Learning to solve arithmetic problems with a virtual abacus

Flavio Petruzzellis[1†], Ling Xuan Chen[1†], and Alberto Testolin[*1]

[1]University of Padova, Italy
[†]Equal contribution

Abstract

Acquiring mathematical skills is considered a key challenge for modern Artificial Intelligence systems. Inspired by the way humans discover numerical knowledge, here we introduce a deep reinforcement learning framework that allows to simulate how cognitive agents could gradually learn to solve arithmetic problems by interacting with a virtual abacus. The proposed model successfully learn to perform multi-digit additions and subtractions, achieving an error rate below 1% even when operands are much longer than those observed during training. We also compare the performance of learning agents receiving a different amount of explicit supervision, and we analyze the most common error patterns to better understand the limitations and biases resulting from our design choices.

1 Introduction

Deep learning systems excel in a variety of domains, but struggle to learn cognitive tasks that require the manipulation of symbolic knowledge [1]. This limitation is particularly evident in the field of mathematical cognition [2], which requires to grasp abstract relationships and deploy sophisticated reasoning procedures. Indeed, even state-of-the-art language models fall short in mathematical tasks that require strong generalization capabilities [3] (though very recent work has shown that performance significantly improves following fine-tuning on large-scale mathematical datasets [4]).

One possibility to tackle this challenge is to endow deep architectures with *ad-hoc* primitives specifically designed to manipulate arithmetic concepts [5, 6]. Nevertheless, alternative approaches suggest that symbolic numerical competence could emerge from domain-general learning mechanisms [7, 8]. Recent work has also tried to deploy deep reinforcement learning (RL) to solve math word problems [9]. Notably, deep RL architectures that incorporate copy and alignment mechanisms seem to discover more sophisticated problem solving procedures [10], and could even learn automatic theorem proving when combined with Monte-Carlo tree search algorithms [11].

In this work we explore whether model-free deep RL agents could learn to solve simple arithmetic tasks (expressions involving sum and subtraction) by exploiting an external representational tool, which can be functionally conceived as a *virtual abacus*. Differently from the above-mentioned approaches, our goal is to take advantage of the way humans solve the task by simulating the interaction of the RL agent with an abacus, which can be partially guided through supervised learning mechanisms. This allows teaching the agent existing algorithms for the solution of arithmetic problems, rather than forcing it to discover possible solution strategies only by trial-and-error. Our work is similar in spirit to recent scientific endeavors directed towards the design of learning systems that are capable of the kind of systematic generalization that is required to execute algorithmic procedures [12, 13]. The main challenge of the problem is to discover an algorithm that can be used to solve arithmetic problems, and thus being able to generalize to never-seen instances of the same class of tasks. In a deep RL framework, this even more difficult since rewards could become very sparse due to the length of the solution procedures.

In particular, we are interested in addressing these two key questions: Is it possible to learn to solve mathematical problems that require long-term planning by only relying on model-free RL?

[*]Corresponding Author: alberto.testolin@unipd.it

https://doi.org/10.7557/18.6805
© The author(s). Licensee Septentrio Academic Publishing, Tromsø, Norway. This is an open access article distributed under the terms and conditions of the Creative Commons Attribution license (http://creativecommons.org/licenses/by/4.0/).

If not, what is the minimal amount of guidance (in the form of learning biases and/or explicit supervision) that is necessary to successfully solve such problems?

We find that our agent is able to solve the sum and subtraction problems with a considerable capacity of out-of-distribution (OOD) generalization over the length of the operands. At the same time, our simulations shed light on difficulties in solving mathematical problems with model-free RL, especially when learning requires to plan in the very distant future or to discover solution strategies in which simple steps should be combined to solve arbitrarily complex problems. By systematically analysing the errors made by the agent in the OOD generalization regime, we also provide some intuitions about the functioning and limitations of the proposed learning framework.

