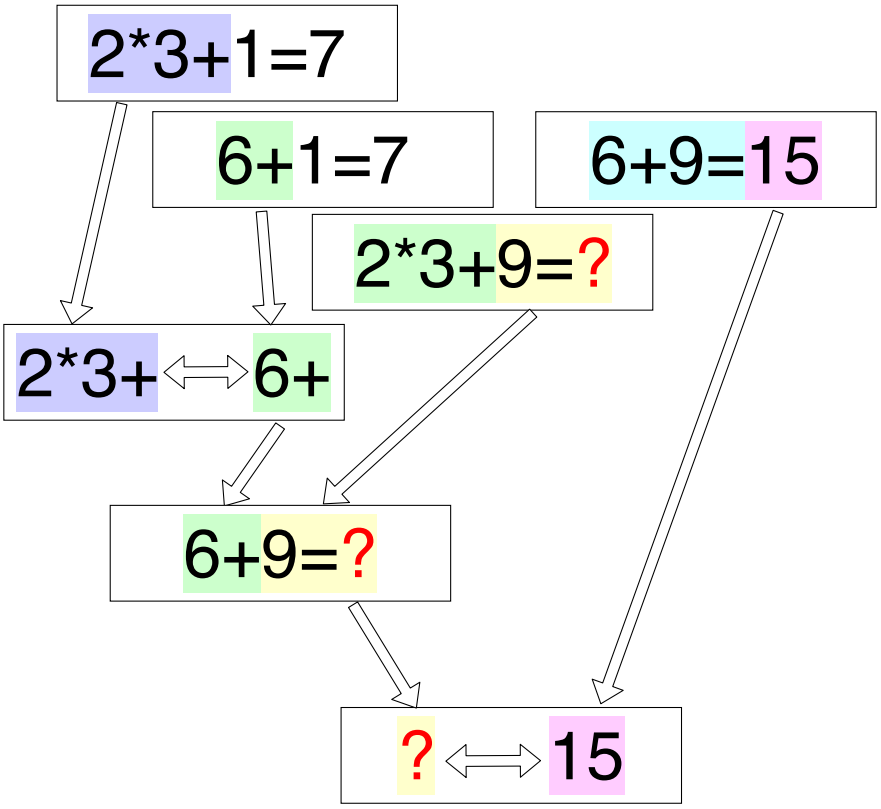
CHALMERS
UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG



Rule-Based Sequence Learning Extension for Animats

Master's thesis in Computer Science – algorithms, languages and logic

Gustav Grund Pihlgren, Nicklas Lallo

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2018

Master's thesis 2018

Rule-Based Sequence Learning Extension for Animats

Gustav Grund Pihlgren Nicklas Lallo

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2018

Rule-Based Sequence Learning Extension for Animats
Gustav Grund Pihlgren, Nicklas Lallo

Supervisor: Claes Strannegård, Department of Computer Science and Engineering
Examiner: Christos Dimitrakakis, Department of Computer Science and Engineering

Master's Thesis 2018
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Solving an artihmetic problem by using knowledge, derived from examples, of which subsequences are equivalent.

Typeset in LaTeX
Gothenburg, Sweden 2018

Rule-Based Sequence-Learning Extension for Animats
Gustav Grund Pihlgren, Nicklas Lallo
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

This thesis introduces a rule-based, sequence learning model. It proposes that parts of this model could be used as a independent extension to other machine learning models, animats specifically. The model uses Q-learning and state space search to generalize which are equivalent. This allows reducing the input state space to train faster and better draw conclusions about the features in the dataset at large. This knowledge can then be used to calculate the best action for the given sequence. The model is implemented in order to evaluate its capabilities. The model is evaluated primarily on the domains of simple arithmetic, Boolean logic, and simple English grammar and then compared to the performance of a Recurrent Neural Network using Long-Short Term Memory-units.

Keywords: Computer Science, Engineering, Machine Learning, Q-Learning, State Space Search, Neural Networks, Animat, Rule-Based, Sequence Learning

Acknowledgements

We would like to thank our supervisor Claes Strannegård for all the help and guidance throughout the project, and also thank our examiner Christos Dimitrakakis for the additional help and constructive criticism with the writing of the thesis.

Gustav Grund Pihlgren and Nicklas Lallo, Gothenburg, June 2018

Contents

List of Figures

List of Tables

1
Introduction

In 1991, S. W. Wilson [48] proposed the animat path to Artificial Intelligence (AI). An animat is described as an artificial animal put in an artificial environment where it should survive or solve some given problem. The suggested path towards AI using animats is to first develop animats that mimic the behavior or intelligence of simple animals such as flies or worms. When the animats can successfully mimic these animals we develop them further and attempt to mimic more complex animals. Through repetition of this procedure we develop complex Artificial Intelligence. C. Strannegård et al. takes it one step further and suggests that this procedure could be repeated until artificial general intelligence develops [37].

It is a common approach in the fields of animats and computer science to take inspiration from and even mimic animals and other lifeforms [7]. For example ant-colony optimization was inspired by research on how ants navigate [9]. A lot of the animat research is done in connection with robotics because of the similarities between autonomous robotic behavior and animal behavior. This can be clearly seen in the proceedings of "Animals to Animats", the leading conference on animats, from 2016 in which 10 out of 31 papers is directly related to robotics [40]. The field of machine learning is important to animats as well as robotics.

A very common model in the field of machine learning is the neural network. However, to train a neural network one often requires thousands of data-points and long training sessions. While neural networks often show great performance with a long training session they often perform very poorly with short training [19]. This means that neural networks are a poor choice when one has an learning environment where either; the number of data-points is limited or the system trains during execution (i.e. a system that needs high performance before it has finished training). Examples of such environments can be found in robotics where robots get data in real time and in animats where artificial animals must quickly adapt to their environment in order to survive [37].

In animats but also other machine learning models one of the most basic concepts is that of generalization. The ability for a animat to formulate general concepts about the properties of the environment. By finding patterns they can infer knowledge about previously unobserved data points by treating them similar to other known data points based on their common properties, that is, their similarities. This often includes grouping similar entities or datapoints together based on their properties and then treating everything in that general group similarly. This allows for higher

level thoughts and problem solving. Importantly in the context of machine learning, this allows for the models to work on data that was not included in the training dataset. In this thesis generalization specifically refers to being able to group a new problem with some other problems. If one can find a similar problem then one can use a the same decision-making process to solve the new problem. Take as an example the problem "Find the solution to $3 + 3$". This problem can be grouped with other mathematical problems since they have similar form. From this it can be determined that the same process of arithmetic can be used to solve this problem as the other problems in the group of mathematical problems.

This thesis introduces a generalization system for sequence learning called the model. This system is intended to be used as an extension to already existing machine learning systems that lack or have poor ability to generalize sequences-learning problems. Specifically the model is intended to extend the animats presented by C. Strannegård et al. [37]. However, to evaluate the model's capabilities this thesis will evaluate it as a standalone system.

This thesis also presents a neural network using Long-Short Term Memory-units. The purpose of the neural network is to be a baseline for the evaluation of the mdeol. The two systems' performance is compared on the domain of simple arithmetic on which the model holds up to and sometimes outperforms the network. The systems are also evaluated simple Boolean algebra, and simple English grammar to show their performance on other common sequence learning tasks. However on these domains the model is mostly outperformed by the network. All these domains are presented in the Datasets and Evaluation chapter and the findings in the Results chapter.

1.1 Problem formulation

Generalization is a broad concept that could be adapted to a large variety of machine learning problems. The system produced by this project is limited to the common machine learning problem of sequence learning. A thorough description of sequence learning can be found in subsection 2.1.4 of the Theory chapter.

While the aim is to introduce a general sequence learning system, the system was evaluated on the domain of simple arithmetic, simple Boolean logic, and a smaller dataset of natural English language. Simple arithmetic, as defined in the Dataset and Evaluation chapter, is a common baseline used by many other machine learning systems [26], [20]. Additionally simple arithmetic has complex connections between the start of the sequence and the end of the sequence making it a good baseline for advanced sequence learning specifically.

1.2 Research question

The central question for this thesis is:

Can a rule-based learning system outperform an LSTM network on sequence learning problems?

While this is a large question to answer this thesis will not touch on rule-base learning systems outside the one introduced in this thesis. This thesis will also go into more detail as to under which circumstances the model performs comparatively well and whether or not using the model as an extension to other systems is viable.

2
Theory

This project relies heavily on different areas in machine learning. Specifically the project relies on the subjects of; LSTM networks, state space search, and Q-learning. This chapter will introduce those fields.

2.1 Machine learning concepts

Machine learning is a big field that borrows much from other fields of computer science and mathematics. What follows are shorter summaries of some of the concepts used in this thesis.

2.1.1 Supervised learning

In machine learning, supervised learning is when you provide a system with both an input, as well as an expected output. The purpose of this is for the system to adapt such that when given that input it should return the expected output. The main difficulty with supervised learning is that you require labeled data, that is, for each entry in the training set you need to provide the corresponding correct output value. Therefore this is mainly useful on domains where there are a lot of prior knowledge.

2.1.2 Reinforcement learning

Reinforcement learning is when a system is given reward for certain achievements and supposed to learn a good decision-making process for gaining as much reward as possible. Often this means that there are some predefined states that are desired and whenever the system achieves such a state is is given the corresponding reward in the form of a real number. The goal of the reinforcement learning system is to maximize the amount of reward it can gain. There may be a trade-off between immediate reward and later reward or not depending on the specific problem at hand [28].

2.1.3 Rule-based learning

Rule-based learning includes all machine learning methods that develop rules which it then applies to the inputs. The goal of the rule-based system is to construct rules such that new data points can easily be sorted into categories which helps the

decision-making process. An example of this is classification problems where rules can divide the data points into categories based on their properties. Then the system can simply give classification depending on which category the data point ends up in [45], [23]. Rule-based systems are meant to be an alternative to specialized systems and should not need to apply any prior knowledge of the specific. However, the algorithms that find these rules often have specific patterns or relations that they look for in the data, which are often tied to some general domain. Rule-based systems typically form rules of the form `if ... then ... else` [41].

2.1.4 Sequence learning

Sequence learning is a machine learning field that aims to create systems able to make decisions based on sequences. While no agreed upon definitions exists we will, in this thesis, be using those given by R. Sun and C. L. Giles in their article "Sequence learning: from recognition and prediction to sequential decision making" [38] and complement these with one definition of our own. In the article they define four different sequence learning problems: Sequence prediction, sequence generation, sequence recognition, and sequence decision-making.

Sequence prediction is the problem of given a sequence of elements, predicting the following elements. In the one-step case only the immediately following element needs to be predicted. That is given elements $[x_1, ..., x_t]$ predict the next element $[x_{t+1}]$. Learning one-step sequence prediction is then in its most general form to find a function f such that $f([x_1, ..., x_t]) = x_{t+1}]$.
The more general case of sequence prediction is to produce the following sequence of elements. That is given a sequence $[x_1, ..., x_t]$ predict the following sequence of elements $[x_{t+1}, ..., x_{t+n}]$. The sequence stops when it reaches x_n where n is either a fix number or x_n some defined end-of-sequence element. A common approach to solve sequence prediction problems is to create a one-step sequence prediction system and run it repeatedly with the predicted elements appended to the sequence. That is $x_{t+k} = f([x_1, ..., x_t, x_{t+1}, ..., x_{t+k-1}])$ for $k \in [0, ..., n]$

Sequence generation is similar to sequence prediction where given a sequence $[x_1, ..., x_t]$ one should return a sequence $[x_{t+1}, ..., x_{t+n}]$. Mathematically prediction and generation are the same problems. Because of this and because sequence generation is not used in this thesis, sequence generation will not be explained further.
Sequence recognition is the problem of predicting whether a given sequence is correct or not. This can be seen as a special case of one-step sequence prediction where $x_t + 1 \in \{$"`True`", "`False`"$\}$

Sequence decision-making is the problem of choosing a good action or sequence of actions given a sequence of inputs. What is a good action depends on what type of decision-making problem is considered: Trajectory-oriented, goal-oriented, or reinforcement-maximizing. In trajectory-oriented decision-making the goal is to find the action a_t that leads to a desired next state s_{t+1} given a sequence of pre-

vious states $[s_1, ..., s_t]$. Goal-oriented is a generalization of trajectory-oriented; the goal is to find the action a_t given the state sequence $[s_1, ..., s_t]$ that leads to some goal state s_G at some point in the future. The goal of reinforcement-maximizing decision-making is to find the action a_j given the state sequence $[s_1, ...s_t]$ that maximizes future reward. How the future reward is calculated may vary from problem to problem.

In addition to the above definitions by R. Sun and C. L. Giles this thesis will also be using the following definition of *equivalent* sequences: Two sequences $X = [x_1, ..., x_n]$ and $Y = [y_1, ..., y_m]$ are considered *equivalent* if all sequences of actions that give the best future reward from X also gives the best future reward from Y.

2.2 Q-learning

Q-learning is a reinforcement learning algorithm for finding the procedure that optimizes the reinforcement reward for some Markov process [43]. It was introduced in the late 1980s as a simple machine learning concept that allows agents to learn how to act efficiently in simple and controlled Markovian domains [42]. It uses so called Q-tables to store the evaluation of all different actions for each known state in the domain, and then dynamically update the values after seeing the result of further actions following the current state, by using long-term discounted rewards. Discount is a factor γ that determines how much a system values future reward. The value r_v of the future reward r is given by $r_v = r \cdot \gamma^t$, where t is the number of timesteps until the reward is given. This way Q-learning can evaluate the quality of each action not only on the next timestep, but on the approximate long term benefits.

With sufficient sampling of the actions from each state in the domain it has been shown that Q-Learning will converge towards optimal action-values consistently [42]. Q-Learning is on its own a fairly simple and primitive system for learning, but it can be implemented in or extended by more complicated systems, which has been done many times in the past [42], [2].

Q-Learning also has some connections to animat based machine learning. The original paper [43] uses animals as a comparison and attempt to build upon ordinary stochastic Markov chain models with Q-Learning to better mimic animals. Here's a quote from the author, Chris Watkins: "Next, it is argued that Markov decision processes are a general formal model of an animal's behavioural choices in its environment" [43].

2.2.1 How to optimize Q-tables

The result of Q-learning is a Q-table that maps state (s) and action (a) to the expected discounted future reward ($Q(s, a)$). When a Q-table have converged the system can optimize its reward by finding the action that gives the highest future

reward from the given state.

When Q-learning starts all Q-values start at 0. Each timestep the algorithm must now decide whether to exploit or explore. Exploit means taking the action that gives optimized reward and explore means taking a random action. This decision is determined by the parameter ξ which is the probability to explore. After each action the system gets a reinforcement reward r and a new state s_{new}. Using the previous state s_{old}, the action a, and the learning rate α the system updates its Q-table as follows:

$$Q(s_{old}, a) = (1 - \alpha) \cdot Q(s_{old}, a) + \alpha \cdot (r \cdot \gamma \cdot \max_{a} Q(s_{new}, a))$$

2.3 Animats

Animat based machine learning is the concept that machine learning should try to mimic the behaviour of animals in order to produce better results. It is also useful for learning about real animals through animat simulations [44]. It is logical to compare the behaviour of a machine learning model to the behavioural policies or tendencies of ordinary animals in order to evaluate their performance and development. It is also possible to mimic how real animals learn from the environment surrounding them and teach machine learning models in a similar fashion. This is especially common when dealing with simple animals, robotics and movement [40], [47] but equally applicable to software only models [37], [43].

As previously mentioned the model presented by this thesis is intended as a potential extension to the animats presented by Strannegård et al. [37]. That project showed a system used for training animats to survive, breed, and evolve, in artificial environments called blockworlds. The animats are agents that can view the world through sensors and acts on the world using a set of actions. Each animat also have a set of needs, like hunger thirst, represented as real numbers. If any of the animat's needs reaches 0 the animat dies. The project introduces a graph-like decision structure that connects the sensors of the animat to different nodes using logic gates. The decision structure is trained using predefined reinforcement learning rules where certain events trigger a change in the decision structure. On top of this the animats uses a version of Q-learning to calculate an action from the active nodes in the network. The following subsections will go into details as to how exactly the animats work.

2.3.1 Blockworld

In the model the animats inhabit an environment called the blockworld. The blockworld is, unsurprisingly, a world divided into blocks. Each block has neighbors and attributes. An animat in the blockworld will at any given point in time inhabit exactly one block. An animat's sensors will be active or inactive depending on which attributes the block has. Note the different blocks may have the same attributes which makes it necessary for the animats to learn to differentiate those blocks by the context they appear in. For example both lake and sea blocks could have the

attributes "blue" and "wet". However if the sea blocks are always surrounded by sand blocks then the animat could learn to not drink from a block if the last block it visited was a sand block.

2.3.2 Animat decision-making

In the model each animats have two primary structures they use to make decisions; the gate network and the tables.
The gate network is a directed acyclic graph from the sensors to some top-level nodes. Aside from the sensors there are two more types of nodes in the gate network; `AND`-nodes and `SEQ`-nodes. In their paper C. Strannegård et al. presents further possible nodes, however the animat presented can only learn to form `AND`-nodes and `SEQ`-nodes. Therefore an animat that starts learning from an empty network will only have sensors, `AND`-nodes, and `SEQ`-nodes.

`AND`-nodes functions the same way as an `AND`-gate would. If its two incoming edges are active (an edge is active if the node they come from is active) the `AND`-node becomes active. The `SEQ`-node is a bit more complicated. The `SEQ`-node detects when its two incoming edges are activated one timestep after each other. The incoming edges of the `SEQ`-node are ordered such that if the first one was active in the previous timestep and the second one is active in the current timestep the node will activate as well.

Using these activation mechanics the activity in the gate network is calculated from the sensors up. This is done such that the activity of each node is determined after the activity of its incoming edges. The gate network is finally used to define the set of top-active nodes as the set of all active nodes that do not have an outgoing edge to another active node.

The tables are two tables of Q- and R-values that are used in a variant of Q-learning [42] here called local-Q-learning. The Q-table contains so-called local Q-values $Q_i(b, a)$ that gives the average change of need i when action a is performed when node b is top-active. Note that this change is not the average instantaneous reward but uses future rewards in the same way as standard Q-learning. The R-table contains $R_i(b, a)$ which is the reliability of the Q-value for the same inputs. The reliability is a directly correlated to the standard deviation of the change. The animat can now calculate the so-called global Q-value $Q_i(t, a)$ where t is the set of top-active nodes.

$$Q_i^g(t, a) = \frac{\sum_{b \in t} Q_i(b, a) \cdot R_i(b, a)}{\sum_{b \in t} R_i(b, a)}$$

Now the animat has all it needs to make a decision for this timestep. The animat will for all needs and all actions calculate the corresponding global Q-values. The animat will then select the action after performing which the expected lowest value of any of its needs is a high as possible. However to allow the animat to learn from its environment it might, with probability ξ select a random action instead.

2.3.3 Animat learning

As mentioned earlier the animat has two structures that it uses for decision-making: The gate network and the tables. The animat learns by updating these two structures based on what it experiences in the world.

The tables uses a method almost identical to that of Q-learning to update its local Q-values. If action a was taken and node b was top-active then $Q_i(b, a)$ and $R_i(b, a)$ will be updated, otherwise they will remain the same. $Q_i(b, a)$ updates based on the reward in need i, the learning rate α, the discount rate γ, and the expected optimal reward given the new global Q-value. The last parameter ensures that the animats can learn long-term thinking due to bad states being perceived as better if they in turn can lead to good states. For example an animat that walks in a desert will receive a negative reward, but since walking in the desert has at some point lead to an oasis with positive reward the act of walking while in the desert gets a higher Q-value since it can lead to something with a high Q-value. The R-values are updated to reflect the variance in their respective Q-values.

The structural learning complement the Q-learning and helps the animat form a more nuanced view of the world. If $Q_i(b, a)$ changes a lot during updating and $R_i(b, a)$ is high the animat become surprised. Surprise means that there is probably some detail to the world that the animat cannot yet grasp. Surprise therefore triggers structural learning. Structural learning will first attempt to form a AND-node between some nodes β and β' if they were both active during the timestep. An AND-node connecting β and β' will be formed if it would have changed very little during the last update.

If no `AND`-node is formed then the structural learning will instead try to form a `SEQ`-node. Forming a SEQ-node works similar to forming an `AND`-node with the difference that the node β would have to have been active in the timestep before the last one and β' active in the last timestep.

2.4 State space search

A common problem-solving approach in classical AI that has remained useful to this day is the idea of searching for the solution of the problem within the action space [50]. The approach attempts to build a graph with the state of the agent as nodes and the possible actions in each states as edges. Each edge connects the states to some other states. Given that an agent can form a sufficiently accurate graph of states and actions it can use search algorithms from graph theory in order to find the sequence of actions needed to get to some goal state. As one usually searches from the current state and branches from each state with possible actions and because it is often unnecessary to visit the same state more than once these graphs often takes the shape of tree-graphs. This makes way for usage of more efficient search-algorithms specialized on searching tree shaped graphs.

State space search is a powerful algorithm in problem-solving as given a sufficiently good graph and enough time state space search always find a solution (if there is one) and on some problems even the optimal solution. However the two major downsides of state space search in problem-solving is that it requires good graph representations of the problem and has a high time complexity [50]. In general state space search has a the worst case time complexity of $O(a^d)$, where a is the maximum number of actions for a state and d is the worst-case depth of the goal state [34].

In order to lower the time of state space search it is common to use heuristics, to lower the branching factor [31]; value functions, to allow shorter searches [34]; or a combination of the two. Recently machine learning have been used to learn better heuristics and value functions to make state space search more useful on problems with large depth and branching factor. A successful use of this can be found in Google DeepMind's AlphaGo which became the first AI to beat a world champion of the game of Go [35].

Machine learning have also been used to tackle the problem of being able to form good graphs of the problem. In the case when the effects of actions are unknown different variants of action model learning can be used to learn the state-action transitions and enable the agent to form more accurate graphs the more it interacts with the world [5]. The downside of this approach is that the agent must make actions before it can determine a path to the goal state, which might result in the agent ending up in bad states.

One way to keep track of the world is to maintain a Markov chain or, as we will refer to it, a transition table. A transition table is a table with the probabilities for a given action in a given state ending up in each other state. By keeping track of what states the agent have been in and what actions it has performed one can create a transition table which can either be the graph or help to form it. However in the case with probabilities the state space search will have to take into account the reward of a branch as well as its probability. The expected reward of an action in a given state is determined by $a_r = p_1 \cdot r_1 + p_2 \cdot r_2 + ... + p_n \cdot r_n$, where p_k is the probability of the action going from the given state to state k and r_k is the reward of state k.

2.5 Alice in Wonderland

A work that introduces a rule-based sequence learning system is "Bounded Cognitive Resources and Arbitrary Domains" [29]. The work introduces a system called "Alice in Wonderland" which given a subset of sequences from some set attempts to learn the underlying rules for whether or not a given sequence is in that set. For example it can be given the sequences "1+0=1" and "2+0=2" and create the abstract hypothesis that "X+0=X" where X is a character from some subset of characters.

