Production-based Cognitive Models as a Test Suite for Reinforcement Learning Algorithms

Adrian Brasoveanu
UC Santa Cruz, Linguistics
1156 High St., Santa Cruz, CA 95064
abrsvn@gmail.com

Jakub Dotlačil
Utrecht University
Utrecht, The Netherlands
j.dotlacil@gmail.com

Abstract

We introduce a framework in which production-rule based computational cognitive modeling and Reinforcement Learning can systematically interact and inform each other. We focus on linguistic applications because the sophisticated rule-based cognitive models needed to capture linguistic behavioral data promise to provide a stringent test suite for RL algorithms, connecting RL algorithms to both accuracy and reaction-time experimental data. Thus, we open a path towards assembling an experimentally rigorous and cognitively realistic benchmark for RL algorithms. We extend our previous work on lexical decision tasks and tabular RL algorithms (Brasoveanu and Dotlačil, 2020b) with a discussion of neural-network based approaches, and a discussion of how parsing can be formalized as an RL problem.

1 Reinforcement Learning and Production-based Cognitive Models

We introduce a framework in which we can start exploring how Reinforcement Learning (RL; Sutton and Barto 2018) algorithms scale up against human cognitive performance, as captured by complex, production-based cognitive models. Our ultimate goal is to focus on sophisticated cognitive models of linguistic skills, e.g., the parsers in Lewis and Vasishth (2005); Hale (2011); Engelmann (2016), because cognitive models that use theoretically-grounded linguistic representations and processes call for richly structured representations and complex rule systems that pose significant challenges for RL algorithms.

These cognitive models, which capture human-participant accuracy and latency data obtained from forced-choice and reaction-time experiments, can provide exacting, experimentally established benchmarks for the performance of artificial RL agents. These benchmarks will enable us to see if and when different RL algorithms fail, and how exactly they fail. In this paper, we report a small pilot study that exemplifies three modes of failure. Neural-network based function-approximation approaches sometimes (i) fail to learn even fairly simple rule systems in a stable manner. Even when they seem to learn, (ii) they fail by learning a lot of noise (incorrect rules), particularly in more complex tasks. Tabular approaches fare better, but (iii) learn complex tasks much more slowly, and still learn a lot of noise in more complex tasks, albeit less so than neural-network approaches.

Bridging the RL–cognitive modeling divide also promises to shed new light on the issue of cognitive model learnability. The learnability problem for production-rule based models can be divided into two parts: (i) rule acquisition – forming complex rules out of simpler ones, and (ii) rule ordering – deciding which rule to fire when. We focus here on the easier problem of rule ordering, and show how, on one hand, linguistic cognitive models provide a benchmark for RL algorithms and, on the other hand, RL provides a framework to systematically investigate cognitive model learnability in a formally and computationally explicit way.

We investigate the issue of rule-ordering learning using the Adaptive Control of Thought-Rational (ACT-R) cognitive architecture (see Anderson and Lebiere 1998; Anderson 2007). The advantage of using ACT-R is that this cognitive architecture and RL have very close, albeit largely unexplored, connections (Fu and Anderson 2006, Sutton and Barto 2018, Ch. 14). ACT-R tries to address the rule acquisition and ordering problems, but its proposed solutions – production compilation and rule-utility estimation, respectively – have not been systematically applied to complex models for linguistic skills (apart from Taatgen and Anderson 2002, which investigates the role of production compilation in

Proceedings of the Workshop on Cognitive Modeling and Computational Linguistics, pages 28–37
Online Event, November 19, 2020. ©2020 Association for Computational Linguistics
https://doi.org/10.18653/v1/P17

morphology acquisition).

After a brief overview of ACT-R and the description of a linguistic task for which rule-ordering learning will be studied (Section 2), we show how the linguistic task can be analyzed as an RL problem (Section 3), and discuss the results of our experiments with tabular and neural-network based Q-learning algorithms (Section 4). We then briefly discuss how parsing can be formalized as an RL problem (Section 5), and conclude with a summary and some directions for future work (Section 6).

2 Learning Goal-conditioned Rules in Lexical Decision: A Simple Test Case

There are two types of memory in ACT-R. On one hand, we have declarative memory ('knowing that'), which encodes our knowledge of facts. Facts are represented as chunks / attribute-value matrices, e.g., the lexical chunk for the word *elephant*:

(1)

ISA:	word
FORM:	elephant
MEANING:	⟦elephant⟧
CATEGORY:	noun
NUMBER:	sg

On the other hand, we have procedural memory ('knowing how'), which consists of the set of productions that fire in series to generate cognitive behavior / processes. These productions have the form of rewrite rules in formal grammars (e.g., context free / phrase structure grammars), but in ACT-R, they are conditionalized cognitive actions: the ACT-R mind fires a production, i.e., takes the action encoded in it, if the current cognitive state satisfies the preconditions of that production. Procedural memory and its production rules are the focus of our investigation and RL experiments here.