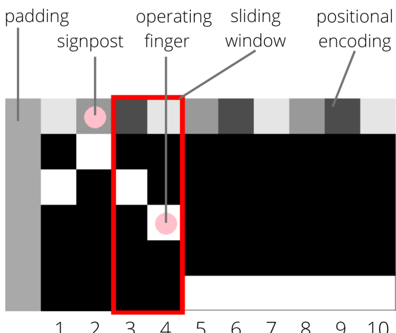

Figure 1: The environment consists of a simulated abacus with 10 columns, a padding on the left and a periodic positional encoding on top. The agent observes the abacus through a sliding window composed of two columns, interacts with it using an 'operating finger' and uses a 'signpost' to mark the column where the current operation is going on.

2 Methods

In this section we describe the task to be solved, the design of the environment with which the agent interacts, the agent architecture, and the training procedure.

2.1 Task

The task of the agent is to compute arithmetic operations between integers, which are provided as a sequence of input symbols that can be either an operation (plus or minus) or an operand. Operands are represented as sequences of digits: during training, the length of the operand is sampled uniformly in $\{1, 2, 3, 4, 5, 6\}$, and each digit is sampled uniformly in $\{0, 1, 2, 3, 4\}$, except for the first digit which cannot be 0. The first operand of any operation is always the one represented in the current state of the abacus, whose initial configuration represents the value 0.

2.2 Environment

The learning environment is conceived as a simulated abacus, featuring 10 columns with 5 positions for each column: that is, we represent numbers in base 5 (see Fig. 1). On the top edge of the abacus there is an additional row representing a periodic positional encoding, which allows to associate each column with a value in the sequence [0.25, 0.5, 0.75, 0.25 ...]. Such additional input allows to effectively encode the position of the fingers in the abacus (as explained below), thus improving learning speed and generalization.

The agent can manipulate the abacus using an 'operating finger', which serves as a pointer positioned over the abacus at a given location at every time step. Furthermore, the agent can use an indicator (dubbed the 'signpost') that can be used to signal the column where the current operation is occurring, and where the operation must resume once a carry operation is over.

The agent partially observes the abacus through a sliding window composed by two columns: the one where the operating finger is, and the one to its left. If the signpost is in the sliding window, the agent can see it in superposition to the positional encoding. In case the operating finger is on the first column, the agent observes a padding instead of the column to the left of the operating finger.

The agent can interact with the environment performing one of the following actions: a movement of the operating finger in the four directions, a movement of the signpost along two directions (left, right), a slide of the beads in the column where the operating finger is currently positioned (dubbed the 'move and slide' action) and an action to signal that it has finished processing the current digit (dubbed

the ‘submit’ action).

When the agent acts, the environment changes its state and gives as output the reward and a flag indicating whether the current episode is over. At each time step, the agent receives as input either an operation symbol or the next digit that should be processed, both represented as one-hot vectors. The operation is also signalled using a flag throughout the duration of the operation. The episode ends if the abacus reaches its maximum representational capacity, or if terminated early by the environment (see section 2.4).

2.3 Agent’s architecture

We simultaneously train an actor and a critic network. Both are memory-less feed-forward networks and thus receive as input a stack of the last 3 observations, processed by a feature extractor implemented as a 3D convolutional network with three layers of 128, 255 and 512 channels, respectively. We use kernels of size 3 in the first layer and size 2 in the last two, all with padding of 1. The output of the feature extractor is then concatenated with the one-hot encoded symbol received from the environment; the critic also receives as input the previous action. Both networks then process this input via 5 feed-forward layers with 2048, 1024, 512, 256 and 128 neurons, respectively[1]. Finally, the actor network returns a probability distribution over the actions, while the critic returns an estimate of the value of each action.