"Alice in Wonderland" determines the value of a given sequence by attempting to simplify the sequence using a finite set of rules it has learned over the cause of train-

ing in combination with state space search. Alice will simply attempt every possible order simplification rules on the sequence up to a maximum length of a rule chain, then it will pick the one of the resulting sequences that has the simplest form. Due to a bound on both the number of rules and the length of simplification chains a upper time-bound can be guaranteed.

Alice learns the rules by trying to find pattern-matching hypotheses such that each hypothesis does not contradict any expression it has been given and that as many expressions as possible can be simplified to that hypothesis. For example given the expressions "1+0=1", "2+0=2", "1 $\neq$ 0" Alice will probably form the hypothesis that "X+0=X" since it is consistent with all expressions and this hypothesis can be used to correctly evaluate two-thirds of expressions.

2.6 Neural networks

Neural networks is a catch-all term for a large group of machine learning systems that take inspiration from biological neural networks found in animals (i.e. their brains). Neural networks in general work by connecting a number of usually simple computational units called neurons into a network of inputs and outputs. While the inner-workings of neurons can vary between systems they usually work by taking the weighted sum of all their input connections, running it through a so-called activation function, and outputting the result to all its output connections. Since the inputs, outputs, and weights all can be represented as vectors neural networks can often be represented by matrix calculations.

In recent years neural networks have seen a rise in popularity as they have been successfully used to achieve state-of-the-art results on a large number of problems. Image-recognition [36], [14], speech-recognition [12], [15], playing Go [35], and playing Atari[27], to name a few.

Artificial Neural Networks are useful for many different types of problems. Two of the more common are the classification problems, and the regression problems. Linear Regression is a very useful tool from statistical processes for estimating how different variables relate to each other, and how to minimize a function, and therefore also have much use in machine learning. Neural Networks when applied to a regression problem usually try to minimize a loss function in a similar fashion, and have shown good results at doing this [30].

2.6.1 Recurrent neural networks

Recurrent Neural Networks (RNN) are a type of neural network that allow for persistent knowledge [17], [46], they use a form of memory in order to allow the neurons in the gate network to loop information back between iterations, thus allowing the network to weight previous iterations into making future decisions. This is very useful when dealing with sequential inputs, such as for example text sentences.

RNNs work similar to ordinary neural networks except these special Neurons [16]. There are several different versions of these neuron units that all implement the memory in different ways, such as simple RNN cells or more advanced LSTM cells [16].
RNNs are very useful for sequence prediction and generating [11], since such sequential data structures require the network to keep some information about previous states (commonly words) as it continues forward. It has been shown that LSTM networks, a specialized form of RNNs, perform well on sequence prediction problems with complex sequences [10], [11], [16].

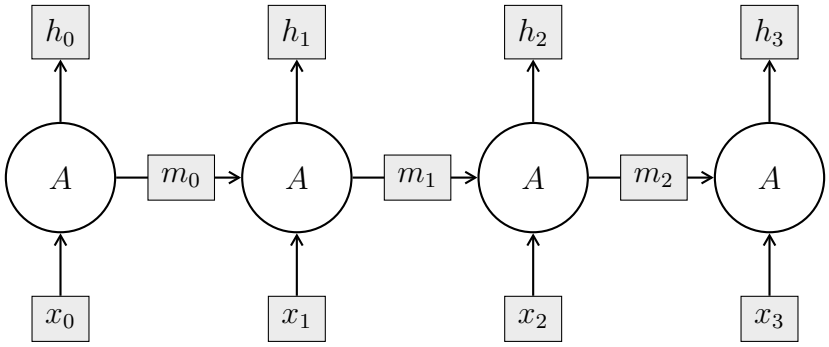


Figure 2.1: An RNN model unrolled over 4 timesteps.

Figure 2.1 shows the flow of data through an RNN. In the figure A is the RNN architecture, x_t the input, h_t the output, and m_t the memory vector at timestep t. It shows how during each timestep, for each input the network produces both a regular output as well as a memory vector that is used as additional input for the next timestep.

2.6.2 LSTM

LSTM networks, or Long Short Term Memory networks, are a special kind of recurrent neural networks. Many recent breakthroughs in sequence learning have been using LSTM networks [8], [39], [49]. They were first invented by Hochreiter & Schmidhuber [16] in 1997 but much work has been made to expand upon the initial concept in the time since.

One of the main problems of traditional RNNs is the way it struggles with long-term dependencies. That is, if a certain input x_t would be useful for predicting the output h_T, then the difference in time between t and T can't be that big, otherwise the RNNs would become unable to learn to connect the information over such a long time. This has been studied in depth [3].

LSTMs solve this problem by having the ability for both short term memory, as well as long term memory. To do this the neurons use so called gates to regulate what information to add and to remove. The gates are composed of sigmoid neural network layer, this layer outputs values between zero and one which are then multiplied with the information to determine how much is let through. There are three of

these gates that together control the cell's memory capabilities. These are trained together with the rest of the cells using backpropagation.

Between each iteration of running the network the LSTM cells both deliver a output, but they also update their internal cell state. The gates are useful for controlling how the cell state interacts with any new inputs to create the next output as well as the next cell state. The exact implementation of these LSTM cells and gates tend to vary a bit from paper to paper, but for the purpose of this these we use that of the original paper [16].

2.6.3 Deep neural networks

Deep Neural Networks (DNNs) are neural networks that implement so called Deep Learning. DNNs use multiple hidden layers of neurons connected to each other in between the input and the output. This allows the network to model significantly more complex problems and non-linear relationships, in addition to this it also helps the network to model the same problems as a shallow network can but now with significantly fewer neurons. DNNs fall under the machine learning category deep learning

LeCun et al. describe that "Deep learning allows computational models that are composed of multiple processing layers to learn representations of data with multiple levels of abstraction." [24]. Deep learning has also brought about many of the recent machine learning breakthroughs, especially in sequential datasets such as text prediction [25]. Furthermore it has also commonly known that the performance off deep learning increases with the size of the training set better than most other machine learning models [13].

2.6.4 Stochastic gradient descent

Stochastic Gradient Descent is an iterable optimization algorithm for differentiable functions [4]. It is used to search for minimums of a function by calculating the gradient of the function at some point, and then finding a new point by stepping along the negative gradient. The step length is proportional to the absolute value of the gradient which means that the steps will be longer if the absolute value is large. For continuous functions this means that gradient descent will slow down as it approaches a minimum.

The goal of any machine learning model is to solve an underlying optimization problem, which in turn translates to the accuracy of the model or a loss function. Stochastic Gradient Descent is one of the optimization algorithms that help the the model converge towards a local or global minimum faster [21]. The basic gradient descent algorithm uses only the first order derivative of the function but more advanced versions also exists, although potentially slower in real time applications.

The stochastic element is useful to help the algorithm escape from local minimum

[32], something that is not guaranteed with ordinary gradient descent. This is usually commonly implemented as a stochastic objective function upon which the first-order gradient descent algorithm is applied.

2.6.5 Backpropagation

Backpropagation is an optimization algorithm for neural networks and is used to train the weights and other potential parameters of the network. Backpropagation is fundamentally gradient descent used to minimize the loss function of the neural network. Given the loss function ($E(y_o, y_e)$, where y_o is the output and y_e the expected output) of the neural network gradient descent is used to optimize the network. By having the neural network consist entirely of differentiable functions the derivative of the loss can be calculated with respect to the parameters of the network. From these derivatives one can calculate for each weight and variable whether it needs to be larger or smaller to minimize the loss. By updating all parameters accordingly with some learning rate the outputs gets closer to the expected outputs as the loss function gets minimized [6], [33].

A problem arises when one tries to use backpropagation on RNNs. In simple feed-forward networks the derivative of each parameter with respect to the loss depends only on the later parameters in the network. Thus network can be easily backpropagated by going from output to input. However in the case of RNNs the derivative of a parameter depends on the parameter's next value as well. So if one wants to train a RNN to take previous states into account one have to backpropagate not only from output to the input of this timestep but also from output to the input of previous timesteps as well. This means that a lot of information of what each neurons output were each timestep has to be stored. When this is done the network becomes unrolled. Rather than seen as a single network with memory it is seen as several networks with the same parameters that feeds their values forward. Another problem with this is that one cannot make the network infinitely large and thus can only unroll it a finite number of times. This parameter must be chosen carefully as the length of association in an RNN becomes more limited with shorter unrolling but the training takes longer the longer the unrolling is. Figure 2.1 contains an RNN unrolled over 4 timesteps.

2.6.6 Adam optmizer

In order to improve upon the basic stochastic gradient descent algorithm there exist several optimizers. One of these is the Adam Optimizer [22]. Adam improves upon and combines two other main optimizations, namely: Adaptive Gradient algorithm (AdaGrad) and Root Mean Square Propagation (RMSProp).

AdaGrad improves training by using individual learning rates for each parameter as well as updating these learning rates dynamically. Parameters that get small or infrequent updates have higher learning rates while parameters with large or frequent updates have smaller learning rates.

RMSProp also uses individual learning rates. These learning rates are calculated by the sum of squares of previous gradients. To prevent the sum from simply growing indefinitely the algorithm decays the values of previous timesteps. Simply put the new learning rate for some point p is given by:
$\gamma_p^{new} = d \cdot \gamma_p^{old} + (1 - d) \cdot \nabla_p^2$, where d is the decay factor, γ the learning rate, and ∇ is the gradient.

2.7 TensorFlow

TensorFlow is a free and open source library developed and produced at Google [1]. TensorFlow is written for the programming language Python to act as a framework for machine learning and more specifically neural networks. Tensorflow provides many highly optimized and functional implementations of various Neural Network concepts that will be used to produce some of the results in this thesis, such as Long Short Term Memory Neurons.

2.8 Radix tree

A Radix Tree, sometimes called a compact prefix tree, is a simple data structure, usually used to store sequences. The purpose of a radix tree, or even a ordinary prefix tree, is to have a tree structure representing a set of sequential data such that each subsequence of data that occurs multiple times (in the same locations) in the set only has to be included once. In a ordinary prefix tree each edge represents one or more bits of sequential data, and each child to each node represents the different combinations of data that can follow the previous sequence. So for words for example each edge could represent a single character, and going from the root node down to one of the end nodes, each word in the dataset can be read. This allows for searching the tree by the prefixes of the included data, thus they make excellent search trees.

The radix tree introduce a simple optimization to the ordinary prefix tree by making it so that each node that is the only child node gets combined with the parent node. This allows for each edge to store not only one prefix element, but a sequence, at the same time. This can make the radix tree more space efficient than a prefix tree for longer sequences in smaller datasets.

3

Datasets and Evaluation

The model and LSTM network implemented in this thesis were trained and evaluated on multiple datasets. In this chapter each of these datasets and how the systems were evaluated on them will be explained in order. The implementation of the model and the LSTM network can be found in their respective chapters. The results of the evaluations on these datasets are presented in the Results chapter.

The following method is used to represent the sequences in this thesis:

- Any symbol is marked with a box around them ($\boxed{Symbol}$) to keep them separate from any other meaning of these characters.
- Any character not in a box is a variable used to represent some subsequence of symbols.
- To allow the systems to finish sequences ϵ is used as a special end-of-sequence symbol ($\boxed{\epsilon}$).

3.1 Evaluation domains

The system was mainly evaluated on the domain of simple arithmetic which was supplemented by additional problems to show its capabilities and limitations. What these domains are and how the system was evaluated in each is described in this section.

The model and network was evaluated on four datasets from the domains of simple arithmetic, Boolean logic, and simple English grammar to show its capabilities. These domains are introduced in the following sections. All datasets were created for this thesis rather than being taken from somewhere else.

Note that each sequence in each dataset have two parts. The first part which is the input sequence to the systems and the second part which is the expected output. In all domains, except simple English grammar, the first part is all symbols up to and including the equality sign ($\boxed{=}$) and the second part is all remaining symbols. For the domain of simple English grammar the entire sentence is the input and the expected output is a simple Boolean for whether or not the sentence was grammatically correct.

3.1.1 Simple arithmetic

The arithmetic dataset includes sequences of numbers, operands, and the equality sign. All symbols are only identifiable by their index in a symbol lexicon. This dataset was interesting because it required a lot of different functions from the machine learning model.

Simple arithmetic has no rigid definition, the definitions in this section are based on commonly used baselines with increasing complexity to be able to compare the performance of the system as the problems get more complex. In general an expression in the domain of arithmetic follows the form "*expression=digits*" In this domain the system is trained by feeding it correct arithmetic expressions as well as incomplete arithmetic expressions and allowing the system to decide on an action. Each action corresponds to printing one of the digits 0 to 9 or to end the sequence (ϵ). The model receives reward only when it takes the action corresponding to end-of-sequence and will then receive a positive reward for outputting a correct expression and a negative reward for an incorrect one.

Within the domain of simple arithmetic several different datasets were used to evaluate the model. These were of varying size and complexity and are each presented in the following subsections

3.1.1.1 Equalities, single-, and double multiplications of single digit numbers

This dataset, called Arithmetic1, represents all single digit multiplications with two and three operands. Arithmetic1 contains 1400 sequences consisting of the following symbols:
$\boxed{0}$, $\boxed{1}$, $\boxed{2}$, $\boxed{3}$, $\boxed{4}$, $\boxed{5}$, $\boxed{5}$, $\boxed{6}$, $\boxed{7}$, $\boxed{8}$, $\boxed{9}$, $\boxed{*}$, $\boxed{=}$, $\boxed{\epsilon}$

The sequences are defined as follows:

- $x\boxed{=}x\boxed{\epsilon}$, $\quad \forall x \in \{\boxed{0}, ..., \boxed{3}\boxed{0}\boxed{0}\}$
- $x\boxed{*}y\boxed{=}x \cdot y\boxed{\epsilon}$, $\quad \forall x, y \in \{\boxed{0}, ..., \boxed{9}\}$
- $x\boxed{*}y\boxed{*}z\boxed{=}(x \cdot y \cdot z)\boxed{\epsilon}$, $\quad \forall x, y, z \in \{\boxed{0}, ..., \boxed{9}\}$

So, for example, when given the following sequence as input:
$\boxed{4}\boxed{*}\boxed{5}\boxed{=}$
The corresponding correct output would be:
$\boxed{2}\boxed{0}\boxed{\epsilon}$

3.1.1.2 Equalities, single- and double multiplications and additions of single digit numbers

This dataset, called Arithmetic2, is an extension of Arithmetic1 where the additional operator of addition. Arithmetic2 contains 4500 sequences consisting of the following symbols:
$\boxed{0}$, $\boxed{1}$, $\boxed{2}$, $\boxed{3}$, $\boxed{4}$, $\boxed{5}$, $\boxed{5}$, $\boxed{6}$, $\boxed{7}$, $\boxed{8}$, $\boxed{9}$, $\boxed{*}$, $\boxed{+}$, $\boxed{=}$, $\boxed{\epsilon}$

The sequences are defined as follows:

- $x\boxed{=}x\boxed{\epsilon}$, $\forall x \in \{\boxed{0}, ..., \boxed{3}\boxed{0}\boxed{0}\}$
- $xay\boxed{=}\texttt{Result}\boxed{\epsilon}$, $\forall x, y \in \{\boxed{0}, ..., \boxed{9}\}, \forall o \in \{\boxed{*}, \boxed{+}\}$
- $x_1o_1x_2o_2x_3\boxed{=}\texttt{Result}\boxed{\epsilon}$, $\forall x \in \{\boxed{0}, ..., \boxed{9}\}, \forall o \in \{\boxed{*}, \boxed{+}\}$

3.1.2 Boolean Algebra

The domain of Boolean Algebra is very similar to that of simple arithmetic, but has fewer possible symbols to write; $\boxed{\texttt{True}}, \boxed{\texttt{False}}, \boxed{\epsilon}$. Training in this domain is exactly like that of simple arithmetic with the exception that only a single output character is required (in addition to the $\boxed{\epsilon}$).

This dataset, called Logic1, contains 170 sequences defined as follows:

- $x_1\boxed{=}x_1\boxed{\epsilon}$, $\forall x \in \{\boxed{\texttt{True}}, \boxed{\texttt{False}}\}$
- $x_1o_1x_2\boxed{=}\texttt{Result}\boxed{\epsilon}$, $\forall x \in \{\boxed{\texttt{True}}, \boxed{\texttt{False}}\}, \forall o \in \{\boxed{and}, \boxed{or}\}$
- $x_1o_1x_2o_2x_3\boxed{=}\texttt{Result}\boxed{\epsilon}$, $\forall x \in \{\boxed{\texttt{True}}, \boxed{\texttt{False}}\}, \forall o \in \{\boxed{and}, \boxed{or}\}$
- $x_1o_1x_2o_2x_3o_3x_4\boxed{=}\texttt{Result}\boxed{\epsilon}$, $\forall x \in \{\boxed{\texttt{True}}, \boxed{\texttt{False}}\}, \forall o \in \{\boxed{and}, \boxed{or}\}$

Example input:
$\boxed{\texttt{False}}\boxed{or}\boxed{\texttt{True}}\boxed{and}\boxed{\texttt{True}}\boxed{and}\boxed{\texttt{True}}\boxed{=}$
Corresponding correct output:
$\boxed{True}\boxed{\epsilon}$

3.1.3 English grammar

The domain of English grammar is a set of grammatically correct and incorrect short sentences in English. In this domain the model has only two actions $\{\texttt{True}, \texttt{False}\}$ for whether or not the sentence is grammatically correct. The model will receive positive reward if it correctly classifies a given sentence. This dataset is called Grammar1, contains 707 sentences, and can be found in Appendix A.

Some examples of grammatically correct sentences from the dataset:

- it is delightful
- we are young
- they're free

Some examples of grammatically incorrect sentences from the dataset:

- you active
- wrong we're
- I'm walked

For both the model and the network each sentence have been replaced by a representative sequence in which each unique (blank space separated) word is mapped to a single symbol and all blank spaces have been removed. For example: With the mappings of "we" $\rightarrow \boxed{1}$, "are" $\rightarrow \boxed{3}$, and "young" $\rightarrow \boxed{2}$ the sentence "we are young"

becomes the sequence 1 3 2.

Example input:
2 1 3
Corresponding correct output:
`False` ϵ

3.2 Training and evaluation

At the beginning of each training session each dataset was randomly split into two parts: The training set and the validation set. The validation set is created by taking a fraction, henceforth called the *fraction as validation*, of the dataset randomly. The remaining entries of the dataset forms the training set.

In the simple case of the dataset Arithmetic1 with a fraction of validation of 0.1 this would correspond to having a machine learning system being fed most of the multiplication table from 0 to 9 and most equalities between 0 to 300 and asking it to figure solve the remaining table entries.

Training and evaluation of the model and network were done in similar fashions. The two systems were each fed entries, both input and expected output, from the training set to learn what inputs had what expected output. After being fed a certain number of training set data the systems' current state were saved, prevented from learning, and then asked to calculate their output for every entry in the validation set. The fraction of the systems' output the match the expected output is recorded, the systems are restored to their saved states, and learning is reactivated. The systems are then, once more fed a number of training set entries. This process is repeated such that, in total, the model is fed the training set five times and the network is fed 300000 entries from the training set. How each of the two system uses the given training data is further specified in their respective chapters.

This process was repeated a few times for each dataset for three different fractions as validation (0.1, 0.5, and 0.9). For each repetition the training and validation sets were randomly created anew to prevent bias that can exist in a single validation set.

With regular intervals both models were paused during the training for evaluation. During this they receive no feedback so the weights or tables in the models were not updated. They are shown each entry in the validation set one by one with a reset in between, and their accuracy at predicting the correct results were measured. The accuracy were recorded and plotted on graphs to show the progression over many training iterations. This is shown in the next chapter, Results.

For both systems evaluation is straight forward. The systems are given a input sequence and told to output the best action in regards to that input. Since this in all presented domains is sequence prediction each action corresponds directly to which symbol should come next. If the system outputs the wrong symbol the evaluation

of that input is counted as incorrect and the evaluation proceeds to the next input. If the correct symbol is outputted the system is fed the input sequence once more with the new symbol appended at the end. This process is repeated until a wrong symbol is outputted, which is counted as incorrect, or all of the expected output has been outputted, which is counted as correct.

Note that the length of the output has a max size (x) in all domains and both systems will automatically output the end-of-sequence symbol as the xth symbol.

4
The Model

This chapter describes the model created in the project. The chapter begins with a high-level description of the model and proceeds to further detail the exact implementation.

4.1 Interface

This section describes how the model interacts with its environment. While the model have been evaluated on sequence prediction tasks it is better described as a sequence decision-making system. Like a sequence decision-making system, the model is given a sequence and outputs an action.

This means that the model will only take one action per input sequence. If one wants to use it to predict a sequence of actions one can simply feed it the original sequence again but with a new symbol appended to the end. Which symbol this is depends on the action and the environment. Since the action does not necessarily correlate directly to a symbol the function $f^e(y)$ is used to compute the symbol to be appended for some action y in the current environment e.

f and e may vary from problem to problem. Take an example where the model controls an agent on a grid where each action is a direction to walk. In this example e could be the the agent's position and $f^e(y)$ could be the new position of the agent. Thus $f^e(\texttt{north})$, where $e = (3, 4)$ would be $(3, 3)$. When using the model for sequence prediction, however, f is the identity function which does not depend on e.

In addition to a sequence the model is also given a reward for the previous action. In general this would make the model a reinforcement learning system. In this thesis, however, this reward is given solely to inform the model whether it has finished a task or not and if it has whether it finished successfully or not. When the model is fed a new starting sequence the the reward is set to `null`. This is so the model understands that it have not done anything to be rewarded yet and that the current sequence is a starting sequence.

In Figure 4.1 the models interaction with input and output is described over three timesteps (t to $t + 2$).

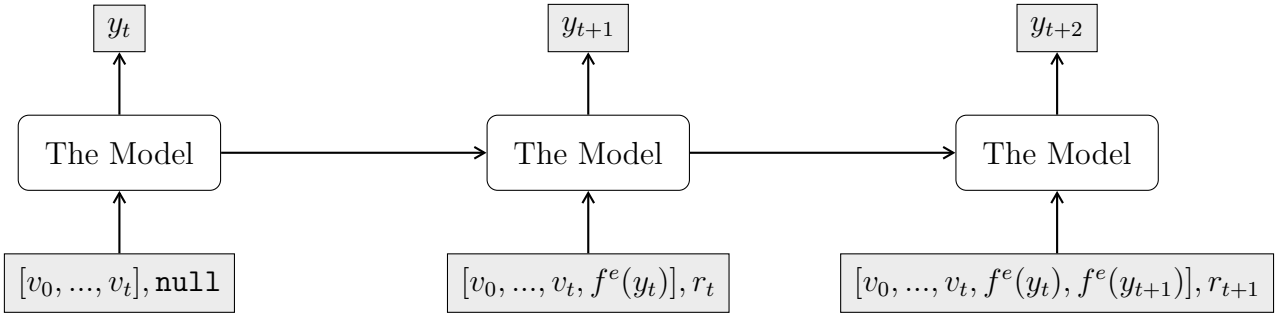


Figure 4.1: The inputs and outputs of the model over 3 timesteps.