An example production is provided in (2): if the current cognitive state is such that the goal buffer (which drives cognitive processes in ACT-R) encodes a TASK of 'retrieving' the lexical entry for the FORM 'elephant,' then ($\Longrightarrow$), we take the action of placing a Retrieval (buffer) request to search declarative memory for a word with the FORM 'elephant,' and we consequently update the TASK in the goal buffer to one of 'retrieval done.'

(2) Goal> | TASK: retrieving | FORM: elephant | $\Longrightarrow$

Goal> | TASK: retrieval done |

Retrieval> | ISA: word | FORM: elephant |

Implicit in this example production is that an ACT-R mind is composed of modules, which include declarative and procedural memory, but also visual and motor modules etc. Modules are not directly accessible: they can only be accessed through their associated buffers, e.g., the retrieval buffer is associated with declarative memory. Buffers serve a dual purpose: individually, they provide the input/output interface to specific modules; as a whole, however, buffers represent the current cognitive state of the mind. Crucially, productions fire based on the current cognitive state, i.e., they are conditioned on the contents of various buffers.

The ACT-R architecture constrains cognitive behavior in various ways, two of which are that (i) buffers can hold only one chunk, and (ii) only one production can fire at any given time.

The framework and the range of issues that emerge when we try to systematically bridge RL and ACT-R are best showcased with a simple kind of linguistic tasks: lexical decision (LD) tasks. We briefly outline in Section 5 how to extend this approach to parsing models implemented in ACT-R. In an LD task, human participants see a string of letters on a screen. If the participants think the string of letters is a word, they press one key (`J` in our setup). If they think the string is not a word, they press a different key (`F` in our setup). After pressing the key, the next stimulus is presented. We will investigate the extent to which two kinds of RL agents can be used to learn goal-conditioned rules in an ACT-R based cognitive model of LD tasks.

The main point of proposing and examining an ACT-R model of LD tasks is to construct a simple example of a production-rule based model that enables us to study learnability issues associated with RL algorithms. Our discussion recapitulates the main results in Brasoveanu and Dotlačil (2020b), and extends them with an initial foray into neural-network based RL approaches. The LD model and RL algorithms can be scaled up in future work to more complex and cognitively realistic syntactic and semantic parsing models, since LD is basically a subcomponent of parsing.

The LD model provides the basic scaffolding of production rules needed for LD tasks, which is all that we need for our purposes: fleshing it out

to capture major experimental results about LD, or comparing it to previously proposed cognitive models of LD is not our focus here.

We model three LD tasks of increasing length, hence difficulty: (i) a 1-stimulus task consisting only of the word *elephant*, (ii) a 2-stimuli task consisting of the word *elephant* and a non-word, and (iii) a 4-stimuli task consisting of the word *elephant*, a non-word, the word *dog*, and another non-word.

The model components are split between declarative memory, which stores the lexical knowledge of an English speaker, and procedural memory, which stores rules that enable the model to carry out the LD task. LD tasks can be modeled in ACT-R with a small number of rules (see Brasoveanu and Dotlačil 2019, 2020a). We will assume 4 rules, provided in standard ACT-R format below. These rules were originally hand-coded to fire serially by conditioning all the actions on specific goal states. The goal conditions are stricken out, indicating that goal states were not provided to the RL agents: the order of the rules was not hand-coded for them. Instead, we want the RL agents to learn the rule ordering.

Rule 1: Retrieving

~~goal>~~ | ~~STATE: retrieving~~ |

visual> | VALUE: =val
VALUE: ∼FINISHED |

⟹

goal> | STATE: `retrieval_done` |

+retrieval> | ISA: word
FORM: =val |

Rule 2: Lexeme Retrieved

~~goal>~~ | ~~STATE: retrieval_done~~ |

retrieval> | BUFFER: full
STATE: free |

⟹

goal> | STATE: `retrieving` |

+manual> | CMD: press-key
KEY: J |

Rule 3: No Lexeme Found

~~goal>~~ | ~~STATE: retrieval_done~~ |

retrieval> | BUFFER: empty
STATE: error |

⟹

goal> | STATE: `retrieving` |

+manual> | CMD: press-key
KEY: F |

Rule 4: Finished

~~goal>~~ | ~~STATE: retrieving~~ |

visual> | VALUE: FINISHED |

⟹

goal> | STATE: `done` |

With fully specified, hand-coded rules, the LD task unfolds as follows. Assume the initial goal STATE of the ACT-R model is `retrieving`, and the word *elephant* appears on the virtual screen of the model, which is automatically stored in the VALUE slot of the visual buffer. At this initial stage, the preconditions of **Rule 1** are satisfied, so the rule fires. This starts an attempt to retrieve a word with the form *elephant* from declarative memory, and the goal STATE is updated to `retrieval_done`. When the word is successfully retrieved, **Rule 2** fires and the `J` key is pressed. At that point:

i. in the 1-stimulus task, the text FINISHED is displayed, then **Rule 4** fires and ends the task;

ii. in the 2-stimuli task, the non-word is displayed, then **Rule 1** fires again; the retrieval attempt fails since we cannot retrieve a non-word from declarative memory, so **Rule 3** fires and the `F` key is pressed; at that point, the text FINISHED is displayed, then **Rule 4** fires and ends the task;

iii. in the 4-stimuli task, the first non-word is displayed, **Rule 1** fires again, then, just as in the 2-stimuli task, **Rule 3** fires and the `F` key is pressed, after which the word *dog* is displayed, **Rule 1** fires for the third time followed by **Rule 2**, which means that the `J` key is pressed and the second non-word is displayed; then, **Rule 1** fires for the final time, followed by **Rule 3**, which triggers an `F`-key press, after which the text FINISHED is displayed, so **Rule 4** fires and ends the task.