2.4 Reward function

We define specific algorithms to solve the sum and subtraction problems using the virtual abacus, and use such ideal solutions to provide supervision to the learning agent (pseudo-code is provided in Algorithm 1)[2]. In order to encourage the agent to learn such algorithms, we designed a modular reward function that makes it possible to provide an increasing amount of supervision through the following feedbacks:

- A penalty of -0.05 for each action performed, to discourage long sequences of actions.
- A reward of 0.10 whenever a movement action (left, right, down, up) moves the operating finger into a position that is closer to the target algorithmic solution. A penalty of -0.10 is given if the opposite happens.
- A reward of 1 if the signpost correctly moves to the next position when a partial sum is done, or when the agent correctly resets the signpost.
- A reward of 1 for correct move and slide.
- A reward of 1 when the agent chooses the submit action, and the abacus configuration represents the correct partial result.
- If the agent does any of the previous three actions in the wrong way, the episode is terminated and the agent is given a penalty of -1.

Algorithm 1 Addition algorithm. c_r and c_l are the two columns (left and right) visible to the agent in the sliding window. $S(c)$ is a function that returns the symbol encoded in a column.

```
1:  while not done do
2:      Read symbol s from environment
3:      if s is digit then
4:          Write (S(c_r) + s)%5 on c_r
5:          if signpost in c_l then
6:              Move signpost right
7:          else if S(c_r) + s ≥ 5 then
8:              carry ← True
9:          else
10:             carry ← False
11:         while carry = True do
12:             Move operating finger right
13:             Write (S(c_r) + s)%5 on c_r
14:             if S(c_r) + 1 ≥ 5 then
15:                 carry ← True
16:             else
17:                 carry ← False
18:         while signpost not in c_l do
19:             Move operating finger left
20:     else                                        ▷ Reset the abacus
21:         while c_l is not padding do
22:             Move operating finger left
23:         while signpost not in c_r do
24:             Move signpost left
```

We determine the maximum length of an episode dynamically, in order to avoid long loops of meaningless actions. The agent is given 32 timesteps for each correct move and slide action - i.e., the maximal theoretical distance between any two possible move and slide actions in a solution trajectory

[1] We have chosen all hyper-parameters for the feature extractor and the feed-forward networks starting from small architectures and increasing their size until we achieved a satisfactory performance.

[2] We do not report the subtraction algorithm since it only differs in lines 4, 7, 13 and 14 where we implement the actual operation and check if a carry is necessary.

according to the reference algorithm. This mechanism grants the agent enough timesteps to complete the operations that have long trajectories (i.e. resetting the abacus at the end of an operation or computing a long carry), while also allowing the agent to explore the functioning of the virtual abacus during training.

2.5 Model training

We use the Proximal Policy Optimization (PPO) learning algorithm with a linearly-decaying sinusoidal learning rate, clipped surrogate objective funcion [14], frame stacking [15] and masking [16, 17], as implemented in the Python framework Stable Baselines3 [18]. We have also tried to apply the DQN learning algorithm [15] with frame stacking and the same learning rate decay scheme we used with PPO. However, we observed an extremely slow speed of convergence in all reward settings, and especially with the ones having sparse rewards.

Past information is provided by feeding the last 3 environment states stacked into a 3D tensor. We mask illegal actions, e.g. moving left on the left edge of the board, or actions that do not produce any effect on the environment, such as using move and slide when the abacus is already in the target configuration. We implemented an early stopping criterion, whereby learning is interrupted if the KL divergence between the old policy and the new one is greater than a threshold $\alpha = 0.2$ that was empirically chosen.

3 Results

In this section we describe the simulation results and we provide some insights in the way the agent works by analysing its failures when probed to solve problems involving integers longer than the ones seen during training. [3]

3.1 Solving arithmetic tasks

We designed the simulations with the aim of studying the level of supervision that is necessary to successfully learn the arithmetic task. To this aim, we

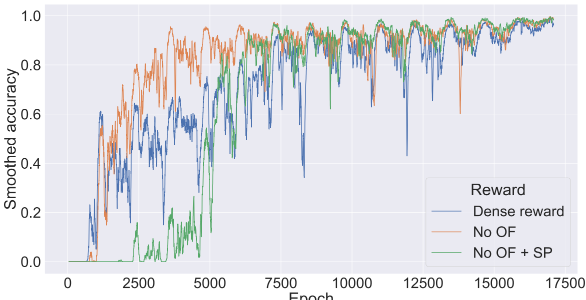

Figure 2: Learning performance of models trained with varying amounts of supervision (oscillations are due to the sinusoidal learning rate). We measure accuracy as the fraction of operations correctly computed in one epoch.