The model has two modes of operation: Training and validation. During validation the model is always given `null` as reward and after a task is completed the model is restored to the state it was before. This mode is used to be able to the test the the model on the same validation data after several training iterations. During training the model adapts to rewards in order to optimize the reward given for each task. How the model achieves this is explained in the remainder of this chapter.

4.2 Overview

This section gives an overview of the inner workings of the model and introduces the three different parts of it: The *Solver*, the *Abstracter*, and the *Rule-Former*.

The Solver is a sequence-decision making system on its own. Given a sequence the Solver will output an action. However, the Solver is based on Q-learning and can therefore only make decisions for sequences it has encountered previously. The purpose of the Abstracter is to transform a given sequence to something the Solver have encountered before. Sometimes this cannot be done which is why the Abstracter can also create goals that the Solver can use as though they were previously encountered sequences. In order for the Abstracter to do this it needs information on how sequences can be transformed and how to create abstract goals. This is the purpose of the Rule-Former. The Rule-Former crawls through the data collected by the model at regular intervals. When doing so it looks for patterns. Patterns indicating which type of sequences are rewarded, patterns indicating which sequences have the same rewarded outputs, or any other patterns that might be useful. After crawling through the data the Rule-Former can form *transformation rules* and *abstract goals* which the Abstracter can use to transform a sequence or generate goals. The interactions between these three subsystems is further explained in the following sections and can be seen in Figure 4.2.

The Solver uses Q-learning, state space search, and transition tables to learn and predict the consequences of different actions. While the Solver is a working sequence decision-making system on its own it has some notable flaws. Due to the combinatorial explosion the number of possible sequences become incredibly large even with limited sequence length and number of symbols. This leads to the problem of encountering sequences for which no possible good action has been found. To

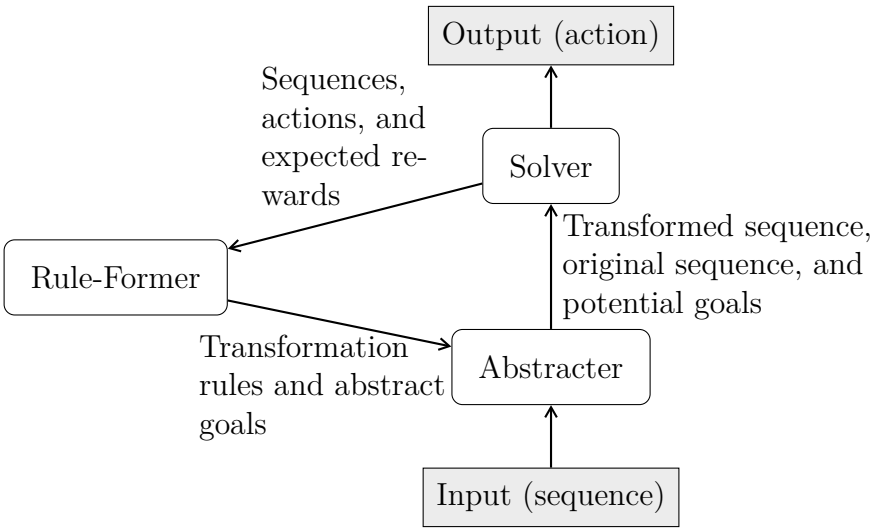


Figure 4.2: Overview of how the three parts of the model interact.

combat this the model needs a system that preprocesses the sequences and gives new sequences and potential goals so that the Solver can compute good actions. This system is, as previously mentioned, the Abstracter.

The Abstracter is a fairly simple system. Given a sequence it will use the transformation rules and abstract goals from the Rule-Former to attempt to transform the sequence into a known sequence or a sequence for which it can create a goal. It does this by using state space search over the transformation rules to transform the sequence in as many different ways as possible. Once it finds a sequence the Solver have managed to solve previously it outputs that sequence to the Solver. If it cannot find such a sequence it may find a sequence that matches some pattern the Rule-Former believes to be rewarded and it will output that sequence and its potential goal. The Abstracter will always give the Solver the original sequence as well so that the Solver knows for which sequence it is really making decisions.

The Rule-Former's purpose generate transformation rules and abstract goals. It does this by going through the knowledge gathered by the Solver trying to find common patterns in sequences. For example the Rule-former might find that the sequences "1+1=", and "2=" has the same best action (i.e "2"). The thing that is different in the two sequences are "1+1" and "2". It will then form the candidate transformation rule that "1+1" is replaceable with "2" and vice-versa. It will test the rule by seeing if, for all other known sequences, the rule is useful at least once and never bad. If that is the case it will keep the rule. Similarly it can find common patterns among rewarded sequences to find abstract goals. For example the following sequences are rewarded "1=1", "23=23", and "13=13", then the system might find the pattern that "$X = X$" where X is a sequence of symbols from the set $\{\boxed{1}, \boxed{2}, \boxed{3}\}$.

4.3 Transformation rules and abstract goals

This section describes the implementation of the two kinds of rules and how they are used.

There are two kinds of rules that the Abstracter can use: Transformation rules and abstract goals. They fill different function but their implementation is very similar. Both rules have an *abstract sequence* that can match, partially match, or not match a given sequence. An abstracted sequence is a sequence of symbols and *symbol-variables*. A symbol-variable is a pattern that some parts of a sequence may match.

4.3.1 Abstract sequences and symbol-variables

An abstract sequence is a template for a certain type of sequences. The abstract sequence is said to match a sequence if that sequence follows the template. If a abstract sequence matches some sequence it is also said to partially match all sequences that can be created by removing any number of symbols from the end. This means that if an abstract sequence matches $\boxed{1}\boxed{2}\boxed{3}$ it will at least partially match $\boxed{1}\boxed{2}$.

To create this template there needs to be some freedom for which symbols can appear. This is the job of symbol variables. A symbol-variable is a part of the abstract sequence that can match several different subsequences. Subsequences may match to a symbol-variable if they consist only of the symbols that symbol-variable allows and are not longer or shorter than that symbol variable allows. When a symbol-variable have matched to one subsequence that same symbol-variable must refer to that exact subsequence for the duration of that context. With this the abstract sequences can represent, for example, mathematical expressions. The following abstract sequence would represent the fact that $x + x = 2x$:

- Exactly one symbol from $\{\boxed{1}, ..., \boxed{9}\}$. We call this subsequence A
- Zero or more symbols from $\{\boxed{0}, ..., \boxed{9}\}$. We call this subsequence B
- Exactly one symbol from $\{\boxed{+}, \boxed{*}, \boxed{-}, \boxed{/}\}$. We call this subsequence C
- $AB\boxed{=}\boxed{2}\boxed{*}AB$

Symbol-variables are implemented as 3-tuples of: A set of symbols which the subsequence may contain, a integer which the subsequence must be longer than or equal to, and another integer that the subsequence must be shorter than or equal to. Take, for example, the symbol-variable $X = (\text{Allowed symbols} = \{\boxed{0}, ..., \boxed{9}\}, \text{Minimum length} = 1, \text{Maximum length} = \infty)$. X can match any subsequence of at least one digit.

Furthermore a specific symbol-variable must represent the same subsequence throughout an entire sequence if the sequence should be considered a matching the given abstract sequence. This means that, with X from the previous example, the abstract sequence $[X\boxed{=}X]$ matches any sequence of mathematically correct equalities

(and some incorrect ones where X begins with 0). The same abstract sequence will also partially match any sequence of digits followed by an equals sign.

A symbol-variable can be specified to match any possible subsequence. Such symbol-variables will be subscripted with *all* for readability. For example: $Y = (\text{Allowed symbols} = \text{all}, \text{Minimum length} = 0, \text{Maximum length} = \infty) = Y_{all}$

4.3.2 Abstract sequence matching

To determine if an abstract sequence matches a given sequence a simple algorithm was developed. It works by, for each symbol or symbol-variable in the abstract sequence, trying to match the symbols of the given sequence. In the end the algorithm returns the list of all possible ways that the abstract sequence can match the given sequence. In the special case where the result is only a partial match it will be flagged as such.

The algorithm is recursive and is given the current positions in the abstract sequence and the sequence to match, and a list of which symbol-variables has been matched to what subsequences. A flowchart of the complete implementation can be found in Figure 4.3. The recursive function is called with `abs` as the abstract sequence, `seq` as the sequence to match, `absPos` and `seqPos` as 0, and `vars` as a map from each unique symbol-variable in the abstract sequence to uninitialized values. Note that additional calls uses copies of the input such that no function call will overwrite the values of another.

The output of the algorithm is a list of mappings from variable-symbols to what they matched against. Each entry in the list is a mapping to subsequences such that if the symbol-variables are replaced with those subsequences the original sequence is formed from the abstract sequence. For partial matches the abstract sequence becomes a the start of the original sequence instead. If there are no entries in the list that means there are matches or partial matches. If each entry in the list is flagged as partial than it is only a partial match. In all other cases there is at least one match.

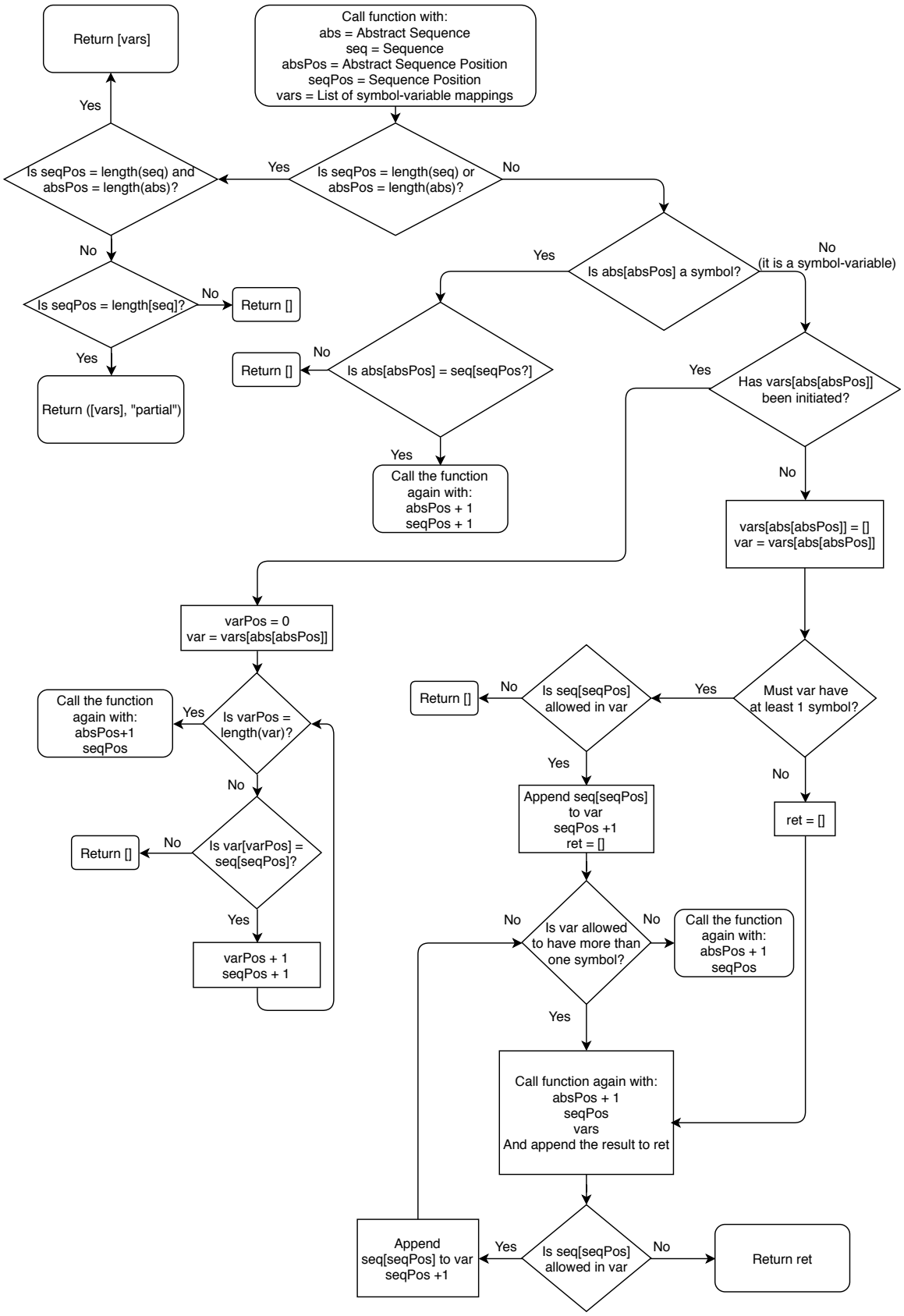


Figure 4.3: Flowchart of the abstract sequence matching algorithm.

4.3.3 Transformation rules

A transformation rule is a rule that tells the Abstracter a way to alter a sequence such that the resulting sequence is equivalent to the the original. Transformation rules are implemented as a 2-tuple of abstract sequences. If a sequence matches the first abstract sequence in the tuple it means that the rule applies to that sequence. The matching algorithm returns a mapping of the symbol-variables to subsequences. By exchanging the symbol-variables in the second abstract sequence for their respective mappings a new sequence is formed. For efficiency the rule transformation forces the algorithm to return once one match have been found and then uses the mapping of that match.

An example of a transformation rule could be $([X_{all}\boxed{1}\boxed{+}\boxed{1}Y_{all}], [X_{all}\boxed{2}Y_{all}])$. This rule is applicable to all sequences containing the subsequence "1 + 1" and states that such a subsequence can be replaced with "2". According to the rule the sequence "1 + 1 = 2" is equivalent to "2 = 2", which seems correct. However, the rule also states that "11 + 11 = 22" is equivalent to "121 = 22". This specific rule would therefore only be useful in domains in which "11 + 11 =" is not a valid sequence or where "11 + 11 = 22" is indeed equivalent to "121 = 22".

4.3.4 Abstract goals

An abstract goal is an abstraction of what sequences are rewarded. The Abstracter can use abstract goals to check whether or not a found sequence that the model has never encountered is rewarded. Abstract goals are implemented as a 3-tuple of an abstract sequence, an expected reward, and a best action. If a given sequence is a partial-match to the abstract sequence and all symbols of some subsequence needed to make a complete match can be achieved with the model's actions the sequence can lead to the given rule.

An example of an abstract goal could be $(X\boxed{=}X, 1.0, \text{print}(\boxed{\epsilon}))$, where X can be any sequence of at least one digit. This goal states that if one prints the end-of-sequence symbol ($\boxed{\epsilon}$) after a sequence of digits followed by an equals sign followed by the same sequence of digits one can expect a reward of 1.0. Using this abstract goal the Abstracter would consider the sequence "34 =" to be a sequence that can lead to a reward for the system (given that the model can produce the symbols "3" and "4" in the given domain).

4.4 Abstracter

The purpose of the Abstracter is to enable the model to calculate what action to perform when given a sequence the model never have encountered before. When given a such a problem the Abstracter will return a sequence which it believes to be equivalent with the given sequence. Sometimes this equivalent sequence have not been encountered previously either, in that case the Abstracter also provides goals for the Solver to search for. To do this it uses the transformation rules and abstract

goals produced by the Rule-Former as well as the rewarded sequences recorded by the Solver.

The usefulness of such an approach can be seen in the following example. In a text prediction scenario the training dataset could, among others, contain two sequences of the form: [Today][it][is][sunny] and [Today][it's][sunny.]

In this case the Abstracter could help by using the rule that [it's] can be replaced by [it][is] in order to reduce the new sequence [Today][it's][raining] to the already encountered equivalent sequence [Today][it][is][raining] for which the model already knows what to do. Without the Abstracter the decision making process would not have any data that matches the above sequence.

The Abstracter takes, as input, a sequence and outputs an estimated goal and an estimated equivalent sequence from which the goal can be reached. To do this it uses the transformation rules and abstract goals produced by the Rule-Former as well as a tree of rewarded sequences from the Solver. If no good estimated goal and sequence can be found it simply returns the sequence given and a `null` value as the estimated goal.

In another scenario the Abstracter may be given the sequence [4][=]. It cannot find any equivalent sequnces but can find the abstract goal X[=]X which partially matches the sequence. The Abstracter will then output [4][=] to the Solver together with the goal [4][=][4] created from matching the sequence to the abstract goal. While the Solver have no previous experience for the sequence it can use the goal to figure out that outputting [4] is probably a good decision.

The Abstracter does this by using a fairly straight-forward state space search algorithm with limited depth, no value function, and no heuristics. The nodes of the search graph are the different sequences found and the edges are the transformation rules applicable to that sequence. If a previously encountered node is found it is discarded. If a sequence found either can lead to a sequence in the list of rewarded sequences or is a partial match to an abstract goal, then that sequence and its estimated reward is recorded. When the maximum depth of the algorithm is reached it returns the sequence and corresponding goal with the highest recorded reward. The Abstracter can also run on a mode where it will terminate upon finding the first sequence with an estimated reward above a certain number to speed up performance on certain domains. A flowchart of the Abstracter's state space search can be found in Figure 4.4.

The Abstracter uses this search algorithm to reduce sequences to known sequences. In Figure 4.5 one can see an example of the nodes visited by the search as it reduces the sequence "2*5*4*(1+2)" to the, in the arithmetic domain, equivalent sequence "120".

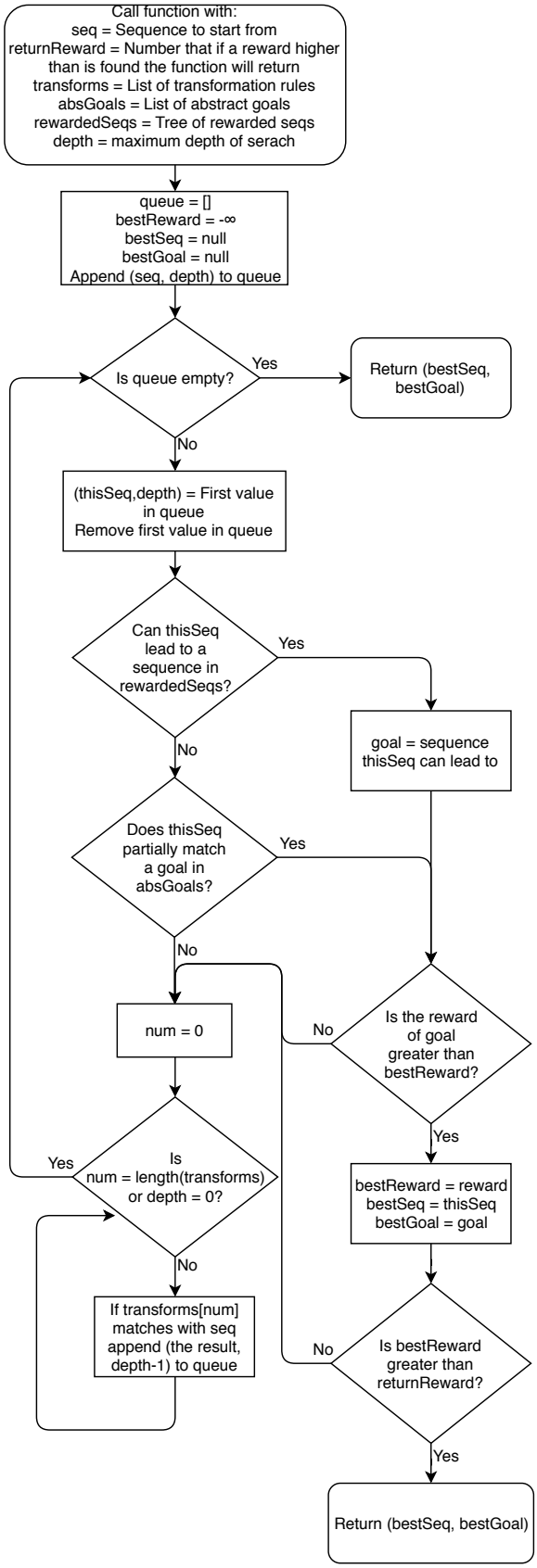


Figure 4.4: Flowchart of the Abstracter's equivalent sequence state space search algorithm.

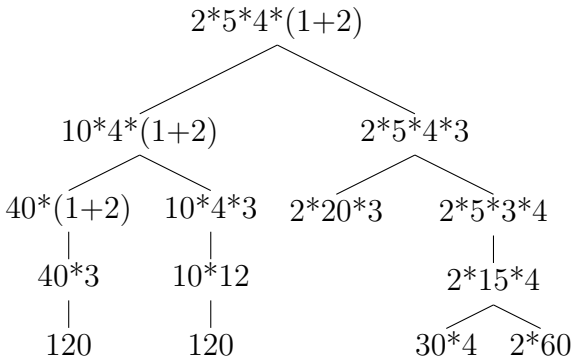


Figure 4.5: The nodes visited by the Abstracter's state space search when trying to reduce "2*5*4*(1+2)" with a maximum depth of 4. The lines represents the successful application of transformation rules.

4.5 Solver

The Solver does two things for the model: First it takes a sequence and potential goals and tries to find the best action to take in order to most reliably get the highest amount of reinforcement reward for the long term. Secondly it stores information gained through interaction with the world for the other systems to use in some convenient data structures. This section will begin by covering how the Solver stores information and then proceeds to explain how it uses this information together with the Abstracter to make decisions.

The Solver has a multitude of data structures that serve different purposes. It is worth noting that while the model is a sequence decision-making system the tasks that we are evaluating it on are sequence prediction tasks. For these tasks much of the Solver is unnecessarily complex, but they are necessary for allowing the model to work in domains outside sequence prediction.

4.5.1 Data structures in the Solver

The Solver maintains six data structures: A transition table, a Q-table, a reward table, two radix-trees of rewarded sequences, and a set of all fed starting sequences.

The transition table is a map of symbols and actions to the probability distribution of what symbol will come next. An key in the map may be an action, a symbol, or both and the value is a distribution of which symbol will come next given that the key symbol was the last in the sequence and the action is performed. The table stores the singleton symbol or action keys in case information is needed about a pair of symbol and action that has not been encountered by the model. This is implemented by having the values be a 2-tuple of the number of stored occurrences of the key and another map from each possible next symbol to the number of times it has occurred. Given a optimal map implementation with a $O(1)$ hashing operation the

look-up speed of a the probability of a symbol occurring given a key should have an average time complexity of $O(1)$. Note that in sequence prediction each action leads to the corresponding symbol with probability 1.

The Q-table is a straightforward implementation of the table as part of the Q-learning algorithm described in subsection 2.2 of the theory chapter.

The reward table is a straightforward map from a key of a 2-tuple of sequence and action to the expected reward.

