Rogueinabox: an Environment for Roguelike Learning

ANDREA ASPERTI
University of Bologna
Department of Computer Science
Mura Anteo Zamboni 7, Bologna, Italy
ITALY
andrea.asperti@unibo.it

CARLO DE PIERI
University of Bologna
Department of Computer Science
Mura Anteo Zamboni 7, Bologna, Italy
ITALY
carlo.depieri@studio.unibo.it

GIANMARIA PEDRINI
University of Bologna
Department of Computer Science
Mura Anteo Zamboni 7, Bologna, Italy
ITALY
gianmaria.pedrini@studio.unibo.it

Abstract: In this article we introduce Rogueinabox: a highly modular learning environment built around the videogame Rogue, the father of the roguelike genre. It offers easy ways to interact with the game and a whole framework to build, customize, run and analyze learning agents. We discuss the interest and challenges of this game for machine learning and deep learning, and then discuss our initial experiments of training.

Key–Words: Machine Learning, Reinforcement Learning, QLearning, Neural Network, Artificial Intelligence, Rogue, Game

1 Introduction

Roguelike games are an interesting challenge for Q-learning and reinforcement learning. The father of all these dungeon crawling games is Rogue, a game developed around 1980 for Unix-based mainframe systems, with a plain ASCII interface (see Fig. 1). The game was ranked in sixth position in a recent list of PC World of the ”Ten Greatest PC Games Ever” [1], and in spite of its age and the spartan, bi-dimensional interface, the game still exerts an indubitable fascination. We shall discuss in the next section the many features of this game that pose interesting challenges for machine learning. One of the most important aspect is that the ASCII nature of games such as Rogue, and of some of it's spiritual successor like Angband and NetHack, allows to bypass many typical issues related to computer vision (which by now are, thanks to the impressive achievements during the last five years, relatively well understood), resulting in a direct focus on planning and strategy development, the most interesting and complex aspects of automatic learning.

Many game environments suitable for Reinforcement learning already exist, most notable examples are Arcade Learning Environment (ALE [2]), OpenAI Universe [3] [4] and VizDoom [5]. Frameworks for interacting with roguelike games also already exist, such as [6] for Desktop Dungeons and BotHack [7] for NetHack, but none was available for the game of choice of this work: Rogue. Rogueinabox aims to offer a highly modular and configurable environment for Rogue, meant to ease the interaction with the game and the definition of agents.

Our framework features a modular architecture, implementing separately all the main components of our learning environment: agents, experience memory, network models, reward functions, states representation, game evaluation, logging and ui. Each module is easily configurable to suit the user needs and can be extended to quickly modify or add new behavior.

2 Relevance for Machine Learning

In this section we highlight some of the main features of Rogue that makes it an interesting test bench for machine learning and, especially, deep learning.

2.1 POMPD nature

Rogue is a Partially Observable Markov Decision Process (POMPD), since each level of the dungeon is initially unknown, and is progressively discovered as the rogue advances in the dungeon. Solving partially observable mazes is a notoriously difficult and challenging task (see [8] for an introduction). They are often solved with the help of a suitable (built-in) searching strategy, as in [9], that is not particularly satisfying from a machine learning perspective. A Neural Network based reinforcement learning technique to learn memory-based policies for deep memory POMDPs (Recurrent Policy Gradients) have been investigated in [10]. The prospected scenarios are similar

Figure 1: A typical Rogue level.

to those of Rogue: partial knowledge of the model and deep memory requirements, but they considered much simpler test cases.

2.2 No level-replay

In most video games, when the player dies, the game restarts the very same level with the same layout and the same obstacles. Learning in these situations is not particularly difficult, but the acquired knowledge will be useful for that level and that level only, hence learning must be started anew for subsequent levels. As observed in [11], standard CNN-based networks - comprising Deep Q-Networks (DQN) - can be easily trained to solve a given level, but they do not generalize to new tasks.