trained the agent with varying amounts of reward: in the simplest case, we included all components of the modular reward function. We then removed the components of the reward related to the movement of the operating finger (OF) and those related to the movement of the signpost (SP). The first component provides very frequent but not strictly necessary supervision, as the agent can discover the correct movement of the operating finger by exploration. The second component provides important information to the agent, in that the correct movement of the signpost is necessary to solve operations that require (possibly very long) carries.

As shown in Fig. 2, the agent is able to solve the task in all cases, reaching an almost perfect accuracy. Surprisingly, reducing the amount of supervision and letting the agent discover the most effective way to use the operating finger leads to a *faster* training and also higher performance in terms of number of consecutive operations successfully computed (see Table 1). Further reducing the level of supervision causes an initial learning slowdown but still allows to reach a very high accuracy later on, although the final performance in terms of number of consecutive operations is lower compared to the intermediate level of supervision.

Next, we have removed the reward for correct move and slide actions, observing that learning becomes so slow that the performance at every epoch is barely improving [4]. Therefore, we find that in or-

[3] We make the code that was used to run the experiments publicly available at this GitHub repository: https://github.com/ChenEmmaL/imitation_abacus.

[4] We did not remove the penalty for long trajectories as it shapes the behavior of the agent only indirectly. We also

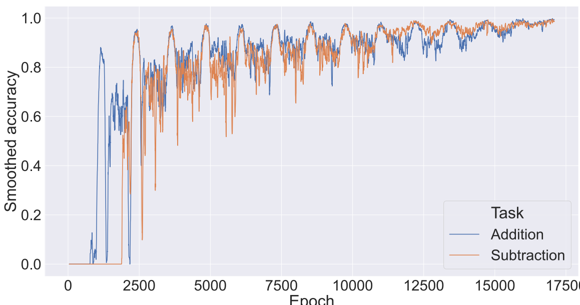

Figure 3: Learning performance of the models trained on sum only or subtraction only with the dense reward function.

der to learn the algorithm in a reasonable time, the agent needs to receive the following essential feedback: the reward for correct move and slides, the reward for correct partial answers given using the 'submit' action, the penalty for long trajectories, and the penalty (including episode termination) in case of wrong move and slides or partial answers.

We also trained the agent on each arithmetical task separately, using the highest level of supervision. As expected, learning a single task is simpler, both in terms of learning speed (Fig. 3) and number of successful consecutive operations (Table 1).

Model	N. op.
Dense reward	515
No OF	1145
No OF + SP	773
Addition	907
Subtraction	675

Table 1: Maximum number of operations successfully completed during training in the different training scenarios.

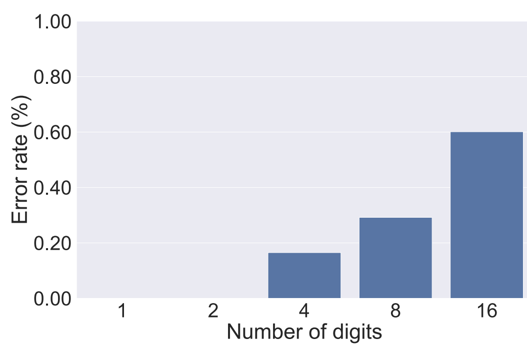

Figure 4: Error rate of the model on operations involving longer operands than those seen during training, which contained at most 6 digits.

3.2 Generalizing to longer operands

Learning to solve arithmetic operations with operands ranging beyond the intervals encountered during training is one of the main challenges for deep learning models [8]. We thus investigated the capability of our best performing agent, namely the one trained on both tasks with an intermediate level of supervision[5] by sampling the two test operands in the intervals $[5^{x-1}, 5^x)$, where $x \in \{1, 2, 4, 8, 16\}$.