Thus, the rule sequences for the 3 LD tasks are as shown in (3), assuming fully specified, hand-coded rules. However, as we mentioned, we do not hand-code the goal-state preconditions, indicated by striking out the goal states in the 4 rules above. We only specify the actions (and preconditions associated with buffers other than the goal buffer) and let the RL agents, *which can select any rule at any given time*, learn to carry out the LD tasks.

(3) 1-stim rules: [**1** – **2**] – **4**
2-stim rules: [**1** – **2**] – [**1** – **3**] – **4**
4-stim rules: [**1** – **2**] – [**1** – **3**] – [**1** – **2**] – [**1** – **3**] – **4**

We see that *proper rule ordering / sequencing* is crucial to successfully completing an LD task, which is like searching for a path through a maze:

(i) the position in the maze is the current cognitive state of the ACT-R mind, (ii) the possible moves (up, left etc.) are the production rules we can fire, and (iii) a path through the maze is given by the proper sequence of production rules we need to fire to complete the LD task.

3 Rule Ordering as an RL Problem

Markov Decision Processes (MDPs) are the stochastic models of sequential decision-making that form the basis of RL approaches to learning. In an MDP, an agent interacts with its environment and needs to make decisions at discrete time steps $t = 1, 2, \ldots, n$. Defining what counts as the agent and what counts as its environment is part of the modeling process. At every step t, all the information from the past relevant for the current action selection is captured in the current state of the process s_t. This is the Markov property: the future is independent of the past given the current state.

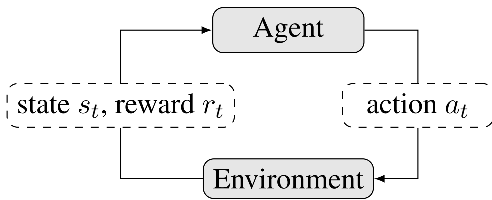

Figure 1: Agent-environment interaction in an MDP

As Figure 1 shows, the environment passes to the agent a state s_t and, at the same time, a reward signal r_t. The agent observes the current state s_t and reward r_t and takes an action a_t, which is passed from the agent to the environment. The cycle then continues: at time step $t + 1$, the environment responds to the agent's action with a new state s_{t+1} and a new reward signal r_{t+1}. Based on these, the agent selects a new action a_{t+1} etc. The definitions of 'state' and 'action' depend on the problem, and are part of the modeling process, just like defining what counts as the agent and its environment.

The agent's *policy* is a complete specification of what action to take at any time step. Given the Markovian nature of the MPD, the policy π is effectively a mapping from the state space S to the action space A, $\pi : S \to A$. A deterministic policy is a mapping from any given state s_t to an action $a_t = \pi(s_t)$, while a stochastic policy is a mapping from any given state s_t to a probability distribution over actions $a_t \sim \pi(s_t)$.

The agent's goal is to maximize some form of cumulative reward over an *episode*, which is a complete, usually multi-step interaction between the agent and its environment. In our case, an episode would be a full simulation of a 1/2/4-stim LD task.

The agent learns (solves/optimizes the MDP) by updating its policy π to maximize the (per-episode) cumulative reward. The standard cumulative reward for an episodic task is the discounted return G: at time step $t < n$ (n is the final step in the episode), $G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{n-t-1} r_n$, i.e., G_t is the sum of the current reward and the discounted future rewards until the final step n of the episode. Future rewards are discounted in finite / episodic tasks because the agent has a preference for more immediate rewards. The present value of future rewards is determined by the discount factor γ ($0 \leq \gamma \leq 1$). We define the (state-)action value function $Q_\pi(s, a)$ to be the expected (discounted) return when starting in state s, performing action a and then following the policy π until the end of the episode.

The agent selects actions with the goal of maximizing its expected discounted return. If we estimate the Q function for a given policy based on the interactions between the agent and its environment, i.e., based on experience, we can improve that policy by 'greedification:' given a state s, we can always select the optimal action in s, i.e., the action with the maximal expected return according to our current Q estimate.

Q-learning algorithms, which are the focus of our investigation here (given their widespread use), come in various flavors. The simplest one is tabular Q-learning (Watkins, 1989; Watkins and Dayan, 1992), which is fairly effective for our LD tasks. We will also investigate approaches that approximate the Q-function with neural networks, specifically Deep Q-networks (DQN, Mnih and al 2015).