Rogue has been one of the earlier examples of procedural generated levels, which was one of the main novelty when the game was introduced: every time a game starts or the player dies, a new level gets generated, every time different from the previous ones. Procedural generated content is partially random, maintaining some constrictions (each level will almost always have nine rooms, for example, but the form, the exact position and the connections between them will vary). This means that extensive, level-specific learning techniques could not be deployed, because the player would eventually die, and the dungeon would change. As a consequence, learning must be done at a much higher level of abstraction, requiring the ability to react to a *generic* dungeon, taking sensible actions. Even with a lot of training data covering all possible configurations, and a rich enough policy representation, learning to map each task to its optimal policy in a reactive way looks extremely difficult. A mechanism that *learn to plan* is likely needed, similarly to the value-iteration network (VIN) described in [11].

2.3 ASCII graphics

Rogue is meant to be played in a terminal, therefore renders all its graphics with ASCII characters using the ncurses library (you can see a game screenshot in Fig. 1). This has two consequences: on one side, the simulation is very fast (in comparison with more modern and complex graphic games); on the other side, the information presented on the screen is already coded and differentiated, which makes it easier to parse and reinterpret it (we see no point in deploying OCR techniques to discriminate the different icons).

Another by-product of the architecture used to develop the game is that every single action the player takes results in a single screen update. This one-on-one relationship between an action and the change of the game state makes it easy to implement an action-reward based learning model.

2.4 Memory

In many situations, the rogue need a persistent memory of previous game states and of previous choices in order to perform the correct move. A very simple example is when searching for secret passages in a section of the wall or at the end of a corridor. In this cases, the hidden passage my appear after an arbitrary number (usually between 1 and 10) of pressings of the search button (s) and you need to recall the number of attempts already done. You also need memory in mazes, since you need (at least) to remember the direction you came from to avoid looping (but a more general recollection of past rogue positions would likely improve the behavior and robustness of the agent). Since the discovery of Long-Short Term Memory models (LSTM) [12, 13], the use of memory in neural networks is increasingly popular, providing one of the most active and fascinating frontiers of the current research (see e.g. the recent introduction of Gated Recurrent Units - GRU [14]). LSTM have been already used for in [15] for Atari games, to replace the sequence of states of [16], and are also exploited in [17]. Although in our preliminary experiments we did not use recurrent networks, exploiting instead some simple and explicit forms of memory (see Section 4.1.2), Rogue seems to provide a really interesting test bench for these techniques.

2.5 Attention

Another hot topic in Machine Learning is attention: the ability, so typical of human cognition, to focus on a specific fragment of a scene of particular interest, ignoring others of lesser relevance, to build a sequential interpretation and understanding of the *whole* scene it's being looked at. Clearly, in a game like Rogue, the environment immediately surrounding the character is the main center of attention, and the agent moving the rogue must have a precise knowledge of it without however loosing the whole picture of the map. Many techniques have been recently introduced for addressing attention, comprising e.g. the recent technique of spatial transformers [18], that looks promising for this application, due to the highly geometrical structure of Rogue rooms and corridors. We are also currently investigating a different technique, inspired by *convolutionalization* [19], and essentially based on aggressive use of maxpooling mediated by an image-pyramid vision of the map (see Section 4.1.3).

2.6 Complex and diversified behaviors

Dungeon-like games offer an interesting combination of diversified behaviors: moving around, fighting monsters, descending/escaping the dungeon, acquiring loot, exploiting the equipment in the inventory. Merging together these activities and their learning is a complex problem. As of now, the agent behavior is traditionally divided into two phases, one involving exploring the map, collecting items, finding enemies, and another one for fighting [20, 21, 17]. Each phase is covered by a specialized network, trained in a specific way. Combining together neural models optimized on different tasks is still an open issue in neural systems.

3 Rogueinabox Modules

In this section we explain in detail the different modules of Rogueinabox. The final aim is to have an environment that allows to conveniently build Rogue agents, and especially self-taught agents with autonomous intelligence, based on deep learning and reinforcement learning. The architectural design of Rogueinabox was driven by the wish to obtain a high modularity and configurable system. Rogue is a complex game, based on many different variables, and being able to tune them individually, precisely and with ease is a priority. For this reason each module is freely configurable to suit the user needs, who can also comfortably add his own methods to the already existing library.