The two radix-trees of rewarded sequences contains all sequences that gave a reinforcement above some predefined number (0 if not otherwise stated). The trees also contain, for each sequence, which actions prompted this reward and what that reward was. The difference between the two radix-trees is that one contains the sequences stored by their prefixes and the other by their suffixes (i.e. stored forward and backwards). The implementation of these uses the next symbols as edges and each node contains a set of the rewarded actions for that sequence. If the set is empty it means that that node is just part of a not in itself rewarded sequence

The set of all starting sequences is a simple set containing all sequences that the model have received but not written itself.

4.5.2 How the Solver maintains its data structures

The transition table is maintained by remembering the previous last symbol and action. When the next sequence is given the number of encounters of that symbol is incremented by one for the entries with keys corresponding to the previous action, the previous last symbol, and the two of them together. The number of accesses to those entries are also incremented.

The Q-table is maintained according to the description in subsection 2.2.

New entries are added to the radix-trees whenever the model gets a reward higher than the predefined threshold. The previous sequence (the current sequence with the last symbol removed) is then inserted to the radix-trees. For the forward facing tree (where sequences are stored from the first symbol to the last) a new sequence is inserted as follows. Starting at the root the insert algorithm follows the edges corresponding to each symbol. Whenever an edge does not exist a new empty node is created, and connected to the current node via an edge corresponding to that symbol. When the algorithm arrives at the node after the edge corresponding to the last symbol in the previous sequence it inserts the action and reward into that node's set. That is the given reward and previous action is stored in that node. The same is done for the reverse radix-tree, only the previous sequence is reversed first.

Whenever a sequence is given and the model have performed no previous action that

sequence is stored in the set of input sequences.

4.5.3 Decision-making by the Solver

The Solver is the subsystem that is ultimately to take what the other subsystems have produced in order to return the best action for the given situation. It takes a sequence and list of potential goals from the Abstracter and performs state space search over the possible actions for the potential goals. The flowchart for the Solver's state space search can be found in Figure 4.6

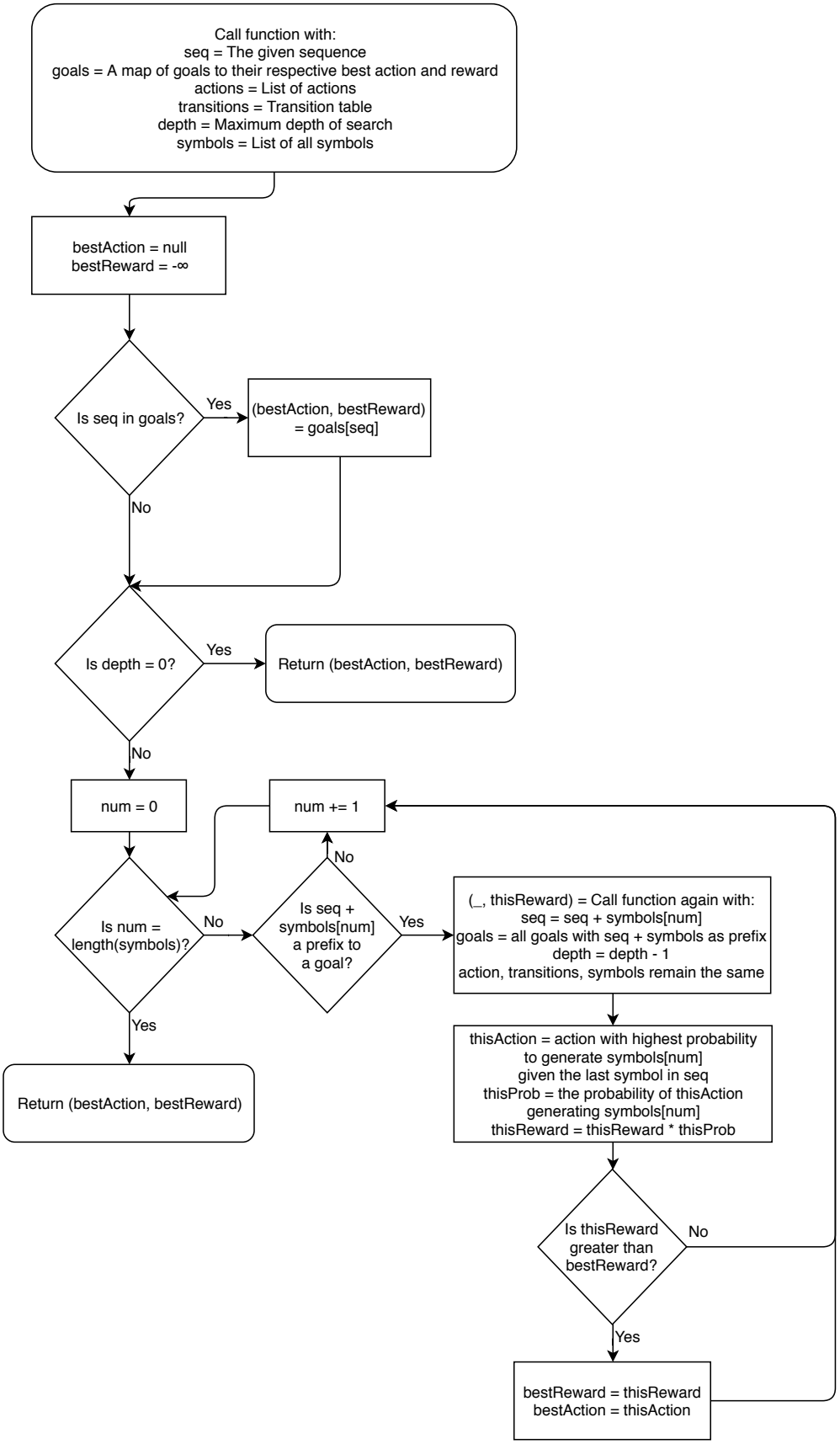


Figure 4.6: Flowchart of the Solver's state space search for already determined goal states.

For application on sequence prediction the model has only real use of the radix-trees and the set. However when it comes to sequence decision-making the extra data structures are necessary. In sequence prediction each action is guaranteed to produce a specific symbol and the sequence of actions is directly correlated to the reward. In sequence decision-making the reward and outcome of an action is based on the state of the world, which the agent may not have access to in its entirety. By storing the different states of the world in a sequence an agent can remember its structure. In that case, sequence decision-making reward is on a scale and may be received continuously throughout one run. Actions are not guaranteed to cause a next specific state which make it difficult to predict what to do next. In this case, the model is dependent on its extra data structures as well as on another version of the the state space search found in Figure 4.6. By having state space search for the highest discounted reward instead of some defined goal states the model can make use of the generality of Q-learning to tackle sequence decision-making. This extended search can be found in Figure 4.7. Note that this algorithm can still utilize the abstract goals created by the Rule-Former. In this more general setting an abstract goal represents a highly rewarded but not necessarily optimal sequence as it did in the previous case.

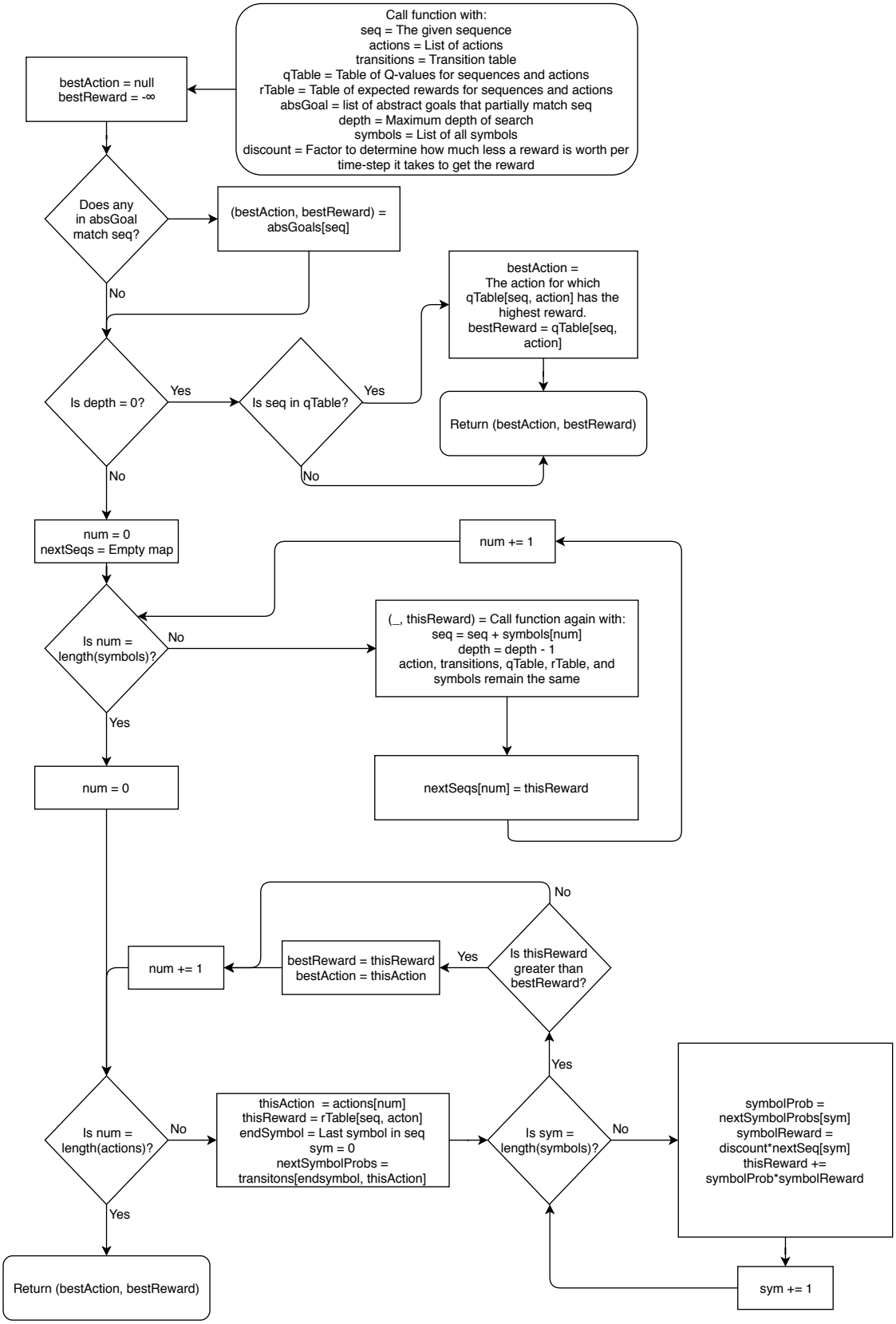


Figure 4.7: Flowchart of the Solver's state space search using Q-learning to solve problems without explicit goal states.

4.6 Rule-Former

The Rule-Former is the subsystem which task it is to create and remove the transformation rules and abstract goals that the Abstracter uses. Since the Rule-Former is a rule-based learning system we will, for clarity, refer to the algorithms the Rule-Former uses to create or remove transformation rules and abstract goals as learning-rules. As is often the case in rule-based learning the learning-rules can be widely different. The Rule-Former's purpose is to contain these learning-rules, to run the learning-rules at appropriate times, and to accept or reject the transformation rules and abstract goals created by the learning-rules.

A learning-rule is any algorithm that creates or removes transformation rules or abstract sequences from the model. Learning-rules may only use the data collected by the model to do this and should be as general as possible (i.e. avoid being domain specific). The iOur system is called "the Model"
Contents:
Sequence Learning
Evaluation
How the Model works
Results and Conclusionntended use of the Rule-Former is for it to have a multitude of learning-rules to cover many areas. The implementation of the model made for this thesis, however, only has one. This one learning-rule only produces transformation rules which means that the current implementation of the model cannot create abstract goals.

The Rule-Former should decide when to run learning-rules. When different learning-rules should run varies and a general implementation of this behavior is therefore not included in this thesis. Rather the task of deciding when to run a given learning-rule is delegated to the learning-rule itself.

4.6.1 Testing transformation rules and abstract goals

All transformation rules and abstract sequences created by the Rule-Former should be tested. The tests for transformation rules and abstract sequences are similar. If a rule fails the test it is removed, if it passes it is up to the learning-rule that created them to decide if they should be added to the list of rules or discarded.

The way the transformation rules are tested is that they are used to transform all known starting sequences (as stored in the Solver's set of starting sequences). If any resulting transformed sequence does not have the same expected output as the original the transformation rule failed the test. If the transformation rule passed the test the number of times it successfully managed to match a sequence whose transformed counterpart had the same expected output is given to the learning-rule that created it. The learning-rule may now decide whether to add this transformation rule to the list of transformation rules or discard it. The flowchart for the

transformation rule testing algorithm can be found in Figure 4.8.

A similar test is used for abstract goals. The abstract goal is tested by matching against all starting sequences.
For each starting sequence, if the abstract sequence partially matches that sequence the abstract sequence is tested against that sequence. A sequence that matches the abstract sequence and has the test sequence as start is calculated. If no or more than one such sequence can be calculated this test sequence is discarded and the test continues. Otherwise the calculated sequence is compared to the expected reward of that starting sequence. If they are not equal the rule fails the test. If it passes all such test sequences the test is passed as well and the number of test sequences that it passed is given to the learning-rule that created it. The learning-rule may now decide whether to add this abstract goal to the list of abstract goals or discard it. Since the current implementation does not use abstract goals this test was not implemented.

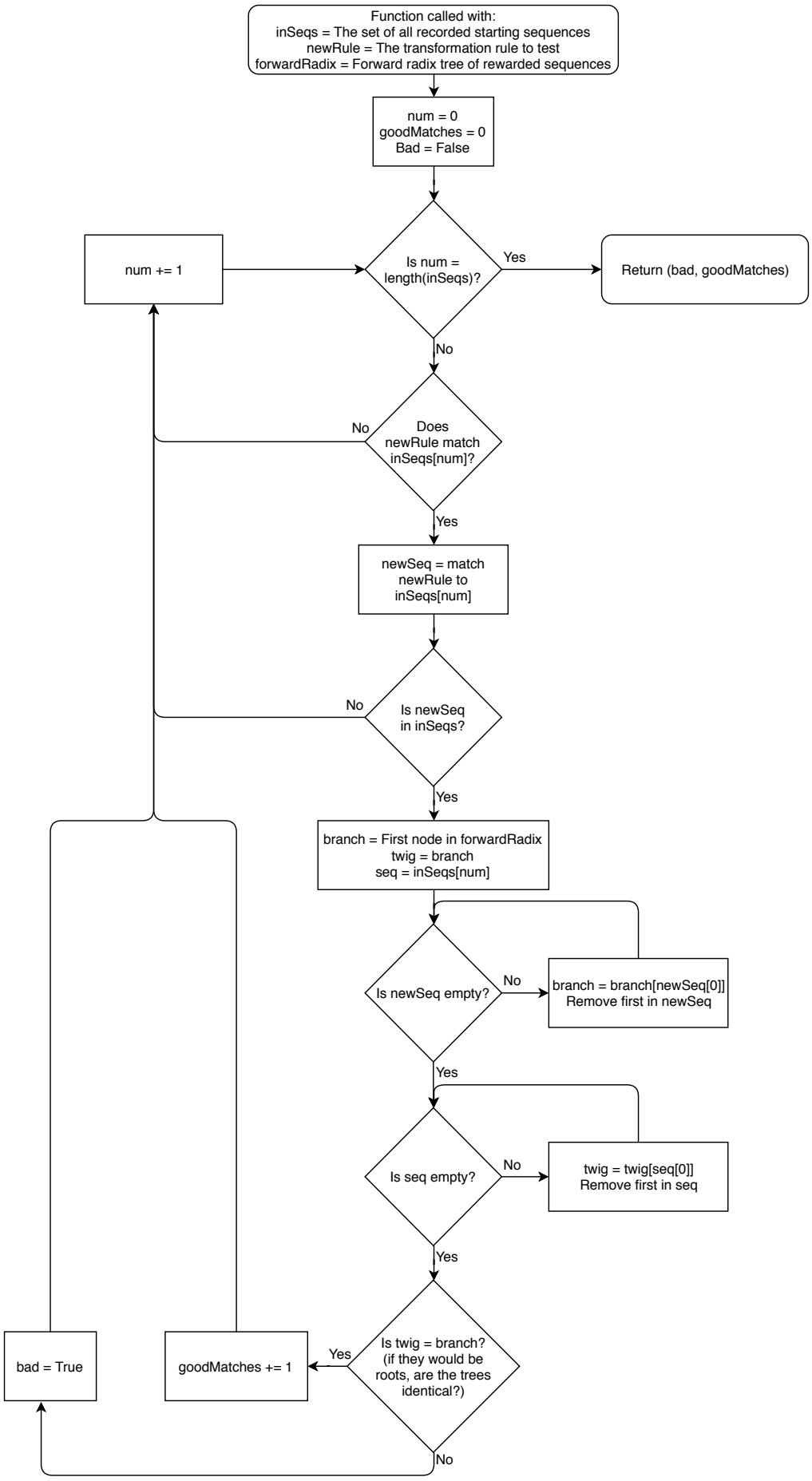


Figure 4.8: Flowchart of the Rule-Former's transformation rule testing algorithm.

4.6.2 The equivalence learning-rule

The equivalence learning-rule is the only learning-rule implemented in the system. Its purpose is to use the definition of equivalent sequences from subsection 2.1.4 to find subsequences that can be considered equivalent.

For example given the sequences "1 + 1 + 1 = 3" and "2 + 1 = 3" it may find that the reward for both sequences is positive and that both sequences share the ending "+1 = 3". Because of this it would seem that the sequences "1 + 1" and "2" can be considered equivalent. The equivalence learning-rule will upon the discovery of such an equivalence create the transformation rule that states that "1 + 1" can be replaced by "2". The new rule will then be tested using the algorithm in Figure 4.8. For example the input sequence "1+11 =" has the rewarded following sequence "12" while "21 =" (the result of transformation by the new rule) does not. If the new transformation rule is considered bad, the new rule is extended by adding one character to the end of the two starting sequences. In this example "1 + 1" becomes "1 + 1+" and "2" becomes "2+". The procedure of testing the rule against starting sequences is then repeated. The transformation rule is kept if it is not bad and have more than n matches among starting sequences. For this thesis $n = 3$.

This algorithm consists of two parts. First, given a rewarded sequence, find the rewarded sequences that share the largest part of their ending with the given sequence. This function also returns the index of the first sequence at which that ending begins. Second, forming the transformation rules and testing them. The second function returns a list of new transformation rules that the Rule-Former then appends to its list of transformation rules. Flowcharts describing the two parts can be found in Figure 4.9 and Figure 4.10 respectively.

This learning-rule is called whenever the system encounters a rewarded sequence. That is, when the system gets a positive reward it checks whether the sequence it encountered have been previously stored as rewarded. If it has not the two algorithms described above (and in Figure 4.9 and Figure 4.10) are called in order with `seq` being the newly encountered sequence, `closestSequences` being the list returned by the first algorithm, and `suffix` is the shared ending as calculated by using the value `lastNum` returned by the first algorithm. All other arguments are data structures from the Solver or parameters of the model.

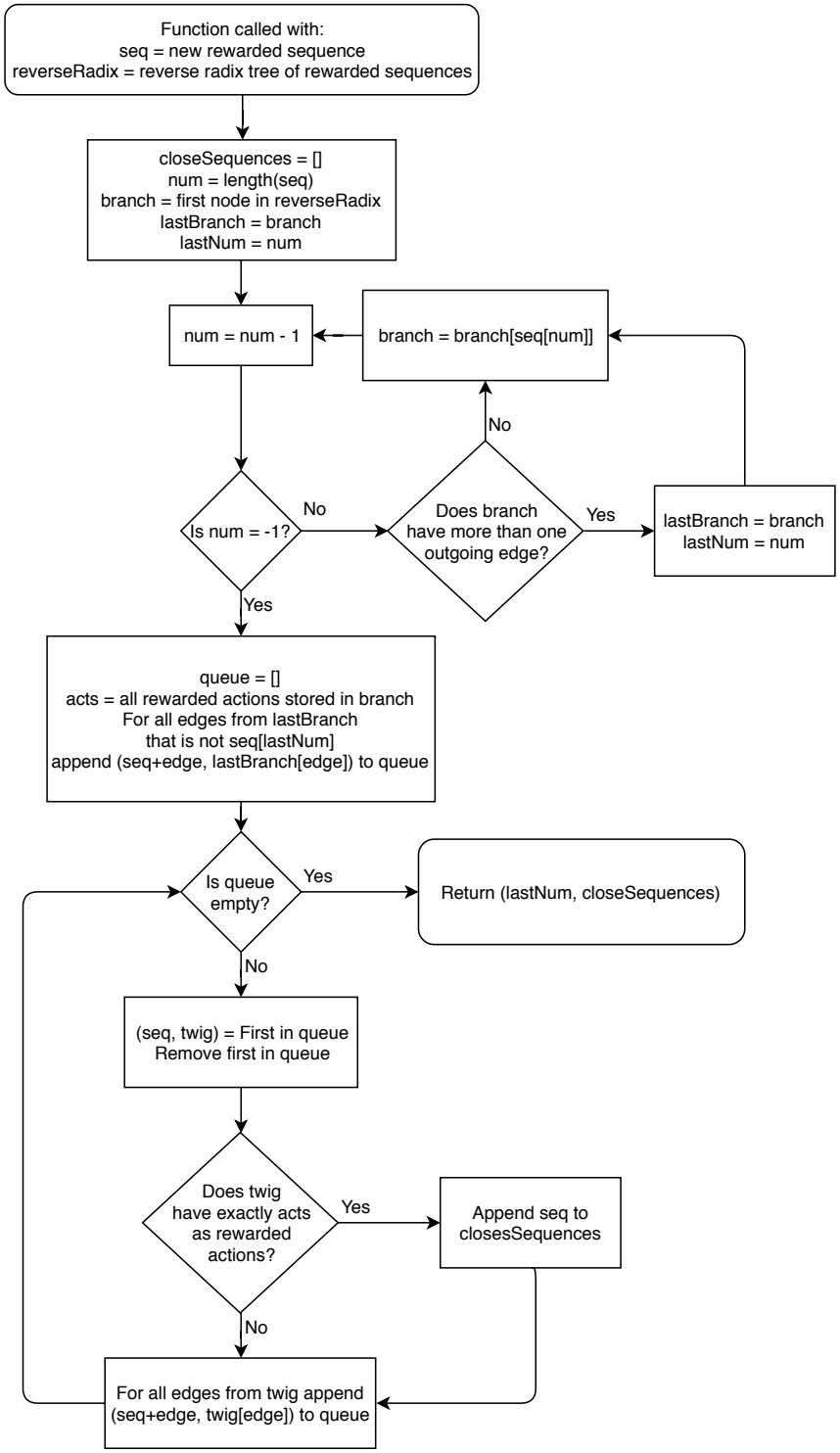


Figure 4.9: Flowchart of the Rule-Former's algorithm for locating rewarded sequences which share an ending with the given sequence.