For each interval, we sampled 100000 operands and counted the number of mistakes made by the agent. Although the trained agent does not exhibit perfect OOD generalization capabilities and its error rate grows with the number of digits involved in the operations (see Fig. 4), it can still solve sums and differences involving operands in intervals unseen during training with an arguably low error rate (e.g., when operands involved more than two times the amount of digits observed during training, the error rate was still below 1%).

3.3 Analysis of errors

We analyzed the pattern of errors made in the most extreme OOD regime, that is, when operands were sampled in the interval $[5^{15}, 5^{16})$, by collecting statistics over 100000 simulations. The agent commits 601 errors, which is coherent with the error rate reported in Fig. 4. We recorded the errors and manually compiled a list of 4 different error classes: errors in the movement of the signpost, errors during a carry operation, and errors occurring during 'simple' addition or subtraction operations. The latter kind of errors might in fact include operations that require a carry; we only selected errors

did not remove the component of the reward that signals if a partial operation was successfully completed, as it provides the agent with the most important signal about the completion of the task.

[5]Note that in this case we extend the abacus to 20 columns in order to represent the extended range of operands.

Error class	%
Simple operation	32.9
Signpost right	27.7
Carry	27.3
Signpost left	12.1

Table 2: Relative frequency of error types. A simple operation is a sum or subtraction between two digits. The Carry and Signpost right classes occur when the agent must compute a carry operation.

that did not happen when performing the carry operation (e.g. removing one unit from the column to the right and fill the current column) but instead occurred when the carry was completed and the agent needed to compute a sum or subtraction.

As shown in Table 2 the most frequent kind of error is the one involving a simple sum or subtraction operation. The second and third most common classes of errors are wrong carries or movements of the signpost rightwards: notice that such a movement is required precisely when the agent must perform a carry operation. These results reflect the fact that the carry operation is the most complicated step in multi-digit sum or subtraction. Lastly, the least frequent kind of error is the one involving the movement of the signpost to the left, indicating that the agent has learned to reset the abacus to the initial position almost perfectly.

Since our system includes a sliding window mechanism that limits the capacity of the agent to observe the abacus, we investigated whether errors are equally distributed on all columns. The histogram in Fig. 5 representing the frequency of errors by column shows that most errors occur on the second column, which is the first one observed outside the initial position of the sliding window. This is consistent with our previous observation that many errors occur during a carry operation, which requires to move the sliding window to the right. We can also observe a periodic pattern of errors from the third column onward, most likely due to the specific choice used in the positional encoding. This reveals that, although this element of the environment contributes to the capacity of the agent to generalize to unseen ranges of operands, it also introduces a regularity in the errors which depends on the column where an operation must be computed.

Finally, in Table 3 we report a few significant examples of errors made by our system in the OOD generalization regimen. Consistently with the previous analysis, we can see that for both sums and subtractions the agent can make a mistake in the early positions (first columns) as well as in the last ones. Also, it is evident that an error in a carry or regrouping can propagate to the following columns. However, it can also happen that the system is resilient to such mistakes, and thus still compute the rest of the operation correctly: this is a desirable property, since it avoids propagating errors and thus keeps the absolute value of the error relatively low.

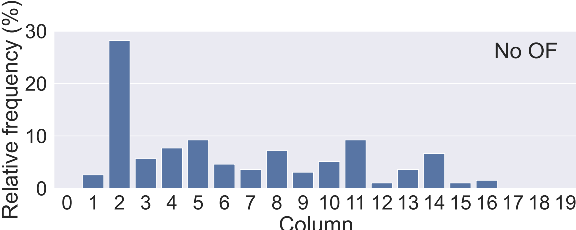

Figure 5: Relative frequency of errors by column of the abacus for the best model. Most errors happen on the second column, and a periodic pattern emerges from the third column onward.