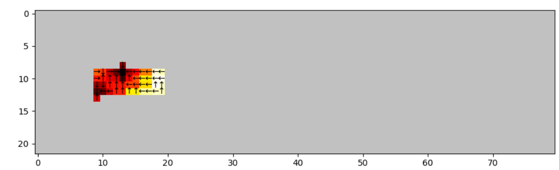


Figure 2: A room heatmap showing the Qvalue in warmer colors. Arrows correspond to the selected action, correctly defining paths leading to doors.

3.1 Rogue wrapper

This library wraps the game itself and is responsible for running, restarting and killing the game process when needed. It sits between Rogue and the rest of the framework and provides methods to send commands to the game, but most of all to parse its response. It offers easy access to status bar information and provides the raw game state (the screen) that will be parsed by the other modules.

3.2 State representation

This module takes raw data from the Rogue wrapper and transform them before passing them to the agent. The shape of the state representation and the amount of information the user might want to give the agent might vary wildly depending on the objective that has to be achieved. For example we might want to hide some information (such as the inventory or the status bar) and focus on solving a simpler problem like moving and fighting. We might also want to vary the shape and the channels of the states, using multiple layers or cropped views.

3.3 Reward functions

This module manages the reward function that will give a score to every agent action. Rogue does not have a proper score system: the final goal is to retrieve the amulet of Yendor, that is very deep in the dungeon; visible state parameters such as the rogue's health and experience change very slowly, and even the gold (and other goods) found along the way provides very sparse rewards. Crafting an agent able to learn from these weak reinforcements is a *really* challenging objective. While this is indeed the final goal, it may be convenient to start addressing simpler scenarios, where the rogue agent is provided with additional rewards, related to the portion of the dungeon being explored, to movement, et similia, aimed to incentivize particular behaviors. The choice of the reward function defines which objective we are pursuing and in which way we are doing it, hence the ability to change it accordingly with our aims is relevant for testing and understanding the agent behavior. The reward module has access to all the raw information presented on the screen (before the state conversion) so its easy to manipulate it and extract whatever data or variation in data we find useful.

3.4 Evaluation

This module takes care of evaluating the overall performance of an agent during a game. It provides hooks to insert in every train or play cycle, and at the end of every game, allowing to calculate a score for the current game. It can be used to grade agents performance, to compare agents as well as to ensure the best weights are always saved. It's possible to easily define new criteria for evaluation, which is not an easy feat and an equally difficult albeit different task than reward definition; this is especially true during early development, since game statistics like gold or dungeon level are poorly portraying the agent doings.

3.5 Network models

This module governs the structure of the neural network that will form the mind of the agent. The model defines how the agent "thinks" and what he sees and focuses on given a particular state. We used Keras [22] as our deep learning framework of choice because of his simple and researcher friendly structure but model construction is abstracted by a model manager, so the user can use whatever framework he likes to build the model and just encapsulate it in an object with a keras like model interface.

3.6 Experience memory

This module manages the agent memory of his past state transitions, which includes actions taken and rewards received. Experience memory has proven to be an extremely valuable tool in reinforcement learning [16]; using this technique is possible to reduce correlation between state transitions. Collecting past experiences also allows to train a different model on an already saved history (given that the state representation is the same) in a time efficient manner. We also provide tools to filter which transition ends up stored into memory, allowing the creation of a more balanced history that better fits the target needs.

3.7 Agents

This module takes care of the different implementations of the agent. We provide 3 base agents; an user controlled one, a random agent and a qlearner agent that is capable of training and running models using a deep Q-learning strategy as shown in [16]. As with any other module the user can write his own agent that uses the tools provided by Rogueinabox and implements a learning algorithm of choice.

3.8 Logging

This module manages the logging of the agent actions. Logs can be printed to various streams (stdout, file, the UI...) and filtered by verbosity levels. This module also provides a way to trace the execution time of sections of code; its most notably use is monitoring the speed and performance of the model updates during training. For certain state representations it's also possible to visualize the agent decision and plot them as seen in Fig. 2.

3.9 UI

This module handles the user interface for Rogueinabox. Since the screen updates require time it is recommended to train with the UI turned off and just parse the log file to retrieve information about the current state of a training. Nevertheless sometimes it might be useful to watch what the agent is doing to hunt down bugs or just to see the result of a training in action. We provide two different UI implementation, one is a TK-Inter GUI (for desktop uses) and one is a Curses UI (for remote headless server uses).