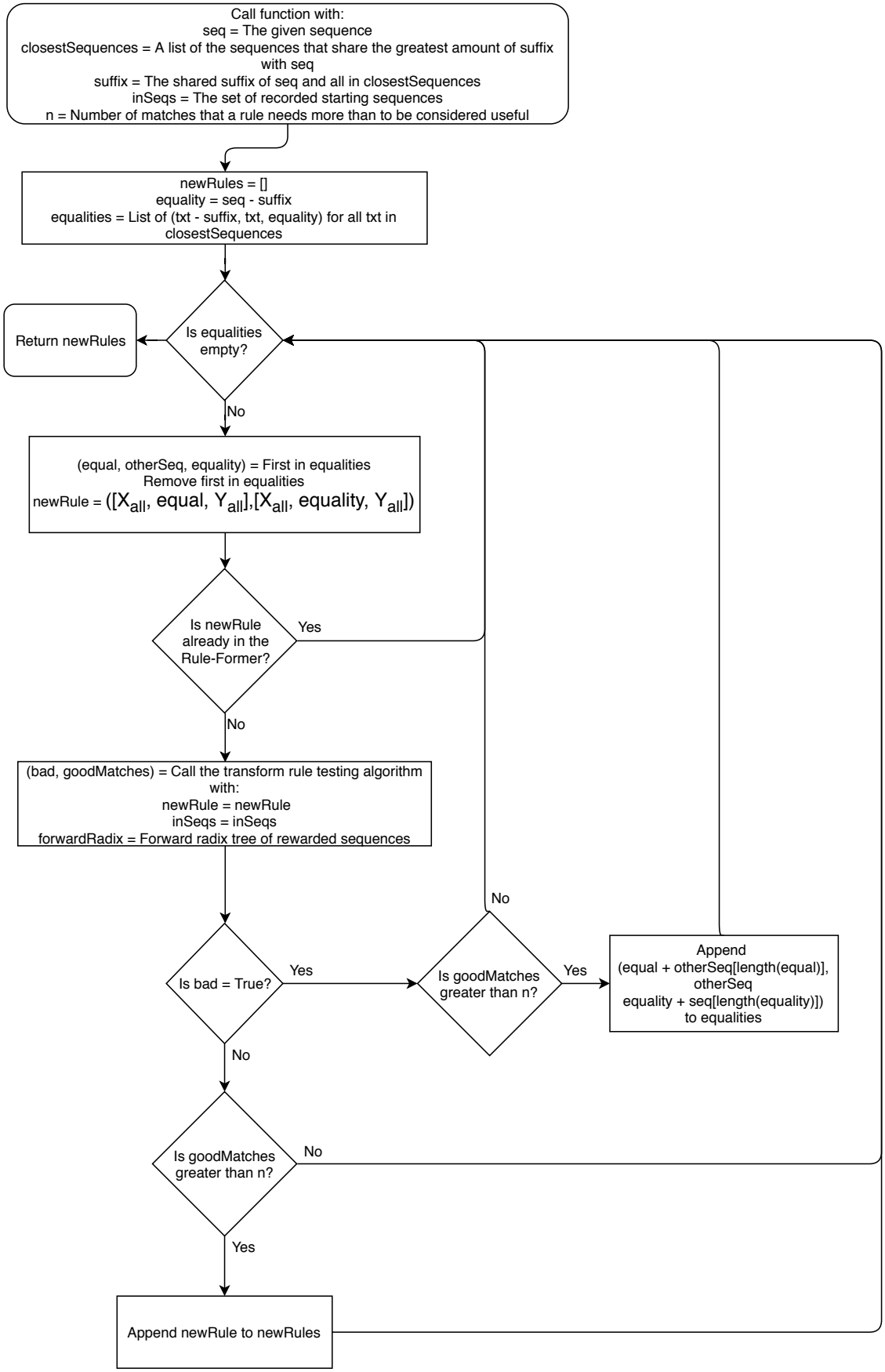


Figure 4.10: Flowchart of the Rule-Former's algorithm for creating new transformation rules based on sequence equivalence.

4.7 Training the model

The model was trained for five iteration. Each iteration ended with an evaluation to visualize how the model improves as the iterations progress. Each iteration consists of the model being fed the entirety of the training set, however what the model does with the training set varies between iterations.

The first iteration is used to teach the model which actions produce what symbols. The model is not given the input and told to do random actions. While the model might randomly produce a correct output sometimes during this iteration these are not used to learn anything other than that those results are good. This is because the Rule-Former is turned off for the first two iterations. Since model doesn't learn anything useful for the validation set the model simply does random actions throughout validation.

The second iteration the model is given the input along with the expected output and learns all the rewarded sequences and actions from the training set. The Rule-Former remains shut-off as it is best used when the model has gathered a lot of information already. Again the model doesn't have any information to use on the validation set and will perform random actions.

During the third to fifth iterations the Rule-Former is turned on and the model is still fed the input together with the expected output. Because the Rule-Former is turned on and the equivalence learning-rule runs when a rewarded sequence is encountered it will try to form new transform rules after every sequence.

5

LSTM Network Baseline

This section describes the LSTM network that was constructed in order to compare the results between the previously described model and this more established LSTM network. The LSTM network was constructed as a baseline for comparison and then trained on the same data.

The neural network was set up as a word prediction model where the initial sequence is first feed into the network while any outputs are ignored. And then it will start to feed its own output as the input until it considers the sequence finished.

5.1 Hyperparameters

When dealing with machine learning models there are many parameters that need to be pre-determined by the developers. Such as but not limited to the amount of layers in a neural network, the number of hidden units in each layer, and the learning rate of the learning algorithm. These Parameters differ from those that the model decides on by itself through training, and are therefore for clarity called Hyperparameters.

A number of different configurations were tested when determining the hyperparameter configuration for this model, but only one configuration was used in the end for the baseline comparisons. Due to both time contraints (the LSTM network was not the primary focus of the project since it brings nothing new to the field) and also due to hardware constraints, it is most likely the case that the LSTM network's hyperparameters can be further optimized in order to produce even better results in the future.

The procedure by which the architecture and hyperparamters was designed was by simple trial and error. For simplicity a fully connected multilayer neural network with LSTM-units with an output layer of simple perceptrons was chosen. The number of layers, units per layer, learning rate, and optimizing scheme was altered until the best settings for which the network could train on 300000 sequences in a relatively short time (less than 4 hours). The following different parameters were tested:

- 2 or 3 layers.
- Learning rates of 0.01 or 0.001.
- 128, 256, 512, or 1024 neurons per layer.

- Optimization with standard stochastic gradient descent or Adam.

Because of the length of the sequences (both input and expected output) the network was unrolled for training over 10 iterations. The input to the network is given as a vector of integers where each integer is mapped to a specific symbol. The vector represents the sequence and the networks first layer is fed each integer one after the other each timestep. After the last integer have been given the output vector can be read. The output vector has length equal to the number of symbols and each value represent how much the network believes that this symbol should be the next one. The next symbol is extracted by returning the symbol corresponding to the output neuron with the highest value. When a sequence was shorter than 10 symbols the start of the sequence was padded with a dummy integer until the sequence was 10 symbols long.

The following were used for training and evaluation:

- 3 Layers of Hidden Units.
- 512 Hidden Units on each layer.
- LSTM Neurons.
- The tanh (hyperbolic tangent) activation function in each LSTM unit.
- An output layer of perceptrons for one-hot encoding.
- A initial learning Rate of 0.001.
- Optimization using stochastic gradient descent with an Adam optimizer.
- 10 iterations of unrolling.

5.2 Implementation

The LSTM network was implemented in python using the TensorFlow machine learning framework [1]. The network used TensorFlow's "BasicLSTMCell", "MultiRNNCell", and "AdamOptimizer" implementations of LSTM-units, multilayer RNN structure and the Adam optimizer respectively. The network was designed to be trained The network was trained on a Nvidia gtx 1070 GPU using the NVIDIA CUDA toolkit as well as NVIDIA CUDA Deep Neural Network library (cuDNN). The basic training sessions were limited to take less than 12 hours each.

5.3 Training the LSTM network

The LSTM network was trained for 300000 iterations. Each iteration a random entry in the training set is given to the network. It is asked to figure out its one-hot prediction for which symbol should come next. The one-hot prediction is then compared to a vector with zeroes on each entry except the correct symbol which is represented by a one. The two vectors are used to calculate the loss of the network that is then backpropagated through the network using the Adam optimizer in order to train it. If the correct symbol had the highest one-hot value the training on that entry continues and the network is given the same sequence with the outputted

symbol appended to end and the process is repeated. This process is repeated until a wrong symbol is outputted, which is counted as incorrect, or all of the expected output has been outputted, which is counted as correct. The reason why the network is only continually trained if it correctly outputs the next symbol is to prevent it from forming an early bias toward the end-of-sequence symbol (ϵ).

During training the LSTM network is evaluated on the validation set every 10000 iterations.

6
Results

This chapter presents the model's performance in the different domains and compares it to that of the LSTM-network. For each domain several results using different settings will be presented; first that of the model followed by that of the LSTM-network. At the end of the chapter a summary table of the final performance of the two in each domain and setting can be seen for comprehension.

The results from the model and the LSTM network are presented in similar ways: As tables over the highest average fraction of the validation set each system correctly solved for some iteration. Recall from section 3.2 that iteration for the two systems means different things. A table of the size of training sets and validation sets as well as a table of the number of runs over which these results have been averaged have also been provided. These tables also contain the average correctness rate of a random sequence generator that will always end with the first end-of-sequence symbol and will, like the two systems, automatically output the end-of-sequence symbol as the xth output (x is the maximum length of any expected output for a given domain).

Note that all written numbers in this chapter have been rounded to three decimal points for readability. Graphs of the averages accuracy over training iterations for the model and the network can be found in Appendix B.

6.1 Results from the domain of simple arithmetic

This section presents the results from the datasets Arithmetic1 and Arthmetic2.

6.1.1 Results from Arithmetic1

Fraction as validation	0.1	0.5	0.9
Accuracy of the model	0.761	0.685	0.060
Accuracy of the network	0.578	0.293	0.125
Accuracy of random output	0.003	0.003	0.003

Table 6.1: The highest average accuracy for both systems on the validation set for Arithmetic1.

Fraction as validation	0.1	0.5	0.9
Runs of the model	20	20	20
Runs of the network	12	8	11
Runs of random output	200	200	200

Table 6.2: The number of runs over which the results have been averaged for Arithmetic1.

Fraction as validation	0.1	0.5	0.9
Size of training set	1260	700	140
Size of validation set	140	700	1260

Table 6.3: The final average accuracy for both systems on the validation set for Arithmetic1.

6.1.2 Results from Arithmetic2

Fraction as validation	0.1	0.5	0.9
Accuracy of the model	0.593	0.507	0.141
Accuracy of the network	0.618	0.366	0.087
Accuracy of random output	0.002	0.003	0.003

Table 6.4: The highest average accuracy for both systems on the validation set for Arithmetic1.

Fraction as validation	0.1	0.5	0.9
Runs of the model	20	20	20
Runs of the network	4	9	3
Runs of random output	200	200	200

Table 6.5: The number of runs over which the results have been averaged for Arithmetic2.

Fraction as validation	0.1	0.5	0.9
Size of training set	4050	2250	450
Size of validation set	450	2250	4050

Table 6.6: Size of training and validation sets for Arithmetic2.

6.2 Results From the domain of logic

Fraction as validation	0.1	0.5	0.9
Accuracy of the model	0.688	0.758	0.535
Accuracy of the network	1.000	0.905	0.710
Accuracy of random output	0.338	0.328	0.334

Table 6.7: The highest average accuracy for both systems on the validation set for Logic1.

Fraction as validation	0.1	0.5	0.9
Runs of the model	20	20	20
Runs of the network	4	2	4
Runs of random output	200	200	200

Table 6.8: The number of runs over which the results have been averaged for Logic1.

Fraction as validation	0.1	0.5	0.9
Size of training set	153	85	17
Size of validation set	17	85	153

Table 6.9: Size of training and validation sets for Logic1.

6.3 Results From the Domain of Simple English Grammar

Fraction as validation	0.1	0.5	0.9
Accuracy of the model	0.992	0.953	0.401
Accuracy of the network	1.000	1.000	1.000
Accuracy of random output	0.335	0.335	0.333

Table 6.10: The highest average accuracy for both systems on the validation set for Grammar1.

Fraction as validation	0.1	0.5	0.9
Runs of the model	20	20	20
Runs of the network	3	5	2
Runs of random output	200	200	200

Table 6.11: The number of runs over which the results have been averaged for Grammar1.

Fraction as validation	0.1	0.5	0.9
Size of training set	636	453	70
Size of validation set	71	454	637

Table 6.12: Size of training and validation sets for Logic1.

6.4 Result summary

Dataset	Fraction as validation	The model	The network	Random output
Arithmetic1	0.1	0.761	0.578	0.003
Arithmetic1	0.5	0.685	0.293	0.003
Arithmetic1	0.9	0.060	0.125	0.003
Arithmetic2	0.1	0.593	0.618	0.002
Arithmetic2	0.5	0.507	0.366	0.003
Arithmetic2	0.9	0.141	0.087	0.003
Logic1	0.1	0.688	1.000	0.338
Logic1	0.5	0.758	0.905	0.328
Logic1	0.9	0.535	0.710	0.334
Grammar1	0.1	0.992	1.000	0.335
Grammar1	0.5	0.953	1.000	0.335
Grammar1	0.9	0.401	1.000	0.333

Table 6.13: Summary of the highest average accuracy for for both systems for each dataset and fraction as validation.

As can be clearly seen in Appendix B there is often a large difference in accuracy between the best and the worst performing runs of both the network and the model. This is mainly due to how the dataset gets divided into the training and validation fractions. Both machine learning systems can produce highly reproducible results when given the exact same subsets repeatedly. The big randomness factor that change the results is what data points that get included in each of the two subsets before training even begins. This is why all runs were performed multiple times and the average results were used here.

7
Discussion

This chapter will begin with answering the research question, commenting on the performance on the systems, and argue why they have the performance they do on the different datasets. Following from there the limits of the domains and the systems will be discussed together with the validity of the evaluation. Then comes a section that touches on the potential errors in the theory or implementations that might affect the results, as well as the steps taken in order to try to prevent these. Finally the chapter will discuss potential uses for the model, improvements that might be made to it, and how to make use of it in more general domains.

7.1 Answering the research question

The research question presented was:

Can a rule-based learning system outperform an LSTM network on sequence learning problems?

From the results presented in the previous chapter it would seem that the answer is "Yes, there exists circumstances where a rule-base system can perform better than a LSTM network". Though this is not such an informative answer as it is easy to think up scenarios in which a rule-base system can outperform a neural network. If the network has poor architecture or hyperparameters it can be easily beaten. If the rule-based system has been specifically designed for the domain it is tested on.

A better answer would probably be: "Yes, the model outperformed the network presented in the LSTM network baseline chapter by more than 10 percentage points in three of the evaluations presented in the Datasets and Evaluation chapter."

Now the question that remains is whether or not this answer is informative. The following six sections dissect the results and tries to answer why the model outperformed the network, if the model is useful outside the presented domains, if the network is representative of LSTM networks in general, and if the results from these domains are representative of other common sequence learning domains.

To help answer these question Table 7.1 contains a summary of which domains the model performed significantly better (>0.1 better), the network performed better (>0.1 better), or they had similar performance.

Dataset\Fraction as validation	0.1	0.5	0.9
Arithmetic1	M	M	-
Arithmetic2	-	M	-
Logic1	N	N	N
Grammar1	-	-	N

Table 7.1: Summary of whether the model (M) or the network (N) had better performance, or if their accuracy where within 10 percentage points of one another (-).

7.2 When and why is the model better than the network?

For Arithmetic1 the model was clearly better than the network with fractions as validation of 0.1 and 0.5. For fraction as validation 0.9 the network performed better, though both systems performed poorly. This domain is set apart from the others in a few different ways. Before getting into that we need to analyze the other domains.

First of all both Logic1 and Grammar1 are one-step sequence-prediction problems. This means that while the systems only have to guess one symbol per sequence. This is good for the network because the network will have to learn which symbol should come next for each symbol in the expected output. This means that the shorter the expected output the less the network will have to learn. For the model on the other hand the length of expected output does not matter. If the model is given a sequence it has not encountered before it will attempt to reduce that sequence to a known equivalent sequence. If it can do this then it can simply output the expected output of that equivalent sequence. This means that for datasets with shorter expected output the model will probably perform worse relative to the network than on datasets with long expected output.

While both Arithmetic1 and Arithmetic2 have longer expected output Arithmetic2 also have mixed operators. In both datasets the model can use the associative and transitive properties of multiplication to form transformation rules. To know that "3*5" can be exchanged for "5*3" can help the model find solutions for any problems containing "3*5" that it has not encountered but which transformed counterpart it has. However, in Arithmetic2 the additional operator addition is difficult for the model to handle. While "5*3" may be replaced by "3*5" the same is not true for "5+3" and "3+5" due to the order of operations. This makes it much more difficult for the model to form useful transformation rules to handle addition. The model still forms some useful rules like "5+3+" can be exchanged for "3+5+" but these are only applicable when both operators are addition and "5" and "3" must be the first two operands. This means that for the entries with two operands (which is almost 89% of the dataset) the model have a good chance of solving the half of the entries with the two operands being the same and a poor chance of solving the problems when the two operands are different.

Simply put we believe that the model was indifferent to some of the problems that the network had to deal with when it comes to having longer expected output sequences. Additionally the model have difficulty handling datasets with many counter-examples to simple transformation rules which is why it only significantly outperforms the network on one of the 3 Arithmetic2 evaluations.

7.3 Is the LSTM network a good baseline?

It has already been mentioned why a LSTM network is a good baseline for sequence learning systems, given its great performance in the area. This section discusses whether or not the performance of the LSTM presented in this thesis is representative of the performance of general LSTM Networks.

The network created in this thesis was neither structured after a network from another thesis, nor does this thesis provide much backing as to why the networks architecture and hyperparameters are what they are.

7.3.1 Hyperparameter optimization

Many of the results presented in this thesis are based on limited testing and optimization. Both the model, but also the LSTM baseline model have a very large number of potential configurations, and not all of them could be compared to each other in a feasible timeframe. The aim was to show the potential on the new model and the extensions it can provide to an animat model and not to compare the results to the baseline in to much detail.

Before further work gets done more optimization could be useful in order to improve the results further. Another option could be to introduce some form of automatic optimization algorithm to the model, similar to the optimizers that already exist for some of the hyperparameters relating to Neural Networks [22].

7.3.2 Interpreting the network's performance

It is obvious that the performance of the network is not comparable to the best general implementations and that it is not the best LSTM network implementation for these domains. This begs the question: What is the point of comparing the performance of the model with that of the network?

While the network is not representative of the best LSTM networks it is still a LSTM network with decent performance on these problems. By comparing on which datasets the network performs worse, as good as, and better than the model we can get an understanding of which datasets a better LSTM network is also likely to perform good or bad on. This also means that for any datasets where the model does not outperform the network by a large margin the model likely has performance equal to or worse than a better LSTM network would. Furthermore it is obvious

that when the network outperforms the model it is obvious that the model could not compete with an even better LSTM network.

7.3.3 Scalability of neural networks

Another important aspects of neural networks in comparison to the model or indeed most rule-based systems is the scalability of neural networks. If one wants better performance from a neural network it is often as simple as increasing its size, training it for longer, or both. We postulate that on any domain where the network was outperformed by the model, an increase in layers, neurons, training time, or all three would lead to the network outperforming the model. This is not the case with the model. As one can see in Appendix B the model's performance spikes after iteration three when it starts applying its one learning-rule. After this any increase in performance comes from randomness. The model cannot perform better unless more learning-rules are implemented, which would require more work.

7.4 Approximate answers

In some domains it is desirable for the incorrect answers to at least be close to the correct answers, for example in natural language a small grammar mistake might be fine if the sentence is still understandable. This is not included in the result section above. The different machine learning models presented in this thesis have different approaches to this problem.

The model has no ability to approximate a solution based on incomplete data. In the event that it encounters an unknown new state that the abstraction system couldn't simplify into a familiar sequence the model will proceed to output randomly. As one can see from the performance of the random output this has poor performance.

Comparatively the LSTM network will never take a truly random action, when training it received feedback after each symbol it output, and could therefor learn incomplete sequences. For example, the sequence $\boxed{7}\boxed{*}\boxed{7}\boxed{=}$ was included in the simple arithmetic dataset, and during one of the runs when that was not part of the training set the validation of that sequence was measured. The LSTM network was unable to consistently output the entire correct sequence ($\boxed{4}\boxed{9}\boxed{\epsilon}$) but it regularly produced various other sequences that all approximated the answer: $\boxed{4}\boxed{x}\boxed{\epsilon}$ where x was one of the other one digit natural number symbols, most often $\boxed{2}$ was observed.

In order to solve this for the animats regular Q-Learning is not the most suitable form of learning. And that part of the model would need to be either replaced or extended to avoid random exploration. More about this is discussed in section 7.11 below.

7.5 Breaking the model

The implementation of the model evaluated in this thesis has only one learning-rule. That rule uses equivalence of sequences to find interchangeable prefixes to form transformation rules around. These transformation rules are formed only if there are no counter example to them. This highlights two flaws in the current implementation of the model.

First the model requires the domain to contain equivalent sequences. Secondly the model requires the domain not to contain counter examples of the useful transformation rules.

7.5.1 Requiring equivalent sequences

The current implementation of the model requires the domain to contain some equivalent sequences to form any new transformation rules. Recall that equivalent sequences are sequences whose best following actions are the same. The model locates equivalent subsequences by finding sequences with the same suffix and reward, this means that any domain without sequences ending with the same suffix are impossible for the model to learn anything about.

Take, for example, the domain of single digit equlities:
$E = \{x\boxed{=}x\boxed{\epsilon}, \qquad \forall x \in \{\boxed{0}, ..., \boxed{9}\}\}$
Given all sequences in E except $\boxed{3}\boxed{=}\boxed{3}\boxed{\epsilon}$ and then asked what symbols should follow after $\boxed{3}\boxed{=}$ the model would produce a random output.

7.5.2 Requiring lack of counter examples

The model will only form a transformation rule if there are no counter examples. That is, if for any starting sequence the rule matches the transformed sequence does not have the same expected output then the transformation rule will not be formed. While this practice prevents the model from learning potentially harmful transformation rules, it also prevents it from learning transformation rules that would, on average, increase performance.

7.6 Limited resources

For this thesis all systems have been run on standard home computers. Because of this the training time of the two systems have been slow and the different evaluations have not been able to run in parallel. Due to this the training time for each evalutation had to be limited so to be completed over a one-night run.