In tabular Q-learning, the Q function $S \times A \to \mathbb{R}$ is represented as a look-up table that stores the estimated values of all possible state-action pairs. The Q table is initialized to an arbitrary fixed value (0). The agent then updates the Q table incrementally at each time step t: the value of the pair (s_t, a_t), where s_t is the state relative to which the agent took action a_t, is updated based on the reward signal r_{t+1} and the new state s_{t+1} that the agent receives back from the environment after taking action a_t.

Q-learning is a form of temporal difference (TD) learning, as shown in (4). The Q^{new} value estimate for the state-action pair (s_t, a_t) is based on the Q^{old} value, updated by some proportion α (the learning

rate; $0 < \alpha \leq 1$) of the TD error.

(4) $$Q^{new}(s_t, a_t) \leftarrow Q^{old}(s_t, a_t) + \alpha \cdot \overbrace{\Big(\underbrace{r_{t+1} + \gamma \cdot \underbrace{\max_{a_{t+1}} Q^{old}(s_{t+1}, a_{t+1})}_{\text{next-state value estimate}}}_{\text{TD target (updated value)}} - Q^{old}(s_t, a_t) \Big)}^{\text{TD (temporal difference) error}}$$

The TD error is the difference between the TD target – which is an updated estimate of the value of the (s_t, a_t) pair – and the Q^{old} value estimate. The TD target consists of (i) the reward r_{t+1} the agent receives after action a_t, which is part of the new data the agent gets back from the environment after action a_t, plus (ii) the estimate of the value of the next state s_{t+1}, where the next state s_{t+1} is the other part of the new data the agent gets back from the environment after action a_t. The Q-learning optimal estimate for the value of the next state s_{t+1} is discounted by γ, since this state is in the future relative to the state-action pair (s_t, a_t) we're currently updating. This optimal estimate for s_{t+1} is aggressively confident / optimistic (in contrast to Expected Sarsa, for example; see van Seijen et al. 2009): the agent looks at all the possible actions a_{t+1} that can be taken in state s_{t+1} and assumes that the action a_{t+1} with the highest Q^{old}-value provides an accurate estimate of the s_{t+1} value.

For tabular Q-learning, the agent (in the RL sense) is a Q-value table that assigns values to all possible state-action pairs and that guides the rule selection process at every cognitive step. The environment is the cognitive state of the ACT-R model / mind, which could conceivably consist of (i) all the modules (procedural memory, declarative memory and visual and motor modules) together with (ii) their associated buffers (goal, retrieval, visual-what, visual-where and the manual buffer). This, however, would lead to a very large state space S, which in turn would lead to a large Q-value table. DQN and similar neural-network approaches can help with the large state-space problem, but we will nonetheless take a state s to consist just of: (i) the goal buffer, (ii) the retrieval buffer, (iii) the value in the visual-what buffer, if any, and finally, (iv) the state of the manual buffer (busy or free). For example, the state after the word *elephant* is retrieved from declarative memory is: goal: {STATE: retrieval_done}, retrieval: {FORM: elephant}, visual_value: elephant, manual: free.

The action space consists of the 4 rules above, namely `retrieving`, `lexeme retrieved`, `no lexeme found` and `finished`, together with a special action `None` that the agent selects when it wants to not fire any rule because it prefers to wait for a new cognitive state.

The reward structure is as follows: (i) the agent receives a positive reward of 1 at the end of an episode (when the LD task is completed), specifically, when the goal STATE is `done`; (ii) the agent receives a negative reward of -0.15 for every rule it selects, other than `None`; (iii) there is no penalty for waiting and selecting no rule, i.e., for selecting the special action `None`, which is optimal when waiting for retrieval requests from declarative memory to complete, for example; (iv) finally, at every step, the agent receives a negative reward equal to the amount of time that has elapsed between the immediately preceding step and the current step (multiplied by -1 to make it negative).

This reward structure is designed to encourage the agent to finish the task as soon as possible by selecting the smallest number of rules. The negative temporal reward (iv) discourages the agent from just repeatedly selecting an action, e.g., `None`. This ends up timing out the LD task in a small number of steps and fast-forwards the agent to the maximum waiting time per stimulus the ACT-R environment allows for, which we set to 2 seconds per word for the LD task.

Thus, given a simple reward structure that incorporates fairly minimal cognitive assumptions, RL enables us to induce proper rule sequences to complete the LD tasks: RL enables us to *leverage the simple assumptions built into the reward structure to solve the much harder rule-ordering problem by direct experience / trial-and-error interaction* with the LD tasks.

4 Experiments and Results

We assume the usual ACT-R defaults, e.g., rule firing time is set to 50 ms. The discount factor γ is set to 0.95 and the learning rate α is set to 10^{-3}. We use an ϵ-greedy policy to balance exploration and exploitation, with ϵ annealed from a maximum of 1 to a minimum of 0.01.