4 Training the agent

Hard-coded agents for roguelike games are already available; examples are Rog-o-matic for Rogue [23], Borg [24] for Angband and BotHack [7] for NetHack. Instead, this chapter will explain the steps taken for building and training a Rogue QLearning agent using the Rogueinabox environment.

Rogue is a very challenging game, even for a human; the Q-learning architecture for Atari Games presented in [16] works very bad in this case: comparable to a completely random agent. This is not very surprising: that network only performs well on a specific class of games, requiring from the user prompt reactions to predictable events (see [25]), little (typically mono-dimensional) movement, and no form of planning (see [11]).

The weakness of the CNN-model of [16] in the case of Rogue also appears to be related to the vision structure: the first convolution filter with dimension 8x8 and stride 4 is far away too rough for the kind of very detailed (pixel-based) perception of the environment required by this game.

To have a better grasp of the problems, we focused on the mere task of exploration: enemies, items, inventory and other Rogue features were intentionally left out. Moreover, we addressed problems of increasing complexity like: exiting from the room, traversing corridors, finding the stairs and taking them. All these experiments are documented in the code; in this article we only discuss the current network architecture, obtained as a result of the previous experiments.

4.1 Deep QLearning Agent

4.1.1 Reward function

Since we are not taking money and other goods into account, we adopted a rewarding mechanism typical of mazes (see e.g. [8]): a large reward for the objective (in this case, the stairs), a negative reward for wrong moves (walking through a wall or trying to descend where there are no stairs), and a small negative reward for every other move (the so called "living" reward). Since the map get progressively discovered as the rogue walks through it, it looks natural to also add a positive reward for every new map tile traversed. Finally, we investigated small "movements" rewards, as suggested in [17].

This mechanism of rewards works reasonably well as long as the rogue is driven by the exploration of new portions of the map, but the agent get confused when he needs to turn back, retracing his steps. This is precisely where Q-learning should step in, allowing to take into account a future reward, in spite of many minor negative moves required to reach it. The problem is making sure the agent is able to correctly recognize the configuration providing the reward, and training it to make such association. The complexity of the problem derives from its generality (the rogue could be in any position in the map), and the need to focus attention on the area surrounding the rogue.

This are the main motivations for the tower architecture discussed in Section 4.1.3, combining a local/global vision of the map based on an image pyramid idea, allowing to combine the "what" and the "where".

Moreover, it also looks important to provide the agent with a persistent memory of its past whereabouts, discussed in the next sections.

4.1.2 State representation

Rogue represents its dungeons using a small subset of ASCII characters, that can be naturally regarded as different color channels. This means each icon has its dedicated (true or false) channel: a channel for the rogue, one for doors, one for walls, and so on. For the sake of simplicity, we collapsed a few classes, and omitted those we are not taking into account (e.g. monsters, goods, and a few others). At present, we use the following 80x22 binary maps (re-scaled to 0-255 for the uint8 datatype):

- *Map channel* Representing the currently visible map (true if visited, false otherwise)
- *Player position channel* for the player position
- *Doors positions channel* for doors
- *Stairs positions channel* for the position of the stairs

As mentioned above, the agent need to have some persistent memory of its past whereabouts. In the future, we plan to integrate this component in the network architecture, either by means of recurrent units such as LSTM or GRU, or some different solution (a simple unary convolution over a sequence of maps could possibly suffice). However, for the present, this information is precomputed and offered as an additional input for the agent.

We made experiments with two different kinds of memory (not yet used in conjunction):

- *Heatmap channel* This is a long-term memory providing a heat-map of past positions; the color intensity represents the number of times the agent walked over a tale.
- *Snake-like channel* This is a short-term memory with a fading-away, Snake-like representation of the most recent rogue positions.

Both maps have their own advantages: the long-term map helps to avoid cycling, while the short term map improves mobility.

4.1.3 Network model

After several experiments, all documented in the code, we ended up in a network architecture exploiting three Towers (see Fig. 3) processing the input at different levels of details; their results are merged together and subsequently elaborated via dense layers.