The chosen LSTM-networks's hyperparameters were limited based on available training time, but also to not be needing more time than the model. In addition to this the use of TensorFlow allowed the network to train quickly on a GPU in addition

to the CPU while the model was limited to training only on a CPU. This is made it difficult to train on large datasets since these require more training for both the model and the network. Additionally the differences in training hardware makes any comparison in training time between the two systems pointless. This is why this thesis have rather focused on the performance in comparison to how large parts of the datasets the systems have been given for training.

7.7 Are the datasets suitable for evaluating the model?

In there are some obvious similarities with the datasets on which the model was tested on. All of them are deterministic, free of outliers, and free of measuring errors. All of them contain an abundance of sequences with the same expected output. All of them have entries in the region of 10^2 to 10^4, which is small in the realm of machine learning. Though all three domains are commonly occurring in machine learning, none of the datasets have been taken from previous works. Following is an analysis of how these facts affect the credibility of evaluation on these datasets.

It is common for machine learning datasets to contain some outliers and measuring errors. The model's performance on datasets containing these is a question left unanswered by this thesis. As has been dicussed the model cannot handle domains without equivalent sequences which could make it difficult for the model to handle outliers and errors. Because its performance is probably negatively affected by the occurrence of counter-examples this is another obstacle posed by errors in the training data. With this in mind it is probable that the model would have pretty poor performance on datasets containing outliers and errors. However, as this was not evaluated in this thesis this guess remains untested.

All of the datasets contain several occurances of sequences with the same expected output. As having the same expected output is almost the same as being equivalent this means that all domains were suitable for the model to work in. Now the question is if this is because datasets with entries having the same expected output is the norm, and the four datasets reflects this, or the datasets shows a bias towards the model. When it comes to classification tasks the point of the task is to find which entries have the same classification. In such cases it is expected of a dataset to contain multiple entries with the same expected output. However common sequence learning tasks such as generating long messages are very unlikely to have the same output due to the number of ways a message can be constructed. While there exists some bias among the datasets towards the model having good performance it is not an completely unfair test viewed from this angle. Again it would be interesting to have the model tested on some common domain where it is not necessarily the case that many entries have the same expected output.

The datasets are small in comparison to other datasets such as the Hutter Prize Wikipedia dataset, the IAM Online Handwriting Database [11], or Twitter100k [18].

The reason for this is that limited resources for this thesis made larger datasets unfeasible, as training on them would have taken too long. As neural networks performs much better when the amount of training data gets large the model would probably be outperformed by the network in such domains, given that they both get to train on all the training data.

Why none of the datasets were taken from previous works mostly comes down to two reasons. First, since most machine learning datasets are quite large the limited resources of this thesis prevented the use of such datasets. Second, we could not find and make use of suitable datasets in time to train on them and therefore used the ones we created ourselves. While using known datasets could have given the model previous work to compare to, the LSTM-network gets to fill that role for the datasets used in this thesis.

Finally the three domains which the datasets represent were chosen because of their common use. They are not domains for which LSTM-networks are known to perform better or worse than other sequence learning domains. They were chosen before the before the design of the model was completed and before the mentioned flaws were discovered. Any bias of datasets toward the model is therefore coincidental.

7.8 Potential errors

As with any measurements there might be errors in the results. It is not believed to be any but the possibility is impossible to rule out. There might be bugs in the code that have not been found that impact the results or the sample sizes over multiple training runs might not be sufficiently big. If any attempts to recreate the work are performed these should be kept in consideration when comparing the results.

The network was never directly compared to any other LSTM neural network implementations in frameworks other than TensorFlow either, but TensorFlow is widely regarded as reliable and with production quality code. It is highly likely that with further updates to the TensorFlow codebase better results can be achieved in the future even on the same hardware as well.

All neural network performance runs were repeated multiple times and the reported results are the averages of the different runs at discrete intervals (validation was performed every 10 000 iterations). This was an attempt to avoid stray runs with rare results (such as getting potentially stuck in any local minimums) and other non reproduceable results.

While minor errors may exist in either system it is highly unlikely that any major error made its way into the final implementation. If it had the performance of either would likely be abysmal. The fact that the systems can get close to perfect average on Grammar1 is indication that both systems are in good shape.

7.9 Extending animats and other machine learning systems

In order to use the model together with an existing animat system, such as the one presented by C. Strannegård et al. [37] some minor changes to the model may have to be done. First of all the model is a framework for learning-rules. The one that is implemented for this thesis has flaws that have already been described. More learning-rules may have to be added to allow the model to cover more domains. The evaluation of new transformation rules is also a problem that could be solved by using better statistics instead of simply debunking the transformation rules that have a single counter-example. Other problems can come from domains where the model have to collect data while running. Such changes to the model are described in section 7.10 and 7.11. The remainder of this section will discuss what potential advantages could come from extending a system with use of the model.

The primary advantage of the model is being able to reduce sequence to other sequences that are similar or have similar best actions. This ability helps counteract the huge number of possible sequences that even a simple domain can contain. This means that even with only a small number of observed sequences the model can help solve new sequences by fining the similar ones. By integrating the Solver to or replacing it by a new system the model can extend the new system by giving it insight as to which sequences have similar use. This can be compared to how AlphaGo uses neural networks to give a min-max system insight as to which moves are probably good and what their expected value is.

The animats presented by C. Strannegård et al. already have a subsystem in place to abstract the world in the form of its gate network. However if the animats were to be used in a specifically sequence learning environment it may be advantageous to replace the gate network with the model. Since the remaining part of the animats use a form of Q-learning from some abstracted information of the domain this part of the animat system is highly suitable for replacing the Solver.

7.10 Difference between sequence prediction and decision-Making

The domains that are the goal for this project all fall under the category of sequence prediction. Despite this the goal of the model is to be a more general sequence decision-making system. When solving problems of sequence prediction there are some short-cuts one can take that one cannot necessarily use on the more general problem.

The first example of this is that in a sequence decision-making problem actions does not necessarily correspond to going to the next state. To put it another way, each action does not always cause the exact same symbol to appear. The model is already

designed to handle this problem, by having its transition tables be probability based and by using Q-learning to learn what action gives the highest long term reward given a sequence.

Following the first example the effects of an action in sequence decision-making is not entirely determined of what states have been previously visited or what actions have been used. One cannot equate the sequences used for decision-making with only the sequence of previous states or only the sequence of previous actions. This problem can be side-stepped by having the input sequences be more than just that. Each symbol of the sequences might represent the state of the environment surrounding the agent, the internal state of the agent, and the action taken at the same time. This gives the model a much more detailed picture of the world. Since this could cause the world to have a large number of symbols which could significantly slow down learning. This is one place where the model could benefit from being integrated with the animat system presented by C. Strannegård et al. [37]. As the local-Q-learning handles states based on their attributes which allows it to handle states it have not encountered based on attributes it has.

Another problem that might occur is that the domain might not contain separable sequences with a defined start and end point. The sequences might be simply the states of the world that keeps updating forever. In this case the model can learn little to nothing of each sequence as the only information it gets is a single sequence that keeps growing indefinitely. Additionally this would require the model to store arbitrarily long sequences. A way to solve this is to set a max length of a sequence and remove the first symbol whenever a new one is introduced. The model should in this case treat each new sequence as a starting sequence in order to store each sequence in its set of input sequences for the Rule-Former to use.

When the world does not not have clearly defined boundaries of its sequences the reinforcement rewards might not come at exactly one given moment. Additionally in a more general domain the rewards might not be only: Neutral, positive, or negative. Rather it would be a scale from some lowest value to some highest and the goal of the model would be to optimize the sum of all rewards it can get, perhaps taking some discount factor into account. The fix for this is to extend the state space search in the Solver to take into account the rewards of intermediate actions and use Q-values to determine the value of a state rather than specifically stored goals. The Solver already have the data structures in place to deal with this in its reward table and Q-table.

7.11 Further improvements

In order to further improve the results a number of improvements and additions are suggested here. Most of these are improvements for the Abstracter and Rules-Former, that would help it be more efficient and improve the generalization capabilities.

7.11.1 Additional learning-rules

The most intuitive way to improve the model is through additional learning-rules. The model can be seen as a framework for these learning-rules and with a working model a few well thought out learning-rules can help cover up the flaws that is present in the current implementation. Following is a few suggestions for such rules.

The model cannot handle domains with entries whose expected output are not the same as any other entry's. This is one of the reasons why the concept of abstract goals was introduced. However, for this thesis, no learning-rule was created that could actually form these. A suggested starting point for such a learning-rule would be to look after patterns in the list of all rewarded sequences to see if there is any patterns that are rewarded. Abstract goals formed from this learning-rule could be tested to see that they do not match to any unrewarded sequences similar to the test used for transformation rules.

Association, transitivity, and commutativity are three common properties found in mathematical domains. These attributes could be worth looking for and creating transformation rules to match. For example in Arthmetic2 the following transformation rule corresponding to the commutative property of multiplication could have been useful:

$$([X_{all} O D_1 \boxed{*} D_2 \boxed{=} Y_{all}], [X_{all} O_1 D_2 \boxed{*} D_1 \boxed{=} Y_{all}])$$

Where D_k can be a subsequence of digits of any length and O can be either $\boxed{+}$ or $\boxed{*}$.

7.11.2 Finding general symbol-variable templates

A problem that can come up when creating more advanced learning-rules is the creation of the symbol-variables. To figure out groupings of symbols and the rules of how one may use these groupings is central to many sequence learning domains. In arithmetic and logic knowing the differences between equality, operators, and digits makes expressions much easier to break down into parts. The same goes for finding nouns, verbs, pronouns, etc. in the domain of English grammar or indeed any natural language domain.

To help with this the rules-former could look for symbols that have the same use and for templates for symbol-variables to use such that each new transformation rule or abstract goal does not need to find these distinctions anew. Furthermore these templates could be predefined to know if a type of symbol may be repeated or skipped.

A suggested way to achieve this could be to form a Markov chain of which symbols leads to which other symbols. By knowing this symbols could be grouped together with symbols that have connections with similar probabilities of as anther symbol.

7.11.3 Heuristic for unknown states

When Q-Learning encounters a completely new state it has no information about what actions to do, if that happens it explores by producing a random output in order to gain more information for the next time this state is encountered. This may produce undesirable results. One way to to improve upon this is to extend Q-Learning with some form of heuristic for new states. For example it could use the average of all other states, to determine the order in which to explore the actions to find their real Q-values. Or it could use local-Q-learning [37] which would allow it to recognize similar states and base the actions upon their Q-tables.

7.11.4 Adopting learning-rules from Alice in Wonderland

As previously stated Alice in Wonderland is a rule-based learning system that does similar things. Downside of Alice in Wonderland is that the system can only reduce sequences and needs to be given known equalities as input. The upside is that Alice in Wonderland have a great system for creating rules needing only a handful of examples. This is great considering that the model needs more ways to create good transformation rules and abstract goals and can form equalities in its current form. This should make it fairly straightforward to implement learning-rules based on Alice in Wonderland into the model.

8

Conclusion

While the model could outperform the LSTM network on some of the evaluations we postulate that this could be solved by increasing the size or training time of the network. We have also suggested some further improvements of the model and although these would probably be able to fix some of its flaws and increase the performance this would require more work. The network could probably achieve a similar increase in performance by simply increasing its size and training time. With this in mind we conclude that the type of rule-based system that the model represents is probably of little to no interest for the field of machine learning.

That being said we are impressed by the performance that the simple equivalence learning-rule provides. The equivalence learning-rule together with the transformation rule testing algorithm, removed from the rest of the model, is something that could definitely be of interest for further development. Not in the least because the learning-rule could easily be adapted to work together with Alice in Wonderland to make that system work on reinforcement learning problems.

This thesis does not have enough breadth to draw any conclusions with regards to the general usefulness of rule-based systems in comparison to neural networks. However, like how state space search was used in conjunction with neural networks to form AlphaGo [35], rule-based systems could perhaps also be used in conjunction with neural networks or other modern machine learning techniques.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Andrew G Barto and Satinder Pal Singh. On the computational economics of reinforcement learning. In *Connectionist Models*, pages 35–44. Elsevier, 1991.

[3] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[4] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[5] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, March 2012.

[6] Yves Chauvin and David E Rumelhart. *Backpropagation: theory, architectures, and applications*. Psychology Press, 2013.

[7] Raymond Chiong. *Nature-inspired algorithms for optimisation*. Springer Verlag, 2009.

[8] Jan Chorowski and Navdeep Jaitly. Towards better decoding and language model integration in sequence to sequence models. *arXiv preprint arXiv:1612.02695*, 2016.

[9] Marco Dorigo and Mauro Birattari. *Ant Colony Optimization*, pages 36–39. Springer US, Boston, MA, 2010.

[10] F.A. Gers, J. Schmidhuber, and F. Cummins. Learning to forget: continual prediction with lstm. *IET Conference Proceedings*, pages 850–855(5), January 1999.

[11] Alex Graves. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013.

[12] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *Acoustics, speech and signal*

processing (icassp), 2013 ieee international conference on, pages 6645–6649. IEEE, 2013.

[13] Sumit Gupta. Deep learning performance breakthrough - ibm it infrastructure blog, Jan 2018.

[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[15] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

[16] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[17] John J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79(8):2554–2558, 1982.

[18] Yuting Hu, Liang Zheng, Yi Yang, and Yongfeng Huang. Twitter100k: A real-world dataset for weakly supervised cross-media retrieval. *IEEE Transactions on Multimedia*, 2017.

[19] Guang-Bin Huang, Qin-Yu Zhu, and Chee-Kheong Siew. Real-time learning capability of neural networks. *IEEE Transactions on Neural Networks*, 17(4):863–878, July 2006.

[20] Łukasz Kaiser and Ilya Sutskever. Neural gpus learn algorithms. *arXiv preprint arXiv:1511.08228*, 2015.

[21] Nikhil Ketkar. Stochastic gradient descent. In *Deep Learning with Python*, pages 113–132. Springer, 2017.

[22] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[23] Sotiris B Kotsiantis, I Zaharakis, and P Pintelas. Supervised machine learning: A review of classification techniques. *Emerging artificial intelligence applications in computer engineering*, 160:3–24, 2007.

[24] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[25] Yann Lecun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[26] Kevin W Mickey and James L McClelland. A neural network model of learning mathematical equivalence. In *Proceedings of the Annual Meeting of the Cognitive Science Society*, volume 36, 2014.

[27] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[28] Abhishek Nandy and Manisha Biswas. Reinforcement learning basics. In *Reinforcement Learning*. Apress, Berkeley, CA, USA, 2018.

[29] Abdul Rahim Nizamani, Jonas Juel, Ulf Persson, and Claes Strannegård. *Bounded Cognitive Resources and Arbitrary Domains*, pages 166–176. Springer International Publishing, Cham, 2015.

[30] Thomas D Parsons, Albert A Rizzo, and J Galen Buckwalter. Backpropagation and regression: comparative utility for neuropsychologists. *Journal of Clinical and Experimental Neuropsychology*, 26(1):95–104, 2004.

[31] J. Pearl. *Heuristics: Intelligent search strategies for computer problem solving*. January 1984.

[32] VP Plagianakos and GD Magoulas. Stochastic gradient descent. *Advances in Convex Analysis and Global Optimization: Honoring the Memory of C. Caratheodory (1873–1950)*, 54:433, 2013.

[33] Geoffrey Sampson. Parallel distributed processing: Explorations in the microstructures of cognition, 1987.

[34] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, and et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, Jan 2016.

[35] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, and et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, Oct 2017.

[36] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[37] Claes Strannegård, Nils Svangård, David Lindström, Joscha Bach, and Bas Steunebrink. The animat path to artificial general intelligence. In *Workshop on Architectures for Generality and Autonomy (AGA 2017)*, 2017. Available at http://cadia.ru.is/workshops/aga2017/.

[38] R. Sun and C. L. Giles. Sequence learning: from recognition and prediction to sequential decision making. *IEEE Intelligent Systems*, 16(4):67–70, July 2001.

[39] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[40] Elio Tuci, Alexandros Giagkos, Myra Wilson, and John Hallam. *From Animals to Animats 14: 14th International Conference on Simulation of Adaptive Behavior, SAB 2016, Aberystwyth, UK, August 23-26, 2016, Proceedings*, volume 9825. Springer International Publishing, 01 2016.

[41] Ryan Urbanowicz and Will Browne. Introducing rule-based machine learning: A practical guide. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO Companion '15, pages 263–292, New York, NY, USA, 2015. ACM.

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

[43] Christopher John Cornish Hellaby Watkins. *Learning from delayed rewards*. PhD thesis, King's College, Cambridge, 1989.

[44] Barbara Webb. Animals versus animats: Or why not model the real iguana? *Adaptive Behavior*, 17(4):269–286, 2009.

[45] Sholom M Weiss and Nitin Indurkhya. Rule-based machine learning methods for functional prediction. *Journal of Artificial Intelligence Research*, 3:383–403, 1995.

[46] Ronald J Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural computation*, 1(2):270–280, 1989.

[47] Stewart W Wilson. Classifier systems and the animat problem. *Machine learning*, 2(3):199–228, 1987.

[48] S.W. Wilson. The animat path to ai. 1991.

[49] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[50] Weixiong Zhang. *State-space search: algorithms, complexity, extensions, and applications*. Springer, 1999.

A

Grammar1

This appendix contains the dataset Grammar1 which has been split into two sections: The first for grammatically correct sentences that has expected output `True` and the second for grammatically incorrect sentences that has expected output `False`.

A.1 Grammatically correct sentences

he talked
he walked
he remembered
he dreamed
he payed
he called
he played
she talked
she walked
she remembered
she dreamed
she payed
she called
she played
it talked
it walked
it remembered
it dreamed
it payed
it called
it played
they talked
they walked
they remembered
they dreamed
they payed
they called
they played
we talked

we walked
we remembered
we dreamed
we payed
we called
we played
I talked
I walked
I remembered
I dreamed
I payed
I called
I played
you talked
you walked
you remembered
you dreamed
you payed
you called
you played
he talks
he walks
he remembers
he dreams
he pays
he calls
he plays
she talks
she walks
she remembers
she dreams
she pays
she calls
she plays
it talks
it walks
it remembers
it dreams
it pays
it calls
it plays
they talk
they walk
they remember
they dream
they pay

they call
they play
we talk
we walk
we remember
we dream
we pay
we call
we play
I talk
I walk
I remember
I dream
I pay
I call
I play
you talk
you walk
you remember
you dream
you pay
you call
you play
he is fun
he is unpleasant
he is delightful
he is free
he is old
he is young
he is active
he is correct
he is wrong
she is fun
she is unpleasant
she is delightful
she is free
she is old
she is young
she is active
she is correct
she is wrong
it is fun
it is unpleasant
it is delightful
it is free
it is old

it is young
it is active
it is correct
it is wrong
they are fun
they are unpleasant
they are delightful
they are free
they are old
they are young
they are active
they are correct
they are wrong
we are fun
we are unpleasant
we are delightful
we are free
we are old
we are young
we are active
we are correct
we are wrong
I am fun
I am unpleasant
I am delightful
I am free
I am old
I am young
I am active
I am correct
I am wrong
you are fun
you are unpleasant
you are delightful
you are free
you are old
you are young
you are active
you are correct
you are wrong
he's fun
he's unpleasant
he's delightful
he's free
he's old
he's young

he's active
he's correct
he's wrong
she's fun
she's unpleasant
she's delightful
she's free
she's old
she's young
she's active
she's correct
she's wrong
it's fun
it's unpleasant
it's delightful
it's free
it's old
it's young
it's active
it's correct
it's wrong
they're fun
they're unpleasant
they're delightful
they're free
they're old
they're young
they're active
they're correct
they're wrong
we're fun
we're unpleasant
we're delightful
we're free
we're old
we're young
we're active
we're correct
we're wrong
I'm fun
I'm unpleasant
I'm delightful
I'm free
I'm old
I'm young
I'm active

I'm correct
I'm wrong
you're fun
you're unpleasant
you're delightful
you're free
you're old
you're young
you're active
you're correct
you're wrong
he is talking
he is walking
he is remembering
he is dreaming
he is paying
he is calling
he is playing
she is talking
she is walking
she is remembering
she is dreaming
she is paying
she is calling
she is playing
it is talking
it is walking
it is remembering
it is dreaming
it is paying
it is calling
it is playing
they are talking
they are walking
they are remembering
they are dreaming
they are paying
they are calling
they are playing
we are talking
we are walking
we are remembering
we are dreaming
we are paying
we are calling
we are playing

I am talking
I am walking
I am remembering
I am dreaming
I am paying
I am calling
I am playing
you are talking
you are walking
you are remembering
you are dreaming
you are paying
you are calling
you are playing
he's talking
he's walking
he's remembering
he's dreaming
he's paying
he's calling
he's playing
she's talking
she's walking
she's remembering
she's dreaming
she's paying
she's calling
she's playing
it's talking
it's walking
it's remembering
it's dreaming
it's paying
it's calling
it's playing
they're talking
they're walking
they're remembering
they're dreaming
they're paying
they're calling
they're playing
we're talking
we're walking
we're remembering
we're dreaming

we're paying
we're calling
we're playing
I'm talking
I'm walking
I'm remembering
I'm dreaming
I'm paying
I'm calling
I'm playing
you're talking
you're walking
you're remembering
you're dreaming
you're paying
you're calling
you're playing

A.2 Grammatically incorrect sentences

he fun
he unpleasant
he delightful
he free
he old
he young
he active
he correct
he wrong
she fun
she unpleasant
she delightful
she free
she old
she young
she active
she correct
she wrong
it fun
it unpleasant
it delightful
it free
it old
it young
it active

it correct
it wrong
they fun
they unpleasant
they delightful
they free
they old
they young
they active
they correct
they wrong
we fun
we unpleasant
we delightful
we free
we old
we young
we active
we correct
we wrong
I fun
I unpleasant
I delightful
I free
I old
I young
I active
I correct
I wrong
you fun
you unpleasant
you delightful
you free
you old
you young
you active
you correct
you wrong
he is talk
he is walk
he is remember
he is dream
he is pay
he is call
he is play
she is talk

she is walk
she is remember
she is dream
she is pay
she is call
she is play
it is talk
it is walk
it is remember
it is dream
it is pay
it is call
it is play
they are talk
they are walk
they are remember
they are dream
they are pay
they are call
they are play
we are talk
we are walk
we are remember
we are dream
we are pay
we are call
we are play
I am talk
I am walk
I am remember
I am dream
I am pay
I am call
I am play
you are talk
you are walk
you are remember
you are dream
you are pay
you are call
you are play
he's talk
he's walk
he's remember
he's dream
he's pay

he's call
he's play
she's talk
she's walk
she's remember
she's dream
she's pay
she's call
she's play
it's talk
it's walk
it's remember
it's dream
it's pay
it's call
it's play
they're talk
they're walk
they're remember
they're dream
they're pay
they're call
they're play
we're talk
we're walk
we're remember
we're dream
we're pay
we're call
we're play
I'm talk
I'm walk
I'm remember
I'm dream
I'm pay
I'm call
I'm play
you're talk
you're walk
you're remember
you're dream
you're pay
you're call
you're play
he is talked
he is walked

he is remembered
he is dreamed
he is payed
he is called
he is played
she is talked
she is walked
she is remembered
she is dreamed
she is payed
she is called
she is played
it is talked
it is walked
it is remembered
it is dreamed
it is payed
it is called
it is played
they are talked
they are walked
they are remembered
they are dreamed
they are payed
they are called
they are played
we are talked
we are walked
we are remembered
we are dreamed
we are payed
we are called
we are played
I am talked
I am walked
I am remembered
I am dreamed
I am payed
I am called
I am played
you are talked
you are walked
you are remembered
you are dreamed
you are payed
you are called

you are played
he's talked
he's walked
he's remembered
he's dreamed
he's payed
he's called
he's played
she's talked
she's walked
she's remembered
she's dreamed
she's payed
she's called
she's played
it's talked
it's walked
it's remembered
it's dreamed
it's payed
it's called
it's played
they're talked
they're walked
they're remembered
they're dreamed
they're payed
they're called
they're played
we're talked
we're walked
we're remembered
we're dreamed
we're payed
we're called
we're played
I'm talked
I'm walked
I'm remembered
I'm dreamed
I'm payed
I'm called
I'm played
you're talked
you're walked
you're remembered

you’re dreamed
you’re payed
you’re called
you’re played
fun he is
fun she is
fun it is
fun they are
fun we are
fun I am
fun you are
fun he’s
fun she’s
fun it’s
fun they’re
fun we’re
fun I’m
fun you’re
unpleasant he is
unpleasant she is
unpleasant it is
unpleasant they are
unpleasant we are
unpleasant I am
unpleasant you are
unpleasant he’s
unpleasant she’s
unpleasant it’s
unpleasant they’re
unpleasant we’re
unpleasant I’m
unpleasant you’re
delightful he is
delightful she is
delightful it is
delightful they are
delightful we are
delightful I am
delightful you are
delightful he’s
delightful she’s
delightful it’s
delightful they’re
delightful we’re
delightful I’m
delightful you’re

free he is
free she is
free it is
free they are
free we are
free I am
free you are
free he's
free she's
free it's
free they're
free we're
free I'm
free you're
old he is
old she is
old it is
old they are
old we are
old I am
old you are
old he's
old she's
old it's
old they're
old we're
old I'm
old you're
young he is
young she is
young it is
young they are
young we are
young I am
young you are
young he's
young she's
young it's
young they're
young we're
young I'm
young you're
active he is
active she is
active it is
active they are

active we are
active I am
active you are
active he's
active she's
active it's
active they're
active we're
active I'm
active you're
correct he is
correct she is
correct it is
correct they are
correct we are
correct I am
correct you are
correct he's
correct she's
correct it's
correct they're
correct we're
correct I'm
correct you're
wrong he is
wrong she is
wrong it is
wrong they are
wrong we are
wrong I am
wrong you are
wrong he's
wrong she's
wrong it's
wrong they're
wrong we're
wrong I'm
wrong you're

B

Result Graphs

This appendix contains graphs of the performance of the model and the LSTM network for all datasets and fractions as validation. The graphs contain the average, best, and worst accuracy on the validation set for each iteration the system has been trained. The average accuracy is in blue while the best and worst accuracies are in orange.