We investigate two types of agents / algorithms: (i) tabular Q-learning (the main results for tabular agents are from Brasoveanu and Dotlačil 2020b), and (ii) DQN. To a large extent, the agents learn by trial and error to successfully carry out the LD tasks: they learn how to properly order the rules

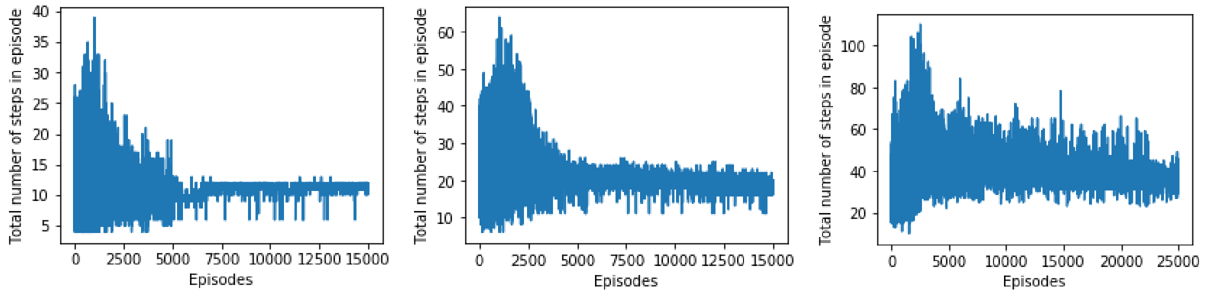

Figure 2: Tabular Q: Steps per episode for the 1-stim (left), 2-stim (middle) and 4-stim (right) tasks

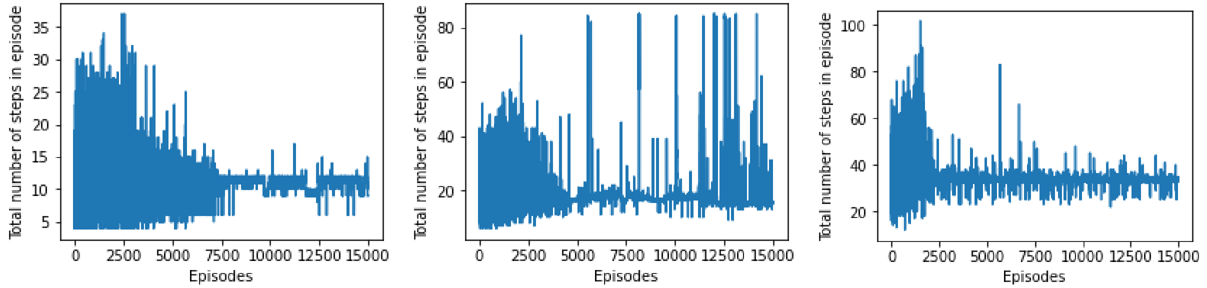

Figure 3: DQN: Steps per episode for 1-stim (left), 2-stim (middle) and 4-stim (right) tasks

and complete the LD tasks as efficiently as possible. This is no small feat given that the actual number of steps, i.e., decision points, when the agents needs to select an action, is larger than the high-level sequences of rule firings in (3) above. For example, for a 1-stim task, there are actually 12 steps where the agent needs to decide whether to wait or to fire a specific rule (when the agent does not complete the task perfectly, it might take much more than 12 steps). The 2-stim task requires 18 such steps (if perfectly completed), and the 4-stim task requires 34 steps (again, if perfectly completed).

The reason for the higher number of steps compared to the number of rules is that our LD simulations involve the visual and motor modules (to read strings of characters and to press keys) in addition to the declarative memory module. Visual and motor actions, just as retrievals from declarative memory, take time, and the agent needs to make decisions while waiting for them to complete.

4.1 Tabular Q-learning in LD Tasks

The higher the number of steps, i.e., the higher number of decision points for the tabular agent, the harder the task is to learn for the tabular agents. As the plots in Figure 2 show, repeated from Brasoveanu and Dotlačil (2020b), learning is faster and less noisy for shorter tasks (fewer stimuli), but the tabular Q-learning agent manages to learn even the most complex 4-stimuli task moderately well.

We simulate $15{,}000$ episodes, i.e., $15{,}000$ LD decision tasks consisting of 1 stimulus only (the word *elephant*), from which the tabular Q agent learns – shown in the leftmost plot in Figure 2. After about $5{,}000$ episodes, the task is completed in ≈ 12 steps, which is the length of the task when completed perfectly. For some episodes, the number of steps is smaller than 12. In these cases, the agent times out the task (e.g., by selecting the `None` action several times) and receives steep negative temporal rewards leading to low returns.

A close examination of the agent's final Q-value table, which stores the agent's rule-firing preferences for any given state, indicates that the agent has learned goal-conditioned rules perfectly. We only look at states for which at least one action/rule has a non-0 value (recall that all Q-values are initialized to 0). For each such state, we identify the action/rule with the highest value. There are 8 states total with at least one non-0 value action, and the maximum-value action for each of these states makes complete sense. For example, `None` is the maximum-value rule at the beginning of every episode when the agent waits for some text to be automatically detected and stored in the visual buffer. Similarly, after a retrieval request is placed, the agent waits for the process to complete.