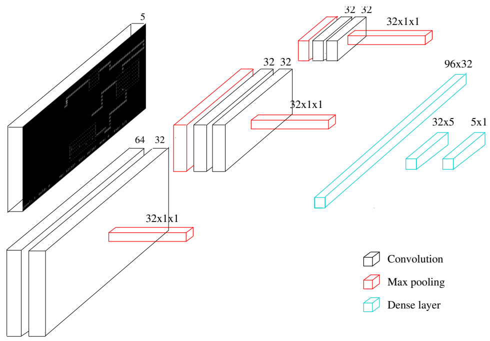


Figure 3: A three Towers model

All towers have a similar mechanism: they perform small (3x3, or 5x5) convolutions on the input and then apply a global maxpooling to focus on the presence/absence of the given feature. The agent learns very rapidly to synthesize features comprising the rogue, and the other entities of interest, hence implicitly focusing its attention on the rogue, with no need to understand his position on the map.

The difference between the various maps is just in the different level of detail at which the input is processed. Currently, this is obtained by a progressive, initial maxpooling of the input, but other techniques can be experimented, in particular progressively augmenting the size and stride of convolutions.

4.1.4 Experience memory

At first we used a standard FIFO strategy for creating experience memory. However, the FIFO queue of past transitions turned out to be full of many useless or replicated states. Moreover the ratio of negative reward transition to positive reward ones was very high, this is because especially in the early stages of training when the exploration is totally random, discovering a new part of the map is hard. Usually the useless and replicated states are negative ones, often the ones in which the agent is stuck on a wall, so this two problems can be solved with a single solution. Instead of a FIFO queue we filtered the value that were being inserted into the history taking all positive values but only a percentage of the negative ones. This new strategy greatly improved results.

4.1.5 Evaluation

Since our focus was on exploration, the evaluation module takes only that into account. It currently computes the score summing all visited tiles on every level the rogue manages to reach. We're using this module mainly during training (a graph with all the scores of a training session can be seen in Fig. 4a) to keep track of the average score and to save the best weights by these standards. Fig. 4b shows how the average score rises during a training of more than two thousand games with three million agent actions. Analyzing the graph is interesting since it shows when the agent had some 'breakthrough' in its learning, with the steep rises in average corresponding to the moments where it learned to consistently exit a room or descend a stairs. It also shows the limit of our current model.

4.2 Training on static memories

Training a DQL agent is a time-expensive operation. Some of this time is taken by the actual training and there is no shortcoming for it, but a huge chunk of it is taken by the environment simulation.

If stored, the transition history created by the agent can be reused over and over for testing different models without simulating the actions again, saving a lot of time.

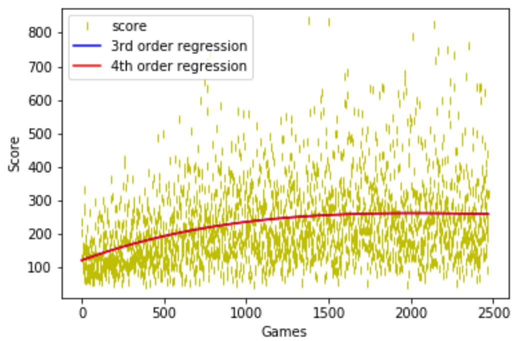
4.3 Rog-o-matic supervised learning

As seen in the last section an agent can be trained using a pre-built history. This pre-built history doesn't have to be built by the Reinforcement learning agent, it can also come from other sources such as human expert players or hard-coded agents. For example, we can use Rogueinabox to run Rog-o-matic [23] an hard-coded agent that has been proved able to win at Rogue.

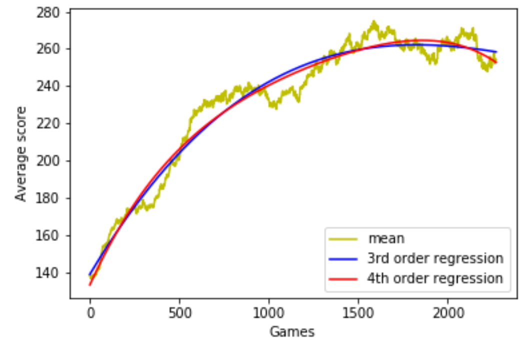
Even if this tool wasn't used for the purposes of this paper the agent is provided with Rogueinabox and could easily be used in later works to improve the behavior of a Reinforcement learning agent.

