cvxtpio>

⁴SC: <http://ifdb.tads.org/viewgame?id=u1v4q16f7gujdb2g>

⁵TRH: http://textadventures.co.uk/games/view/r0fika63aksao_qkq8n31q/the-red-hair

⁶TT: <http://ifdb.tads.org/viewgame?id=stkrrel8m21b37q>

Star Court

The rewards are based on the fact if the player survived; if they did, but got sentenced, the reward scales with the imprisonment time in years. Positive rewards are obtained by either escaping the prison or by “living happily ever after”, with bonuses to using favours to improve quality of life.

The X variable refers to the length of imprisonment, $X \leq 2000$.

Ending substring	Reward
You get a job as a	5
Here on the astral plane, your psychic bodies are as physical	-20
Nah. You die as poison consumes your body. And because you failed trial by poison	-30
You're all out of favors! I guess working as	15
The only thing Pride finds more beautiful than itself is the destruction	-30
Immediately upon starting the battle, the titanic creature falls asleep	-30
You are torn limb from limb by the many-limbed creature!	-30
You remember you training at Psi City and concentrate	-30
And so you do, spacer, so you do.	15
BLAMMO!! You're dead! And what's worse, you're guilty!	-30
The Judge bangs their laser gavel a final time. "Robailiff, you may take the prisoner	$-X/100$
You're dead! I guess that means you're guilty!	-30
You are neither guilty nor innocent, as law has been dethroned in the universe.	-20
You let Star Court evaporate like a bad memory. You're on the other side	10
How does Star Court generate this much trash, you think as you burn.	-20
You got smoked by a crime ghost.	-20
Congratulations, you're innocent! You're also dead.	-20
The knife hits you right between the eyes. You are killed immediately, means you're guilty!	-30

Table A.2: Star Court rewards.

The Red Hair

The rewards are based on whether the player accepts the job and whether they are able to protect the children.

Ending substring	Reward
you lose	-10
all there is left is a red hair	-20
it was the clown statue missing	-20
you stay in the bedroom and eventually the parents come back and thank you	20

Table A.3: The Red Hair rewards.

Transit

Positive rewards are for tackling the right man; negative rewards are assigned to the endings in which the player is arrested or dies.

Ending substring	Reward
if anyone can help you	10
you buy one more can	-20
even though it was just in-passing	20
you make swift use of	-20
the guards know	-10
as you predicted	-10
you close your eyes and submit to death	-20
you're in a country	-10
through the haze of the drinks	10
while the last parts of your mind untouched	-10

Table A.4: Transit rewards.

		Saving John	Machine of Death	Cat Simulator 2016	Star Court	The Red Hair	Transit
PROPERTIES	# tokens	1119	2055	364	3929	155	575
	# states	70	≥ 200	37	≥ 420	18	76
	# endings	5	≥ 14	8	≥ 17	7	10
	Avg. words/description	73.9	71.9	74.4	66.7	28.7	87.0
	Deterministic transitions	Yes	No	Yes	No	Yes	Yes
	Deterministic descriptions	Yes	Yes	Yes	No	Yes	Yes
FINAL REWARD	(4.2) Random agent (average)	-8.6	-10.8	-0.6	-11.6	-11.4	-10.1
	(4.2) Individual game	19.4	15.4	19.4	-2.2	19.3	19.5
	(4.3) Generalisation	-11.2	-15.1	5.7	-13.2	-10.0	-10.2
	(4.4) Transfer learning	19.4	8.7	19.4	-13.3	19.3	19.5
	(4.5) Multiple games	19.4	21.0	19.4	-8.2	19.3	19.5
	(4.1.1) Optimal	19.4	≈ 21.4	19.4	?	19.3	19.5

Table A.5: Summary of game statistics and performance comparison of the SSAQN agent on different tasks.

B. pyfiction

pyfiction is an open-source Python library for simple access to different IF games developed originally for its use in this thesis.

In addition to the text games supported by pyfiction and used in this thesis (see appendix A), the SSAQN agent was also implemented as a part of the library.

Consequently, all experiments conducted in chapter 4 using the SSAQN agent are available as runnable examples in pyfiction.

The complete documentation on how to install pyfiction and run the examples is available on the library’s website:

<https://github.com/MikulasZelinka/pyfiction>.

For future reference to the code base at the time of writing this thesis, we also released version 0.1.0 of pyfiction, always available under its tag:

<https://github.com/MikulasZelinka/pyfiction/releases/tag/v0.1.0>.

SSAQN as implemented in pyfiction

Our implementation of the SSAQN agent is realised in Keras (Chollet et al. [2015]) using the Tensorflow backend (Abadi et al. [2015]).

For purposes of future reference, we provide a visualisation of our implemented model in figure B.1 as plotted by Keras.

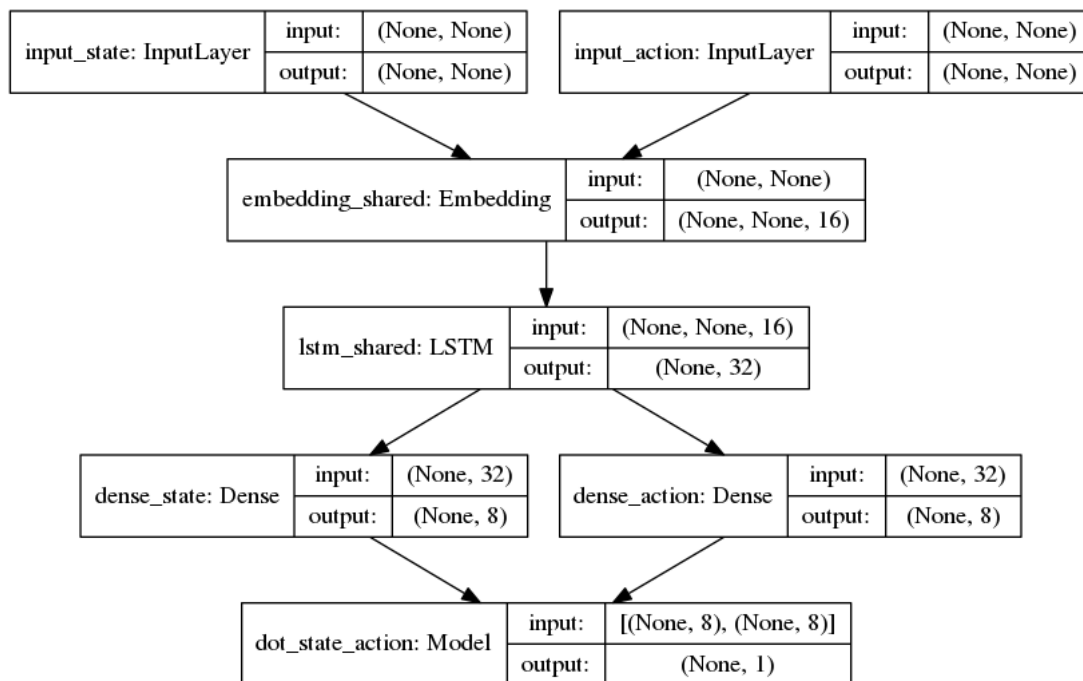


Figure B.1: Keras implementation of the SSAQN model (see figure 3.1). as visualised by `plot_model` in Keras. The first `None` values correspond to the variable batch size. The second `None` values, if present, correspond to the length of the state and action descriptions.