S	2204010440402424	3014100322010344
I	+3422111404102300	+1342122200324413
O	11131122400010024	3014100322340312
T	11131122400010224	4411223022340312
S	4110012024223441	3342443241400324
I	-1300014214322224	-3231303122242113
O	4041442304401212	111040114103211
T	2304442304401212	111140114103211

Table 3: Examples of errors. We report the state of the abacus (**S**), operation input to the system to be executed (**I**), output produced (**O**) and true output (**T**). Operands are in base 5 to facilitate the interpretation of mistakes in carries and regroupings.

4 Discussion

In this work we introduced a framework that can be used to study how an agent can learn to solve arithmetic operations by exploiting deep reinforcement learning and by interacting with external rep-

resentations that incorporate the working principles of an abacus. Our framework is inspired by the way humans interact with external representational tools to solve mathematical problems, connecting to the more general trend of exploring tool use in deep reinforcement learning environments [19, 20, 21]. Differently from similar problems in the deep RL literature, such as learning to play combinatorial games, the problem we propose is characterized by an algorithmic nature, in that the goal of the agent is learning an exact solution algorithm to arithmetic problems, rather than a strategy to win a game.

Our simulations suggest that in order to learn to solve arithmetic problems with model-free reinforcement learning, a memory-less agent needs to receive a certain amount of explicit supervision to overcome its inability to plan in the long-term. Notably, the agent we present is able to solve problems involving operands that are well outside the training range, which can be considered the main challenge of the class of arithmetic problems we propose. The agent is able to do so thanks to specific learning biases: a sliding window, a positional encoding, and the representation format of the operands on the abacus.

It might be possible let the agent fully observe the virtual abacus and learn which part is relevant in any given moment; however, by adopting a relative view on the learning environment through the sliding window the agent can effortlessly generalize the strategy learned on a limited interval of operands to ones that are more than two times longer. At the same time, it turned out that including elements contributing to generalization capability, such as the periodic positional encoding, also introduced unwanted regularities in the errors committed by the system.

An exciting venue for future research would be to explore the possibility to endow the agent with some form of memory (e.g., by exploiting recurrent architectures), which would allow to plan in the distant future and thus learn to solve the problem with even less supervision. Furthermore, it would be interesting to exploit model-based and hierarchical reinforcement learning approaches to endow the agent with native capability to internally simulate the functioning of the external tool and learn to compose simple solution steps into more complex strategies.

References

[1] Brenden M. Lake, Tomer D. Ullman, Joshua B. Tenenbaum, and Samuel J. Gershman. Building machines that learn and think like people. *Behavioral and Brain Sciences*, 40: e253, 2017. doi: 10.1017/S0140525X16001837. 1

[2] Alberto Testolin. The challenge of modeling the acquisition of mathematical concepts. *Frontiers in Human Neuroscience*, 14, 2020. ISSN 1662-5161. doi: 10.3389/fnhum.2020.00100. URL https://www.frontiersin.org/articles/10.3389/fnhum.2020.00100. 1

[3] David Saxton, Edward Grefenstette, Felix Hill, and Pushmeet Kohli. Analysing mathematical reasoning abilities of neural models. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL https://openreview.net/forum?id=H1gR5iR5FX. 1

[4] Aitor Lewkowycz, Anders Johan Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Venkatesh Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, Yuhuai Wu, Behnam Neyshabur, Guy Gur-Ari, and Vedant Misra. Solving quantitative reasoning problems with language models. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=IFXTZERXdM7. 1

[5] Andrew Trask, Felix Hill, Scott E Reed, Jack Rae, Chris Dyer, and Phil Blunsom. Neural arithmetic logic units. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018. URL https://proceedings.neurips.cc/paper/2018/file/0e64a7b00c83e3d22ce6b3acf2c582b6-Paper.pdf. 1

[6] Andreas Madsen and Alexander Rosenberg Johansen. Neural arithmetic units. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL https://openreview.net/forum?id=H1gNOeHKPS. [1]

[7] Lukasz Kaiser and Ilya Sutskever. Neural gpus learn algorithms. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016. URL http://arxiv.org/abs/1511.08228. [1]