5 Source Code

Rogueinabox is free, open-source software under the terms of the GNU General Public License. The source code for the agents, state representations, reward functions, network models and all other modules used in our experiments is also available on the git repository page for Rogueinabox github.com/rogueinabox/rogueinabox. Rogueinabox is written in Python, this choice was made both for the ease of use of the language and for the wide amount of deep learning libraries and tools already available. Keras [22] was chosen as a machine learning frame-



(a) Scores



(b) Average score

Figure 4: Agent scores and their average during a training session.

work, supporting both Theano [26] and Tensorflow [27], it stands out for it simple prototyping and ease of use. As of July 2017, we have been working on the codebase (which consists of more than four thousand lines of code) for 5 month.

6 Conclusion and future work

In this work, we introduced Rogueinabox: a new Reinforcement learning environment to interact with the well known and celebrated Rogue game, precursor of all the rogue-like genre. Rogueinabox was extensively tested through the development of a large number of QLearning agents playing the game, who helped driving the evolution and tuning of the tool.

Rogueinabox can be improved in many different ways, and we are especially open to collaboration. More feature and modules can be added; some technical problems in the interface with Rogue, possibly requiring a deeper integration with Rogue's source code, must be solved.

As for the agent themselves, while their behavior is still far from being satisfactory, a set of interesting milestones was achieved. The recently introduced evaluation mechanism will eventually help to better monitor future advancements, in a precise and documentable way.

As we already mentioned, we plan to tackle the memory issue of the agent, either by means of recursive networks (LSTM or GRU) or some different mechanism. We also plan to continue our experimentation of new networks architectures, possibly testing different and more sophisticated forms of attention.

The agent must also be extended to perform new tasks, such as fighting monsters or retrieving equipment, both as isolated objectives or in conjunction with what it already knows.

The training speed could also be improved; faster training speed means faster research and faster advances. We showed how training on a pre-built history can speed up learning (but with increased risk of overfitting), in the future the learning algorithm could also be improved using asynchronous methods and CPU training (instead of GPU), as shown here [28].

References:

[1] B. Edwards, "The ten greatest pc games ever," http://www.pcworld.com/article/158850/best_pc_games.html, 2009.

[2] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents," *J. Artif. Intell. Res. (JAIR)*, vol. 47, pp. 253–279, 2013. [Online]. Available: http://dx.doi.org/10.1613/jair.3912

[3] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai universe," https://github.com/openai/universe, 2016.

[4] ——, "Openai gym," *CoRR*, vol. abs/1606.01540, 2016. [Online]. Available: http://arxiv.org/abs/1606.01540

[5] M. Kempka, M. Wydmuch, G. Runc, J. Toczek, and W. Jaskowski, "Vizdoom: A doom-based AI research platform for visual reinforcement

learning," *CoRR*, vol. abs/1605.02097, 2016. [Online]. Available: http://arxiv.org/abs/1605.02097

[6] V. Cerny and F. Dechterenko, "Rogue-like games as a playground for artificial intelligence–evolutionary approach," in *International Conference on Entertainment Computing*. Springer, 2015, pp. 261–271.

[7] krajj7, "Bothack," https://github.com/krajj7/BotHack, 2015.

[8] R. S. Sutton and A. G. Barto, *Introduction to Reinforcement Learning*, 1st ed. Cambridge, MA, USA: MIT Press, 1998.

[9] M. Wiering and J. Schmidhuber, "Solving pomdps with levin search and EIRA," in *Machine Learning, Proceedings of the Thirteenth International Conference (ICML '96), Bari, Italy, July 3-6, 1996*, L. Saitta, Ed. Morgan Kaufmann, 1996, pp. 534–542.

[10] D. Wierstra, A. Förster, J. Peters, and J. Schmidhuber, "Solving deep memory pomdps with recurrent policy gradients," in *Artificial Neural Networks - ICANN 2007, 17th International Conference, Porto, Portugal, September 9-13, 2007, Proceedings, Part I*, ser. Lecture Notes in Computer Science, J. M. de Sá, L. A. Alexandre, W. Duch, and D. P. Mandic, Eds., vol. 4668. Springer, 2007, pp. 697–706.