As the middle plot in Figure 2 shows, we also

simulate 15,000 2-stim episodes (LD tasks consisting of the word *elephant* and the non-word *not_a_word*). After about 9,000 episodes, the task is completed in $\approx$ 18 steps, which is the length of this task when the agent completes it perfectly. A close examination of the agent's final Q-value table indicates that the agent has learned goal-conditioned rules almost perfectly. Once again, we only look at states for which at least one action has a non-0 value – a total of 13 states. For each state, we identify the maximum-value action, and for 12 states, this action makes complete sense. However, unlike in the 1-stim task, there is one state-action pair that encodes a questionable rule. We see here how, for more complex tasks, the tabular RL agents learn spurious rules, which are a by-product of the noisy trial-and-error learning process.

This lack of robust learning, which can be characterized as overgeneralization, or as vulnerability to 'adversarial' inputs, becomes even more prominent in the 4-stim task, where the tabular Q-learning agent learns even more spurious rules. As the rightmost plot in Figure 2 shows, we simulate 25,000 4-stim episodes (LD tasks consisting of the word *elephant*, a non-word, the word *dog* and another non-word). We need more episodes for this task because it is longer, hence more complex, than the 1/2-stim tasks. It takes about 22,000 episodes for the task to be reliably completed in less than 40 steps. The task takes 34 steps when the agent completes it perfectly, but even after 25,000 episodes, the agent takes more steps than that because it tries incorrect rules or waits for no reason. An examination of the final Q-value table indicates that the agent has learned goal-conditioned rules fairly well, but there is also a good amount of spurious rules. There are 24 states total with at least one action with a non-0 value. Out of these, 6 states have questionable / nonsensical maximum-value actions.

4.2 DQN in LD Tasks

The DQN agents use an artificial neural network (ANN) to approximate the Q-function. We use a simple multilayer perceptron with a hidden layer of size 64. A small hyperparameter search indicated that a hidden size of 32 seems to be too small, while 128 or 256, for example, seems to be too large.

The ANNs are trained using 1-step semi-gradient TD (a.k.a. semi-gradient TD(0); Sutton and Barto 2018, Chapters 9-11), with the Adam optimizer (Kingma and Ba, 2015) and a mean squared TD error loss function (see (4) above for the TD error).

As the leftmost plot in Figure 3 shows, the DQN agent takes longer than the tabular agent to learn the 1-stim task, but it completes it more or less perfectly after about 7,500 episodes. We inspect the Q-function approximation encoded by the ANN at the end of the simulation by identifying the maximum-value rule for each of the 36 possible states. Unlike tabular approaches, function-approximation approaches aggressively generalize over states by design, which is why they are appropriate for large state (and action) spaces. The final Q-function approximation aggressively generalizes by taking the `finished` rule to be the maximum value action for 31 out of 36 states. This makes sense given that the `finished` rule is immediately followed by the final positive reward of 1.

The other rules are triggered largely only when they are appropriate. For example, the `lexeme retrieved` rule is triggered only in one state – immediately after the word 'elephant' is successfully retrieved from declarative memory. The `None` rule is only triggered in two states: when the agent is waiting for the visual module to auto-detect and encode the text on the virtual screen, and when waiting for the retrieval request to declarative memory to complete. But the DQN agent overgeneralizes the `retrieving` rule. It is appropriately triggered after the text on the virtual screen is stored in the visual buffer, i.e., when visual value is the word 'elephant,' but it is also triggered in one other state when the `None` rule is appropriate because the agent is waiting for the visual module to auto-detect the text on the virtual screen.

As the middle plot in Figure 3 shows, the DQN agent fails to learn the 2-stim task in a stable manner. We tried several different random seeds, and the DQN agent exhibits unstable learning in most of them, sometimes to an even larger extent than depicted here. An examination of the final Q-function approximation reveals that, once again, the `finished` rule is the maximum value action for the vast majority of states (37 out of 48). The `None` rule is triggered only in 4 states. In two of them, the agent is waiting for the visual module to auto-detect the text on the virtual screen and encode it in the visual buffer (whether the manual buffer is free or busy). In another one, the agent is waiting for the retrieval request associated with the non-word to complete. However, the DQN agent has not learned that `None` should also be triggered

when waiting for the retrieval request associated with 'elephant,' and it incorrectly triggers None in a state where retrieving is more appropriate.

The lexeme retrieved rule is triggered in 3 states. One of them is the expected one: immediately after the word 'elephant' is successfully retrieved from declarative memory. Another one is a reasonable overgeneralization to a state that is exactly the same as the first one except that the visual value is the non-word. In the third state, however, the None rule is more appropriate since the retrieval process for the word 'elephant' is still in progress. The agent has clearly not learned when to trigger the no lexeme found rule, which is triggered in only one state for which the retrieving rule is appropriate (since the word 'elephant' has just been read off the virtual screen). Finally, the retrieving rule is triggered in 3 states, one of which is appropriate as it immediately follows the point at which the non-word has been read off the virtual screen. However, the DQN agent also triggers this rule in two other states, for which it does not make much sense.

As the rightmost plot in Figure 3 shows, the DQN agent seems to performs much better than the tabular Q agent on the 4-stim task, learning to complete it efficiently after about 2,000 episodes. But an examination of the final Q-function approximation reveals an unexpected result: the retrieving rule is aggressively overgeneralized to 82 states (out of 108). The finished rule is the maximum value action for 9 states only, the lexeme retrieved rule for 8 states, the None rule for 7 states, and the no lexeme found rule for 2 states.