Note that the accuracy for the model is given in fractions, while it is given in percentage for the network.

B.1 Results from the model

B.1.1 Results from the domain of simple arithmetic

This section presents the evaluation accuracy graphs from the datasets Arithmetic1 and Arthmetic2.

B.1.1.1 Graphs from Arithmetic1

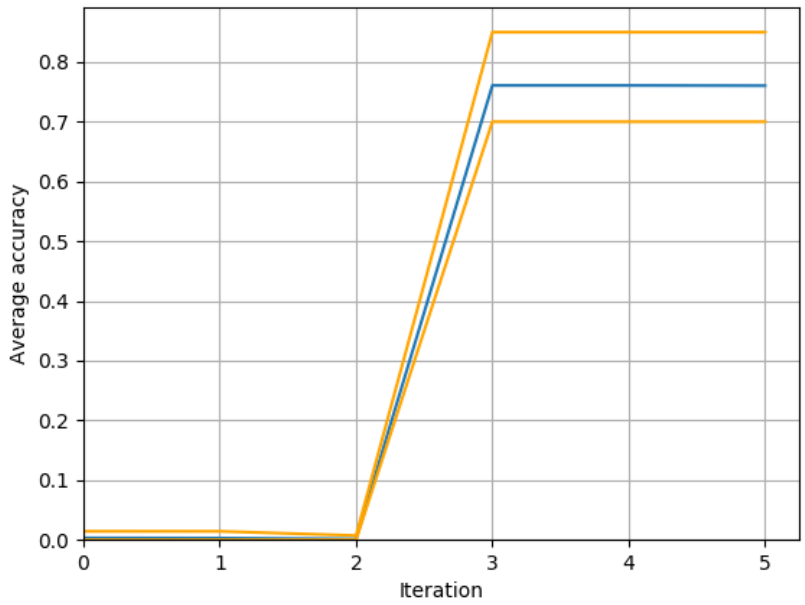


Figure B.1: The average (blue), best (orange), and worst (orange) accuracies for the model on Arithmetic1 with a fraction as validation of 0.1

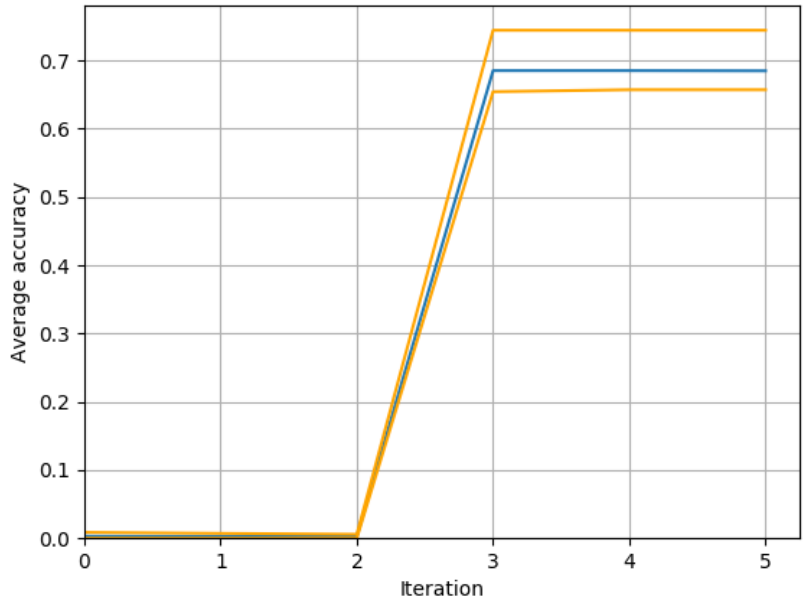


Figure B.2: The average (blue), best (orange), and worst (orange) accuracies for the model on Arithmetic1 with a fraction as validation of 0.5

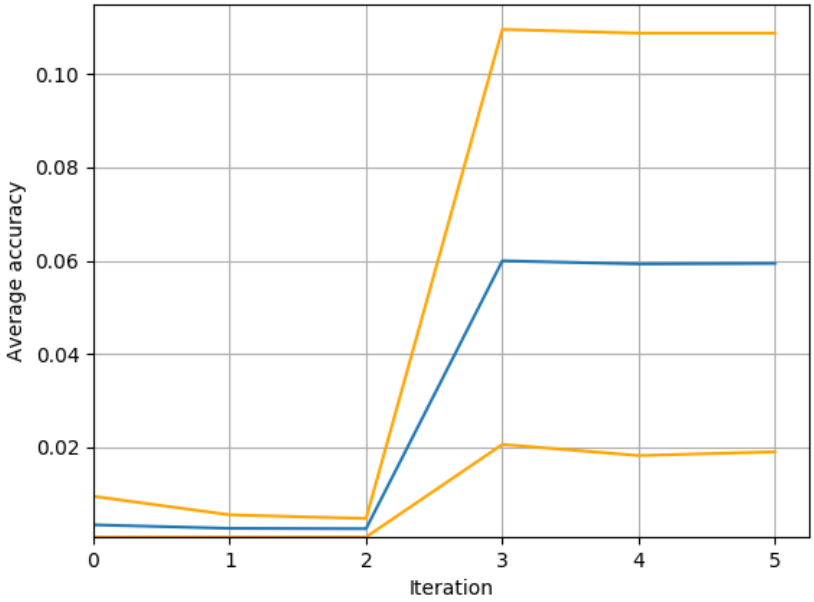


Figure B.3: The average (blue), best (orange), and worst (orange) accuracies for the model on Arithmetic1 with a fraction as validation of 0.9

B.1.1.2 Graphs from Arithmetic2

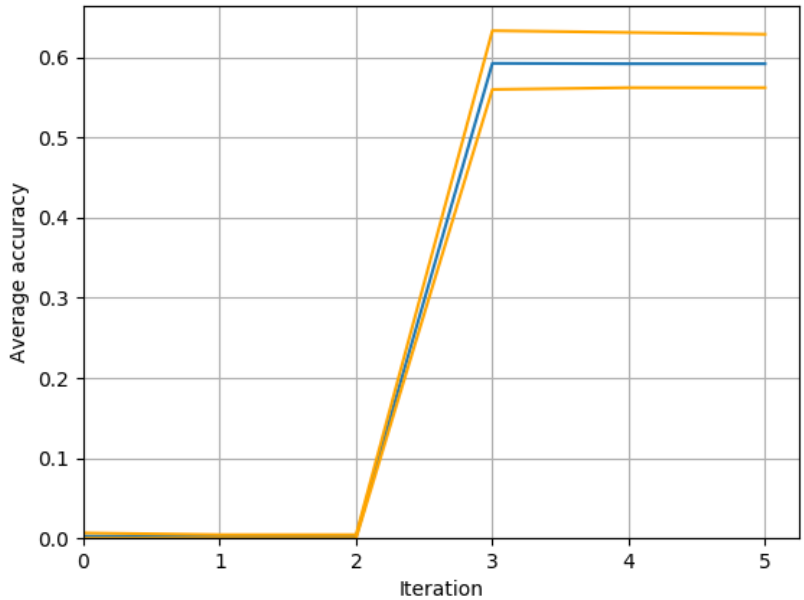


Figure B.4: The average (blue), best (orange), and worst (orange) accuracies for the model on Arithmetic2 with a fraction as validation of 0.1

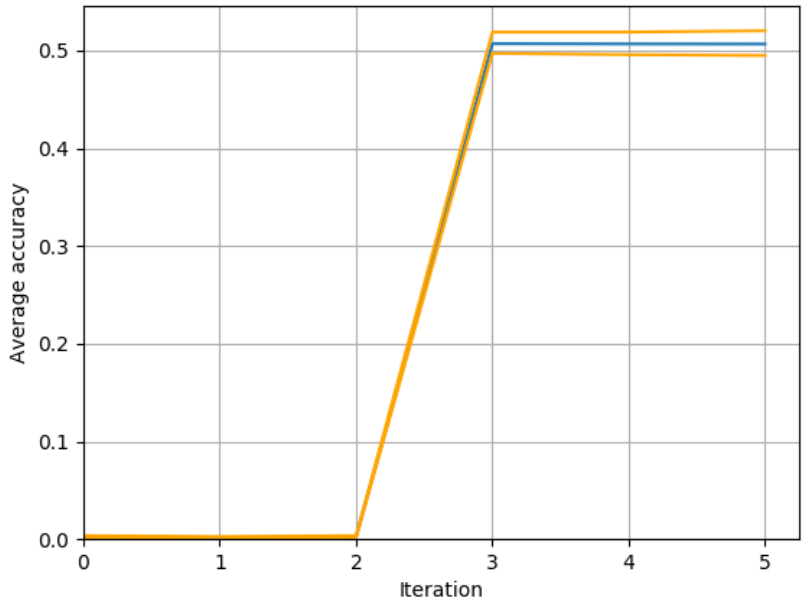


Figure B.5: The average (blue), best (orange), and worst (orange) accuracies for the model on Arithmetic2 with a fraction as validation of 0.5

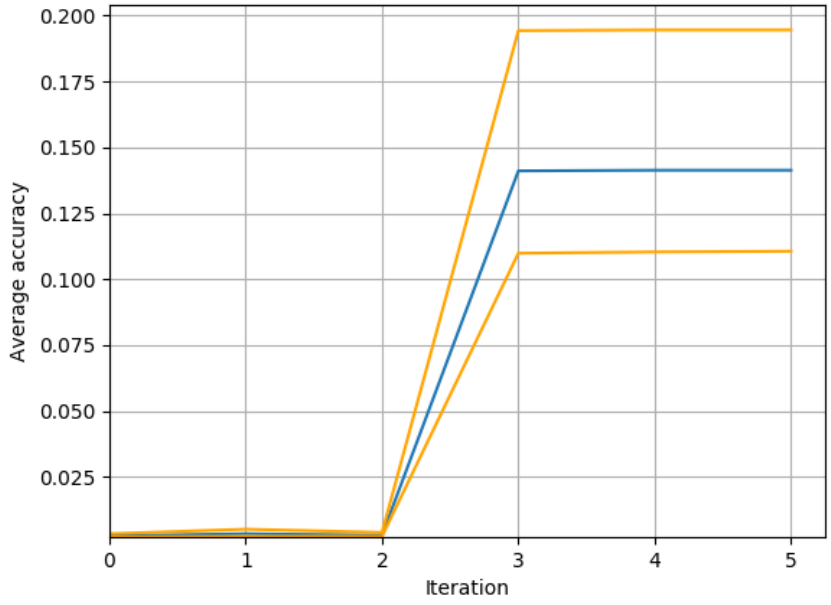


Figure B.6: The average (blue), best (orange), and worst (orange) accuracies for the model on Arithmetic2 with a fraction as validation of 0.9

B.1.2 Graphs from the domain of Boolean logic

This section presents the evaluation accuracy graphs from the dataset Logic1.

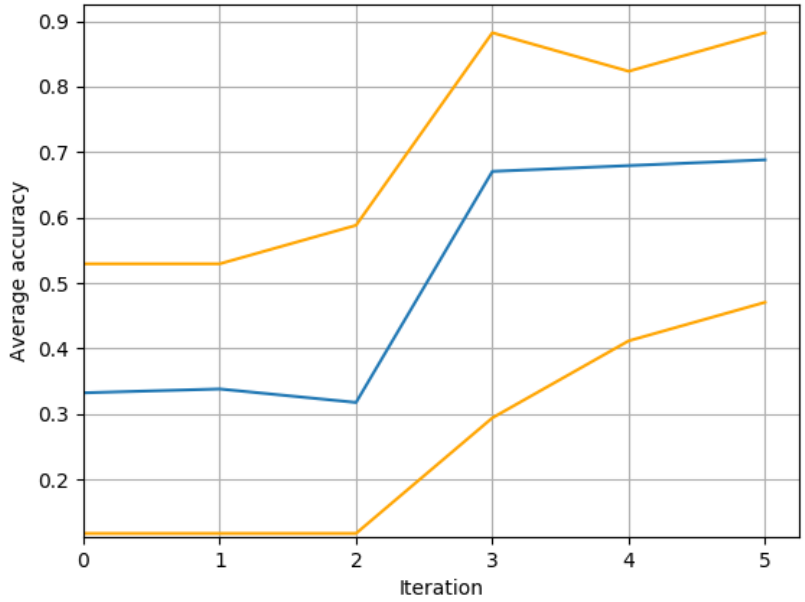


Figure B.7: The average (blue), best (orange), and worst (orange) accuracies for the model on Logic1 with a fraction as validation of 0.1

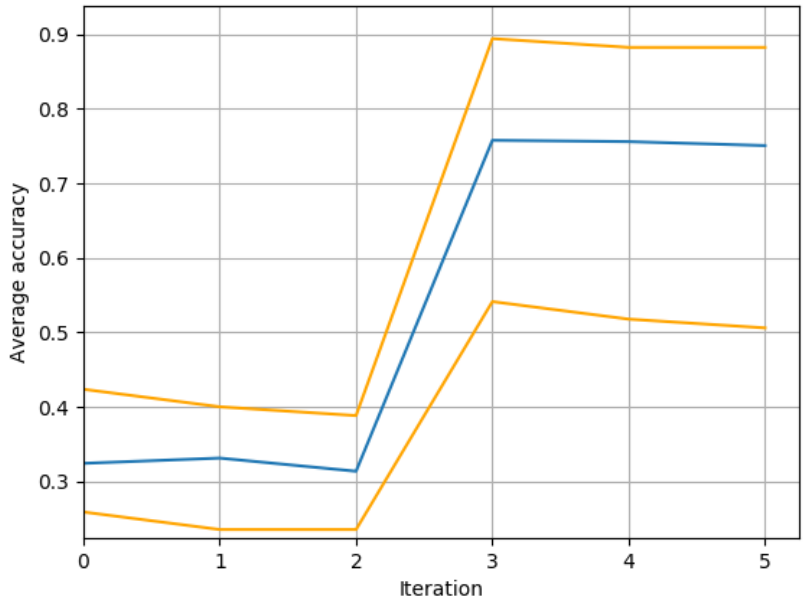


Figure B.8: The average (blue), best (orange), and worst (orange) accuracies for the model on Logic1 with a fraction as validation of 0.5

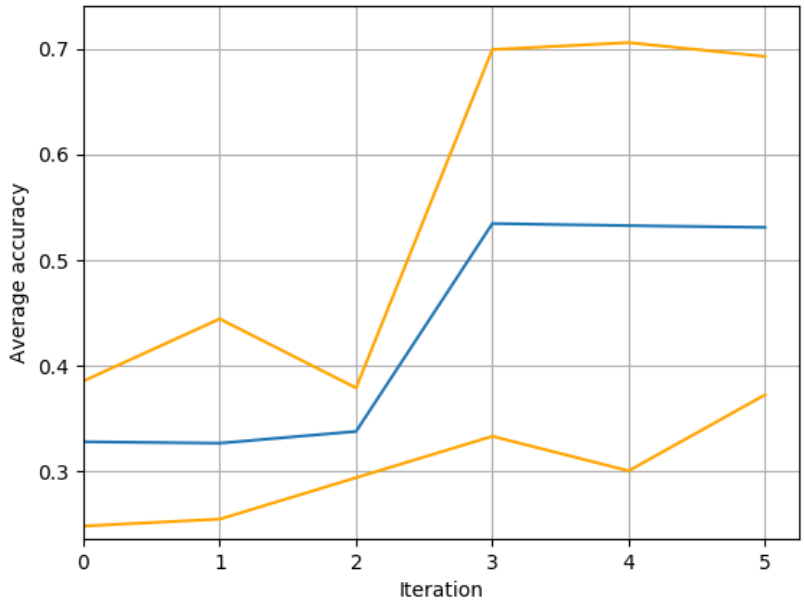


Figure B.9: The average (blue), best (orange), and worst (orange) accuracies for the model on Logic1 with a fraction as validation of 0.9

B.1.3 Graphs from the domain of simple English grammar

This section presents the evaluation accuracy graphs from the dataset Grammar1.

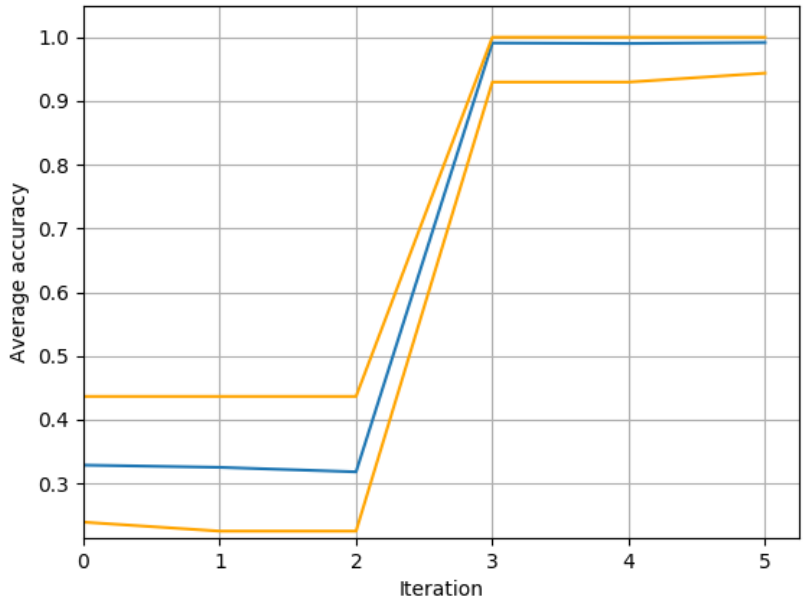


Figure B.10: The average (blue), best (orange), and worst (orange) accuracies for the model on Grammar1 with a fraction as validation of 0.1

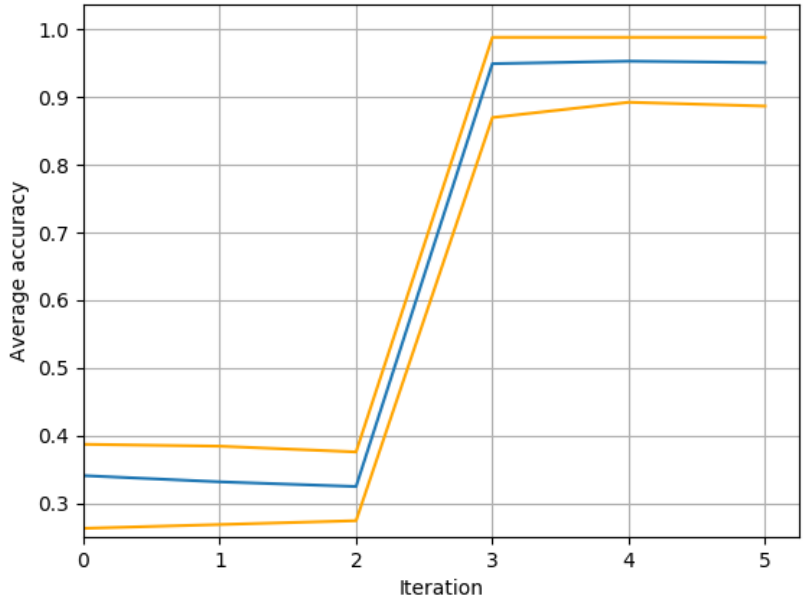


Figure B.11: The average (blue), best (orange), and worst (orange) accuracies for the model on Grammar1 with a fraction as validation of 0.5

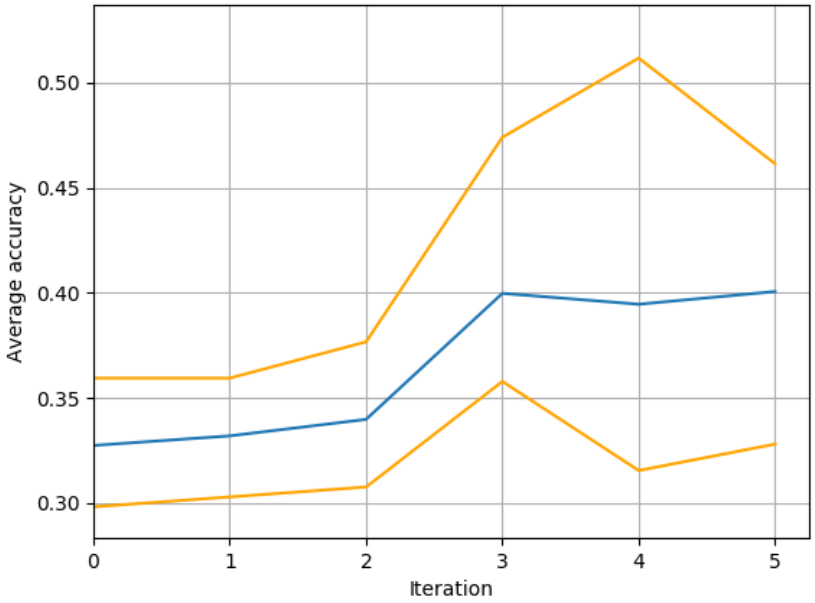


Figure B.12: The average (blue), best (orange), and worst (orange) accuracies for the model on Grammar1 with a fraction as validation of 0.9

B.2 Results from the LSTM network

B.2.1 Results From the domain of simple arithmetic

This section presents the evaluation accuracy graphs from the datasets Arithmetic1 and Arthmetic2.