[11] A. Tamar, S. Levine, and P. Abbeel, "Value iteration networks," *CoRR*, vol. abs/1602.02867, 2016. [Online]. Available: http://arxiv.org/abs/1602.02867

[12] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997. [Online]. Available: https://doi.org/10.1162/neco.1997.9.8.1735

[13] F. A. Gers, J. Schmidhuber, and F. A. Cummins, "Learning to forget: Continual prediction with LSTM," *Neural Computation*, vol. 12, no. 10, pp. 2451–2471, 2000. [Online]. Available: https://doi.org/10.1162/089976600300015015

[14] J. Chung, Ç. Gülçehre, K. Cho, and Y. Bengio, "Gated feedback recurrent neural networks," in *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, ser. JMLR Workshop and Conference Proceedings, F. R. Bach and D. M. Blei, Eds., vol. 37. JMLR.org, 2015, pp. 2067–2075. [Online]. Available: http://jmlr.org/proceedings/papers/v37/chung15.html

[15] M. J. Hausknecht and P. Stone, "Deep recurrent q-learning for partially observable mdps," *CoRR*, vol. abs/1507.06527, 2015. [Online]. Available: http://arxiv.org/abs/1507.06527

[16] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. A. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015. [Online]. Available: https://doi.org/10.1038/nature14236

[17] G. Lample and D. S. Chaplot, "Playing FPS games with deep reinforcement learning," in *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA.*, S. P. Singh and S. Markovitch, Eds. AAAI Press, 2017, pp. 2140–2146. [Online]. Available: http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14456

[18] M. Jaderberg, K. Simonyan, A. Zisserman, and K. Kavukcuoglu, "Spatial transformer networks," in *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, Eds., 2015, pp. 2017–2025. [Online]. Available: http://papers.nips.cc/paper/5854-spatial-transformer-networks

[19] E. Shelhamer, J. Long, and T. Darrell, "Fully convolutional networks for semantic segmentation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 39, no. 4, pp. 640–651, 2017. [Online]. Available: https://doi.org/10.1109/TPAMI.2016.2572683

[20] M. McPartland and M. Gallagher, "Creating a multi-purpose first person shooter bot with reinforcement learning," in *Proceedings of the 2008 IEEE Symposium on Computational Intelligence and Games, CIG 2009, Perth, Australia, 15-18 December, 2008*, P. Hingston and L. Barone, Eds. IEEE, 2008, pp. 143–150. [Online]. Available: https://doi.org/10.1109/CIG.2008.5035633

[21] B. Tastan, Y. Chang, and G. Sukthankar, "Learning to intercept opponents in first person shooter games," in *2012 IEEE Conference on Computational Intelligence and Games, CIG 2012, Granada, Spain, September 11-14, 2012*. IEEE, 2012, pp. 100–107. [Online]. Available: https://doi.org/10.1109/CIG.2012.6374144

[22] F. Chollet *et al.*, "Keras," https://github.com/fchollet/keras, 2015.

[23] M. L. Mauldin, G. Jacobson, A. Appel, and L. Hamey, "Rog-o-matic: A belligerent expert system," in *Fifth Biennial Conference of the Canadian Society for Computational Studies of Intelligence, London Ontario, May 16, 1984.*, 1984.

[24] B. Harrison. Angband borg. [Online]. Available: http://www.thangorodrim.net/borg.html

[25] M. J. Hausknecht and P. Stone, "The impact of determinism on learning atari 2600 games," in *Learning for General Competency in Video Games, Papers from the 2015 AAAI Workshop, Austin, Texas, USA, January 26, 2015.*, ser. AAAI Workshops, M. Bowling, M. G. Bellemare, E. Talvitie, J. Veness, and M. C. Machado, Eds., vol. WS-15-10. AAAI Press, 2015. [Online]. Available: http://aaai.org/ocs/index.php/WS/AAAIW15/paper/view/9564

[26] Theano Development Team, "Theano: A Python framework for fast computation of mathematical expressions," *arXiv e-prints*, vol. abs/1605.02688, May 2016. [Online]. Available: http://arxiv.org/abs/1605.02688

[27] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/

[28] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," *CoRR*, vol. abs/1602.01783, 2016. [Online]. Available: http://arxiv.org/abs/1602.01783