The finished rule is mostly triggered in states in which the visual value is FINISHED (5 out of 9 states), but it is also incorrectly triggered in states in which the 4 stimuli are stored in the visual buffer. It is not clear at all that the agent has learned this rule. The lexeme retrieved rule exhibits a similar profile. It is correctly triggered when the retrieval process for the two words are completed successfully, but there is a lot of noise also: 6 out of 8 states are not states in which this rule should be clearly triggered, and in two of them, the retrieval buffer is empty. Thus, it is far from clear that the agent has learned the lexeme retrieved rule.

The None rule is appropriately triggered when the agent is waiting for the visual module to autodetect text on the virtual screen, and when waiting for retrieval requests to complete for the two words 'elephant' and 'dog.' However, the agent has not learned to trigger this rule when waiting for retrieval requests associated with the two non-words. In addition, this rule is overgeneralized to several states where the retrieving rule is more appropriate. Finally, the DQN agent has clearly learned the no lexeme found rule: it is triggered in only two states, after failed retrieval requests associated with the two non-words.

In conclusion, we see that the DQN agent fails to learn the 2-stim task in a stable manner, learns the 1-stim task more slowly than the tabular Q agent, but exhibits an interesting behavior on the 4-stim task. This task seems to be learned very quickly (compared to tabular Q), but there is a very significant amount of noise in the final Q-function approximation. It is therefore not clear that the appropriate preconditions for most of the rules have actually been learned.

5 Parsing as an RL Problem

In this section, we briefly discuss how parsing can be formalized as an RL problem. Just as the LD task, the parsing task can be implemented in ACT-R (cf. Lewis and Vasishth 2005; Brasoveanu and Dotlačil 2018, 2020a). The parser components are split over various ACT-R modules and buffers: (i) lexical knowledge is encoded in declarative memory, (ii) knowledge of grammar and parsing actions are encoded in procedural memory, (iii) expectations about upcoming syntactic categories are encoded in the goal buffer, (iv) information about the current partially-built syntactic parse is encoded in the imaginal buffer (a secondary goal-like buffer), and finally, (v) visual information from the environment is transferred via the visual buffer.

We consider a simple example, which features an eager left-corner parser (Resnik, 1992). Assume we have a simple grammar with four phrase structure rules: (i) S $\rightarrow$ NP VP, (ii) NP $\rightarrow$ Det N, (iii) VP $\rightarrow$ V, (iv) VP $\rightarrow$ V NP. Also, assume that we are reading the sentence *A boy sleeps* word by word. As shown in Figure 4, we start with the empty visual buffer and our goal stack (the stack of the expected syntactic categories) consists of just S: our goal is to parse a sentence.

We then shift focus to the first word, the information is transferred to the goal buffer, at which point we retrieve its syntactic category Det(erminer) from declarative memory. We can now take a series

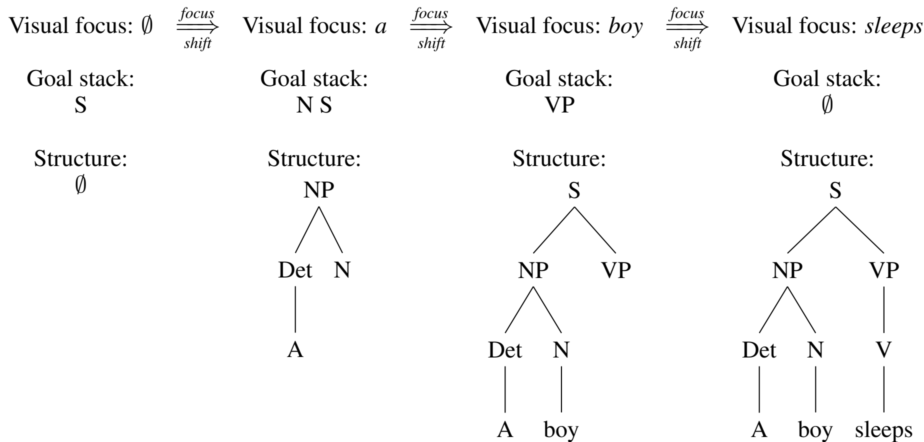

Figure 4: Partial trees built incrementally when reading the sentence *A boy sleeps* word by word

of cognitive steps – that is, we fire a series of productions – that lead to a new state. The new goal stack is N S: we now have the subgoal of finding a N(oun) on the way to S. Also, we build a partial syntactic structure of the form shown in the leftmost tree in Figure 4, and store it in the imaginal buffer. The noun *boy* is then brought into focus, its syntactic category N is retrieved, and we discharge the N goal at the top of the goal stack. At this point, we have the full corner of the rule S $\rightarrow$ NP VP, so we trigger it, which eagerly discharges the S goal and replaces it with the goal of finding a VP (verb phrase). At the same time, a richer partial tree, shown in the middle of Figure 4, is stored in the imaginal buffer. Finally, the verb *sleeps* is in focus, its syntactic category V is retrieved from declarative memory, and we trigger the rule VP $\rightarrow$ V that discharges the VP goal, resulting in an empty goal stack and the rightmost tree structure in Figure 4.