B.2.1.1 Graphs from Arithmetic1

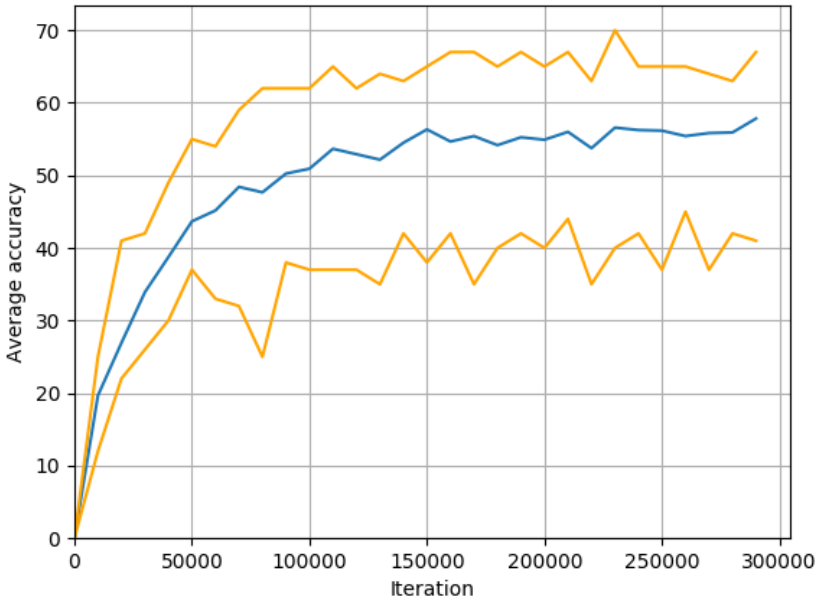


Figure B.13: The average (blue), best (orange), and worst (orange) accuracies for the LSTM network on Arithmetic1 with a fraction as validation of 0.1

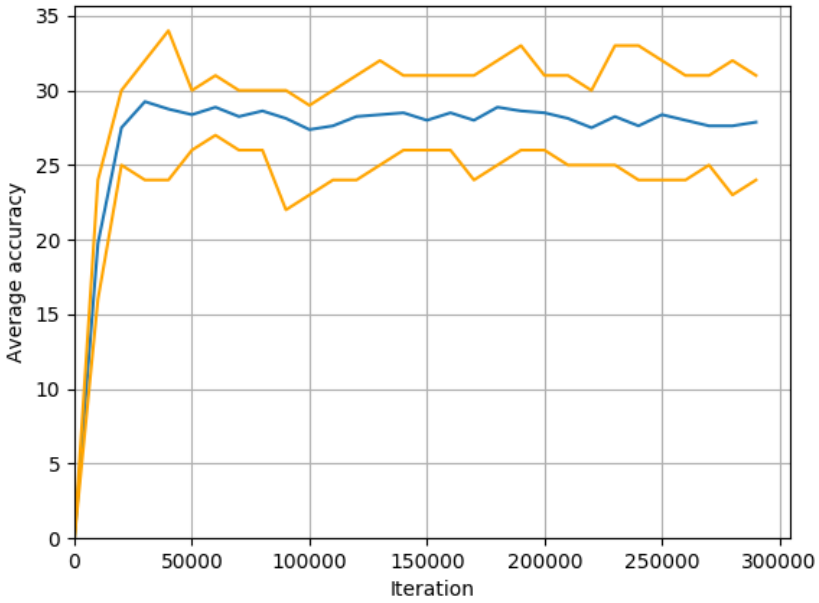


Figure B.14: The average (blue), best (orange), and worst (orange) accuracies for the LSTM network on Arithmetic1 with a fraction as validation of 0.5

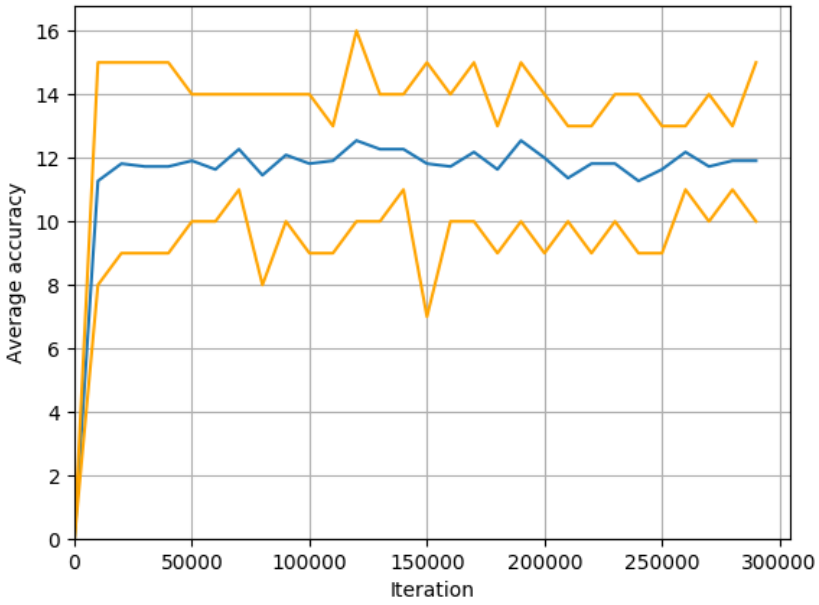


Figure B.15: The average (blue), best (orange), and worst (orange) accuracies for the LSTM network on Arithmetic1 with a fraction as validation of 0.9

B.2.1.2 Graphs from Arithmetic2

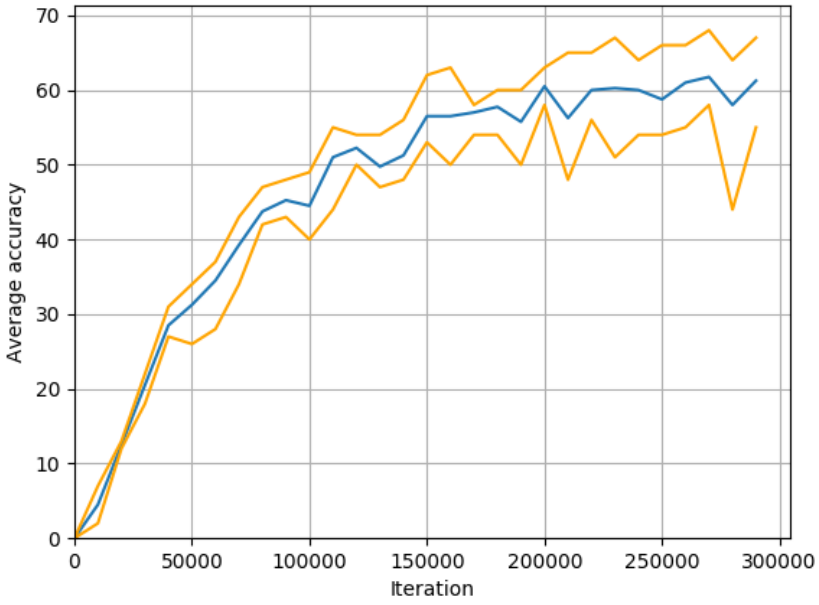


Figure B.16: The average (blue), best (orange), and worst (orange) accuracies for the LSTM network on Arithmetic2 with a fraction as validation of 0.1

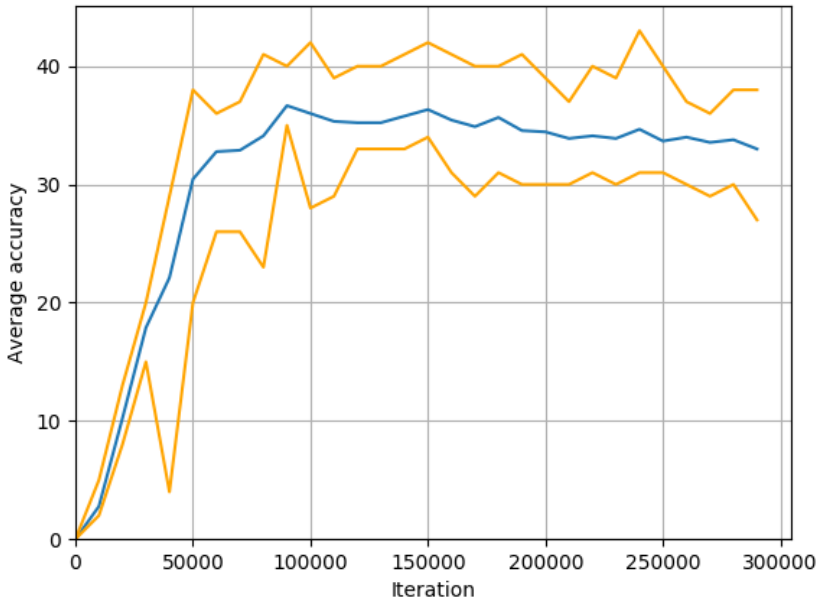


Figure B.17: The average (blue), best (orange), and worst (orange) accuracies for the LSTM network on Arithmetic2 with a fraction as validation of 0.5

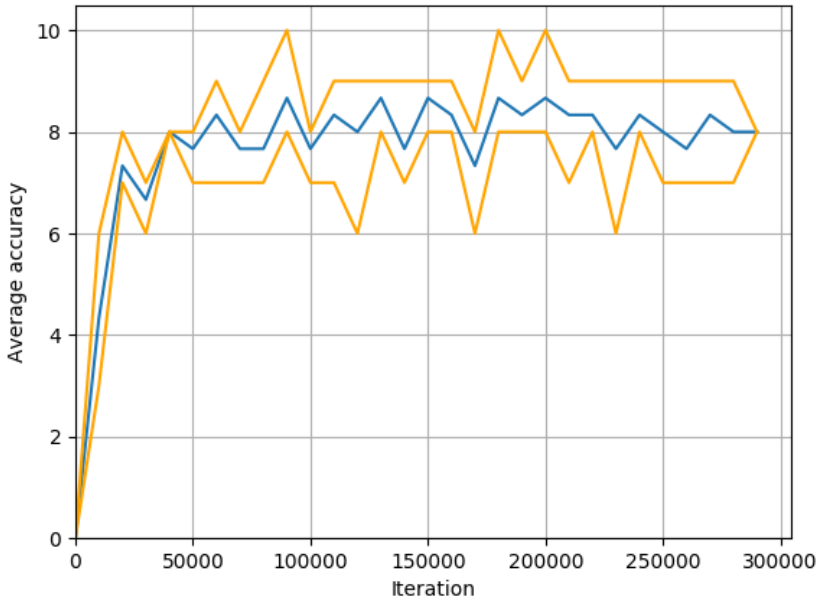


Figure B.18: The average (blue), best (orange), and worst (orange) accuracies for the LSTM network on Arithmetic2 with a fraction as validation of 0.9

B.2.2 Graphs from the domain of Boolean logic

This section presents the evaluation accuracy graphs from the dataset Logic1.

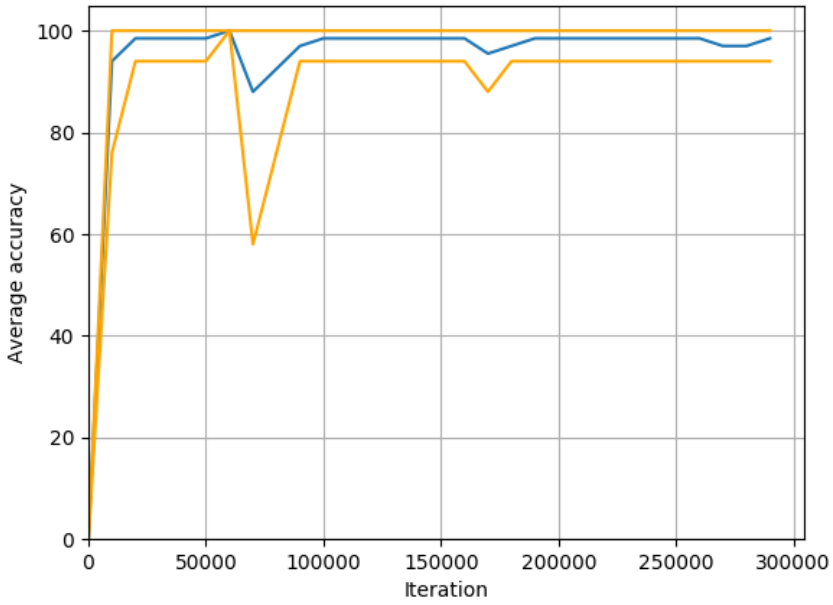


Figure B.19: The average (blue), best (orange), and worst (orange) accuracies for the LSTM network on Logic1 with a fraction as validation of 0.1

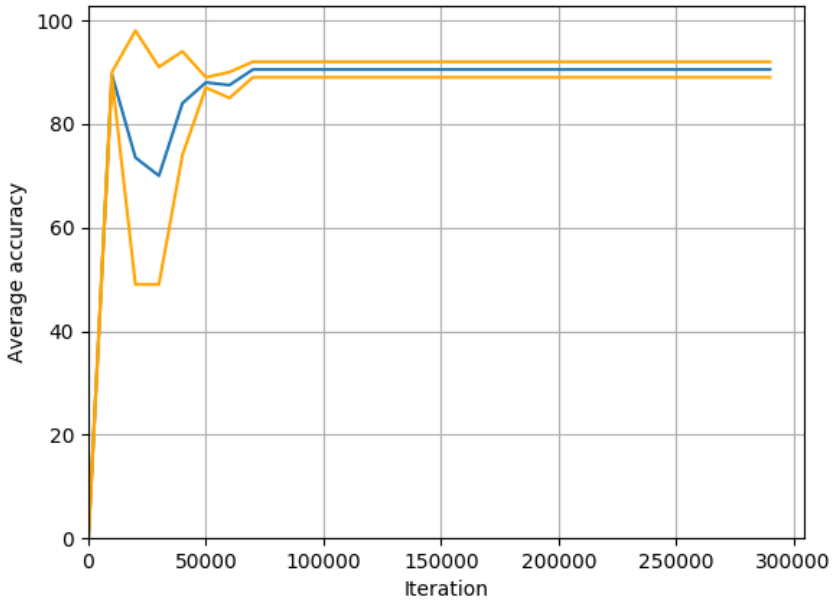


Figure B.20: The average (blue), best (orange), and worst (orange) accuracies for the LSTM network on Logic1 with a fraction as validation of 0.5

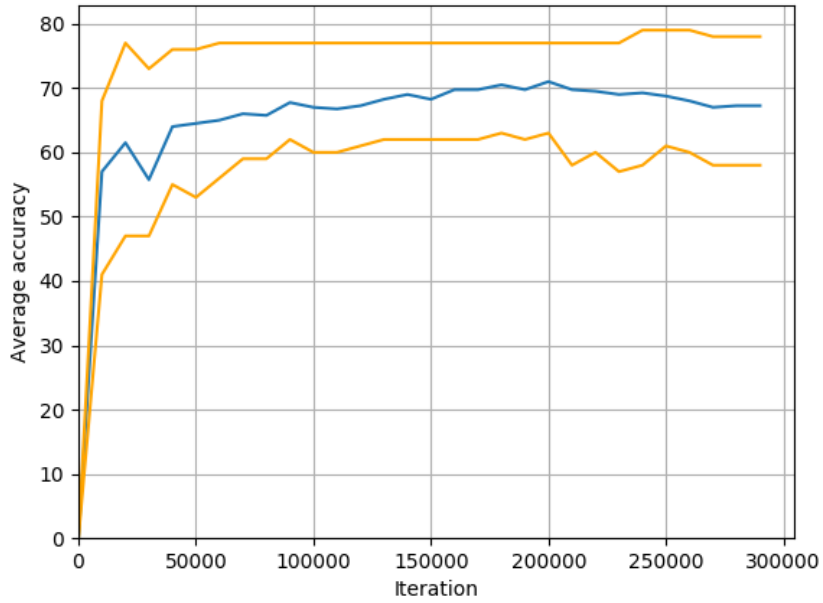


Figure B.21: The average (blue), best (orange), and worst (orange) accuracies for the LSTM network on Logic1 with a fraction as validation of 0.9

B.2.3 Graphs from the domain of simple English grammar

This section presents the evaluation accuracy graphs from the dataset Grammar1.

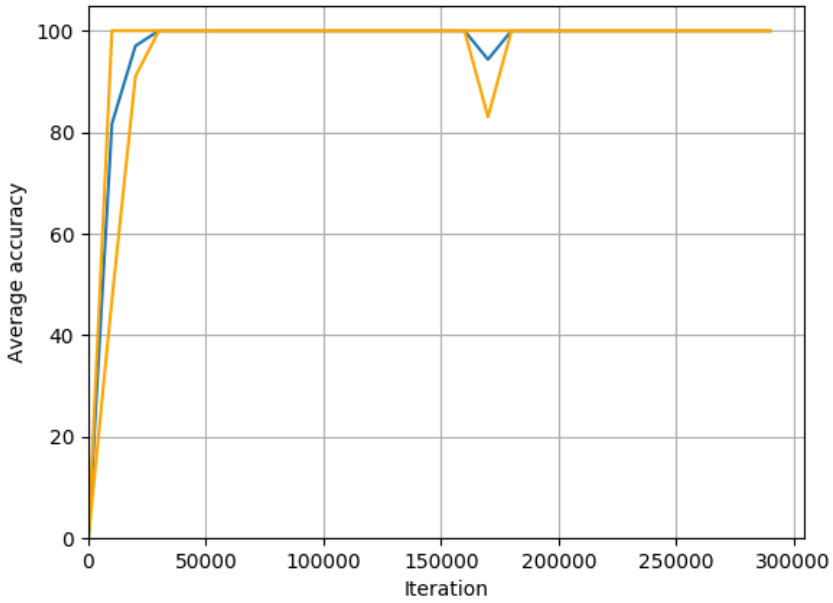


Figure B.22: The average (blue), best (orange), and worst (orange) accuracies for the LSTM network on Grammar1 with a fraction as validation of 0.1

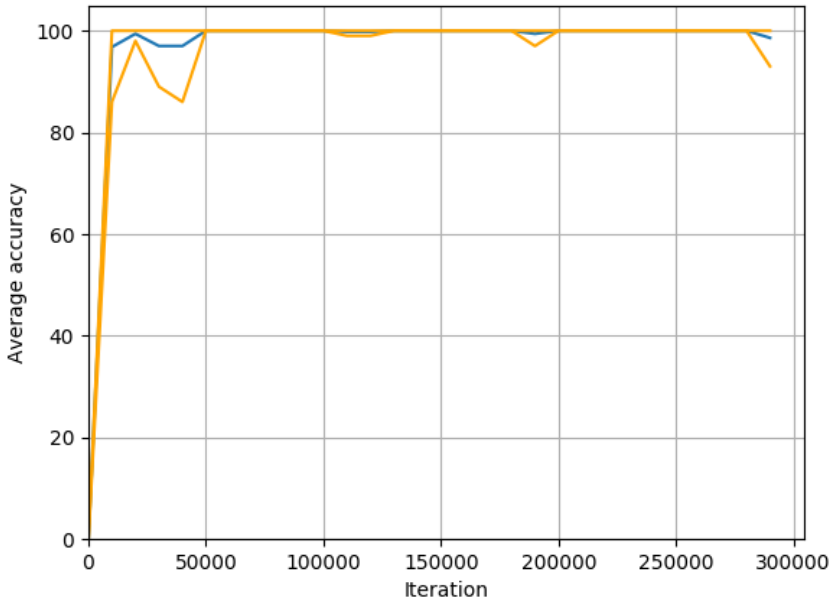


Figure B.23: The average (blue), best (orange), and worst (orange) accuracies for the LSTM network on Grammar1 with a fraction as validation of 0.5

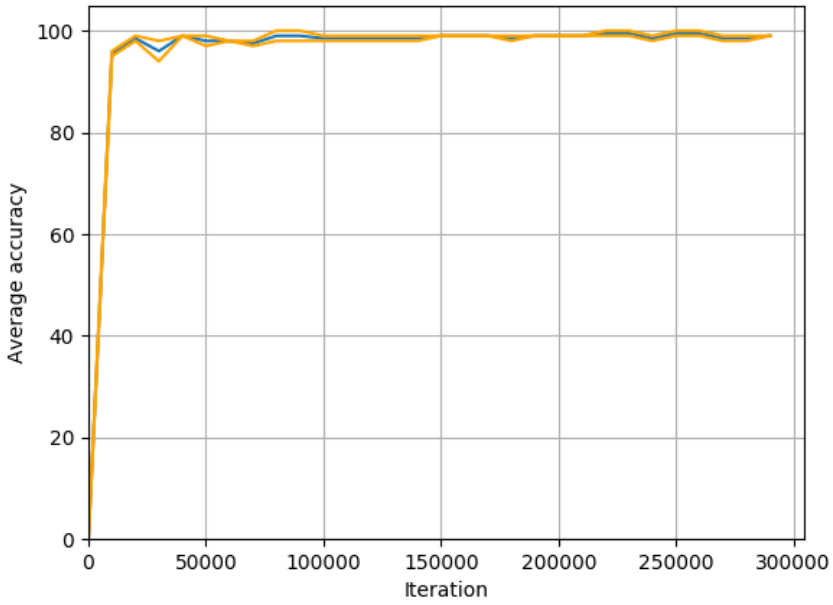


Figure B.24: The average (blue), best (orange), and worst (orange) accuracies for the LSTM network on Grammar1 with a fraction as validation of 0.9