[8] Samuel Cognolato and Alberto Testolin. Transformers discover an elementary calculation system exploiting local attention and grid-like problem representation. In *International Joint Conference on Neural Networks*, 2022. doi: 10.1109/IJCNN55064.2022.9892619. [1, 5]

[9] Lei Wang, Dongxiang Zhang, Lianli Gao, Jingkuan Song, Long Guo, and Heng Tao Shen. Mathdqn: Solving arithmetic word problems via deep reinforcement learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1), Apr. 2018. doi: 10.1609/aaai.v32i1.11981. URL https://ojs.aaai.org/index.php/AAAI/article/view/11981. [1]

[10] Danqing Huang, Jing Liu, Chin-Yew Lin, and Jian Yin. Neural math word problem solver with reinforcement learning. In *Proceedings of the 27th International Conference on Computational Linguistics*, pages 213–223, Santa Fe, New Mexico, USA, August 2018. Association for Computational Linguistics. URL https://aclanthology.org/C18-1018. [1]

[11] Cezary Kaliszyk, Josef Urban, Henryk Michalewski, and Miroslav Olšák. Reinforcement learning of theorem proving. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018. URL https://proceedings.neurips.cc/paper/2018/file/55acf8539596d25624059980986aaa78-Paper.pdf. [1]

[12] Petar Velickovic, Rex Ying, Matilde Padovano, Raia Hadsell, and Charles Blundell. Neural execution of graph algorithms. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL https://openreview.net/forum?id=SkgKO0EtvS. [1]

[13] Avi Schwarzschild, Eitan Borgnia, Arjun Gupta, Furong Huang, Uzi Vishkin, Micah Goldblum, and Tom Goldstein. Can you learn an algorithm? generalizing from easy to hard problems with recurrent networks. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 6695–6706. Curran Associates, Inc., 2021. URL https://proceedings.neurips.cc/paper/2021/file/3501672ebc68a5524629080e3ef60aef-Paper.pdf. [1]

[14] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017. URL https://arxiv.org/abs/1707.06347. [4]

[15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013. URL https://arxiv.org/abs/1312.5602. [4]

[16] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, John Quan, Stephen Gaffney, Stig Petersen, Karen Simonyan, Tom Schaul, Hado van Hasselt, David Silver, Timothy Lillicrap, Kevin Calderone, Paul Keet, Anthony Brunasso, David Lawrence, Anders Ekermo, Jacob Repp, and Rodney Tsing. Starcraft ii: A new challenge for reinforcement learning, 2017. URL https://arxiv.org/abs/1708.04782. [4]

[17] Shengyi Huang and Santiago Ontañón. A closer look at invalid action masking in policy gradient algorithms. *The International FLAIRS Conference Proceedings*, 35, May 2022. doi: 10.32473/flairs.v35i.130584. URL https://journals.flvc.org/FLAIRS/article/view/130584. [4]

[18] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268): 1–8, 2021. URL http://jmlr.org/papers/v22/20-1364.html. [4]

[19] Bowen Baker, Ingmar Kanitscheider, Todor M. Markov, Yi Wu, Glenn Powell, Bob McGrew, and Igor Mordatch. Emergent tool use from multi-agent autocurricula. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL https://openreview.net/forum?id=SkxpxJBKwS. [7]

[20] Sascha Fleer and Helge Ritter. Solving a tool-based interaction task using deep reinforcement learning with visual attention. In Alfredo Vellido, Karina Gibert, Cecilio Angulo, and José David Martín Guerrero, editors, *Advances in Self-Organizing Maps, Learning Vector Quantization, Clustering and Data Visualization*, pages 231–240, Cham, 2020. Springer International Publishing. ISBN 978-3-030-19642-4. doi: 10.1007/978-3-030-19642-4_23. [7]

[21] Silvester Sabathiel, Flavio Petruzzellis, Alberto Testolin, and Trygve Solstad. Self-communicating deep reinforcement learning agents develop external number representations. In *Proceedings of the Northern Lights Deep Learning Workshop*, volume 3, 2022. doi: 10.7557/18.6291. [7]