We see that rule ordering plays two roles in parsing. First, the parser has to correctly sequence actions per word: it has to collect the visual information, move it to the goal buffer, recall lexical information from the declarative memory, carry out a parsing action and move its visual attention to the following word. This sequencing of actions is akin to the one explored in the LD task. In addition, the parser has to find the right path through the sequence of parsing rules, e.g., it has to realize that the Det element at the start of the sentence should trigger the NP $\rightarrow$ Det N rule, followed by discharging the N goal etc. Incorrect sequencing would eventually lead to a dead end. For example, had the parser triggered the VP $\rightarrow$ V NP rule when parsing *sleeps*, it would incorrectly end up with an expectation for a non-existent direct object.

6 Summary and Future Work

We argued that sophisticated production-based cognitive models used to capture human behavioral data (particularly linguistic behavior) promise to provide a stringent test suite for RL algorithms. An immediate follow-up would be to explore how RL algorithms perform on a variety of production-based cognitive models, whether linguistic, e.g., syntactic or semantic parsing, or non-linguistic. We have conducted pilot experiments with simple parsing models and tasks, and they are much more difficult than the LD tasks explored in this paper.

Another direction for future research is investigating other value-based tabular learning algorithms (Sarsa, Expected Sarsa), as well as extensively studying ANN-based function-approximation approaches to reinforcement learning, both value and policy based.

Similarly, we might want to investigate curriculum learning (see Elman 1993; Rusu and al 2016 among others) for increasingly complex tasks. A DQN agent that has already learned the 1-stim task might be able to learn the 2/4-stim tasks quickly and well. Curriculum or transfer learning might also enable agents to learn from much fewer interactions, and/or from explicit instructions.

Acknowledgments

We are grateful to three anonymous CMCL 2020 reviewers for their feedback on an earlier version of this paper, the NVIDIA Corporation for a grant of two Titan V GPUs used for this research, and the UCSC OR and THI for a matching grant for additional hardware. The usual disclaimers apply.

References

John R. Anderson. 2007. *How can the human mind occur in the physical universe?* Oxford University Press.

John R. Anderson and Christian Lebiere. 1998. *The Atomic Components of Thought*. Lawrence Erlbaum Associates, Hillsdale, NJ.

Adrian Brasoveanu and Jakub Dotlačil. 2018. An extensible framework for mechanistic processing models: From representational linguistic theories to quantitative model comparison. In *Proceedings of the 2018 International Conference on Cognitive Modelling*.

Adrian Brasoveanu and Jakub Dotlačil. 2019. Quantitative comparison for generative theories. In *Proceedings of the 2018 Berkeley Linguistic Society 44*.

Adrian Brasoveanu and Jakub Dotlačil. 2020a. *Computational Cognitive Modeling and Linguistic Theory*. Language, Cognition, and Mind (LCAM) Series. Springer (Open Access).

Adrian Brasoveanu and Jakub Dotlačil. 2020b. Reinforcement learning for production-based cognitive models. In *Proceedings of the 2020 International Conference on Cognitive Modelling*.

Jeffrey L. Elman. 1993. Learning and development in neural networks: The importance of starting small. *Cognition*, 48:71–99.

Felix Engelmann. 2016. *Toward an integrated model of sentence processing in reading*. Ph.D. thesis, University of Potsdam, Potsdam.

Wai-Tat Fu and John R. Anderson. 2006. From recurrent choice to skill learning: A reinforcement-learning model. *Journal of Experimental Psychology: General*, 135(2):184–206.

John Hale. 2011. What a rational parser would do. *Cognitive Science*, 35:399–443.

Diederik P. Kingma and Jimmy Lei Ba. 2015. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

Richard Lewis and Shravan Vasishth. 2005. An activation-based model of sentence processing as skilled memory retrieval. *Cognitive Science*, 29:1–45.

Volodymyr Mnih and al. 2015. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533.

Philip Resnik. 1992. Left-corner parsing and psychological plausibility. In *Proceedings of the Fourteenth International Conference on Computational Linguistics*, Nantes, France.

Andrei A. Rusu and al. 2016. Progressive neural networks.

Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT press.

Niels A. Taatgen and John R. Anderson. 2002. Why do children learn to say “broke”? a model of learning the past tense without feedback. *Cognition*, 86(2):123–155.

H. van Seijen, H. van Hasselt, S. Whiteson, and M. Wiering. 2009. A theoretical and empirical analysis of Expected Sarsa. In *IEEE Symposium on Adaptive DP and RL*, pages 177–184.

Christopher J. C. H. Watkins and Peter Dayan. 1992. Q-learning. *Machine Learning*, 8(3):279–292.

Christopher John Cornish Hellaby Watkins. 1989. *Learning from Delayed Rewards*. Ph.D. thesis, King’s College, Cambridge, UK.