Machine Learning for Program Synthesis

Videh Raj Nema and Shruthan R
National Institute of Technology Karnataka, videh.181co158@nitk.edu.in, shruthan.181co250@nitk.edu.in

Abstract - Initially defined by Alonzo Church as the problem to synthesize a circuit from mathematical requirements, program synthesis has become a problem being explored by researchers across multiple domains including programming languages and artificial intelligence. The programming languages community initially followed a rule-based approach followed by a combination of rule-based and search based approach by the artificial intelligence community. With the evolution of learning techniques supported by improvements in computer architecture, a hybrid approach involving both search and learning has gained traction among researchers in the program synthesis community. In our work, we first explore techniques based on supervised learning. We see attribute vectors and attribute functions and how they can be used to generate a distribution of programs based on the input-output examples which then aids the search across the program space. We then look at how encoder-decoder models and attention are incorporated to help improve the learning algorithm and some effective search techniques like tree beam search. We then move on to look at combining search and learning based techniques. We see how the task of synthesizing the program can be represented as a Markov Decision Process that is solvable via reinforcement learning. We see how program synthesis can be done using tree search guided by reinforcement learning and then look at how a popular heuristic search algorithm - Monte Carlo search can be combined with reinforcement learning, drawing parallels with the popular game playing program AlphaGo (by DeepMind). We finally present a few examples of program synthesis in action, where we feed to the model, the input-output examples along with the natural language description of the problem and some other inputs like argument types and return types, and we get the program "synthesized" by the model.

KEYWORDS

Program Synthesis, Machine Learning, Deep Learning, Reinforcement Learning, Software Engineering, Natural Language Processor, Domain Specific Language, Enumerative Search, Monte Carlo Tree Search (MCTS).

INTRODUCTION

This work aims to explore intelligent techniques to improve the traditional search-based techniques for *Program Synthesis*. Program Synthesis is the task of automatically finding programs from the underlying programming language that satisfy user intent expressed in some form of constraints or specifications.

Within the *Programming Languages* (PL) community, program synthesis is typically solved using *enumerative search*, finding correct programs for a given specification by naively enumerating candidates until a satisfying program is found. Various methods have been devised to do this efficiently and scalably.

Unlike typical compilers that translate a fully specified high-level code to low-level machine representation using a syntax-directed translation, program synthesizers typically perform some form of search over the space of programs to generate a program that is consistent with a variety of constraints (e.g., input-output examples, demonstrations, natural language, partial programs, and assertions).

However, using traditional search-based techniques is often insufficient for achieving good performance in terms of computational complexity, resources, etc. The program search space for program synthesis is generally huge. Using learning-based methods (from *Deep Learning* and *Deep Reinforcement Learning*), we can identify certain

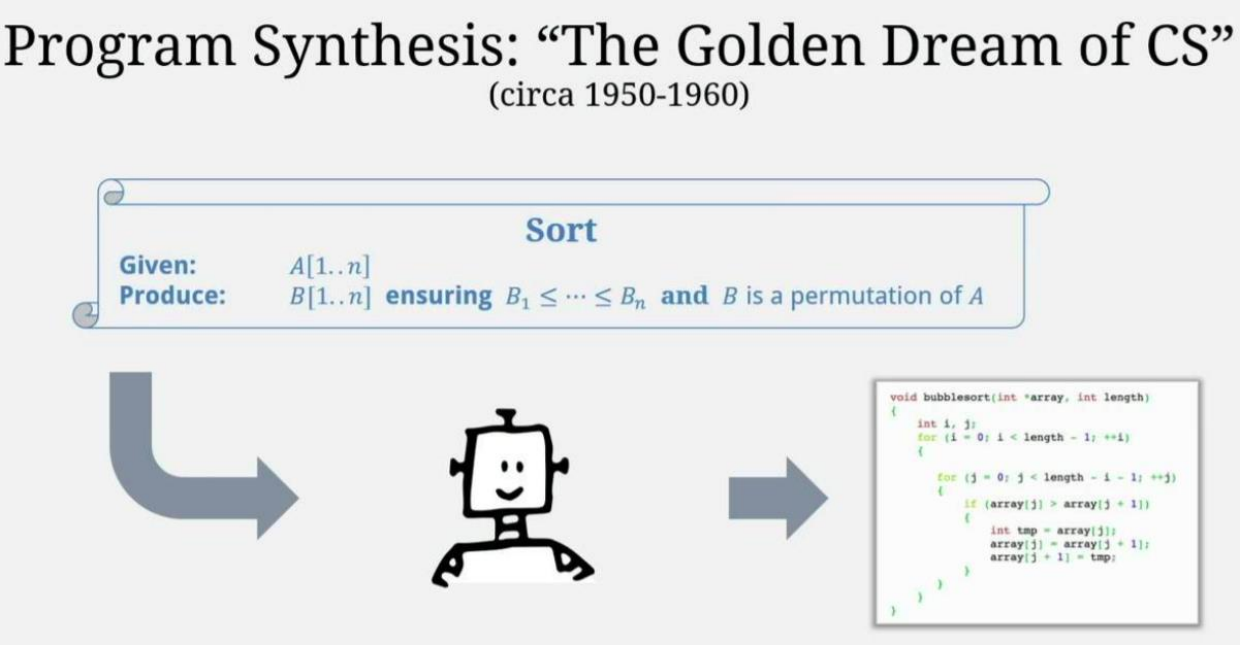


FIGURE 1.

PROGRAM SYNTHESIS PROBLEM FORMULATION FOR GENERATING THE PROGRAM FOR BUBBLE SORT. GIVEN IS AN INPUT DATA STRUCTURE A (ON WHICH THE SORTING PROGRAM IS APPLIED) AND OUTPUT DATA STRUCTURE B IS A SORTED PERMUTATION OF A. THE SORTING OPERATION IS DONE VIA THE CODE GENERATED BY THE PROGRAM SYNTHESIS TOOL. IMAGE CREDITS: [1].

attributes or features from the input-output specification, which can be used to narrow down or rank different programs to choose the most suitable program. Another interesting direction is to combine the search and learning-based methods to get the best of both worlds. This will be one of our key ideas in this project This project’s main objective is to develop and improve methods to synthesize programs, putting minimal human knowledge into the system. The existing methods have much scope for improvement, and we are indeed quite far from *automated programming*. This work aims to contribute in this direction.

Although not apparent, development in such techniques can be extremely beneficial. It can be used successfully in *software engineering*, biological discovery, computer-aided education, end-user programming, and data cleaning. In the last decade, several applications of synthesis in programming by examples have been deployed in mass-market industrial products [2].

Most popular programming languages have many constructs, thereby increasing the search space drastically. Hence most program synthesis tools today try to generate simple programs in *Domain-Specific Languages* (DSL) with a limited number of high-level functions and operations. However, there have been attempts to try program synthesis for more popular frameworks like TensorFlow recently also [3].

The working of a typical program synthesis tool is described below pictorially:

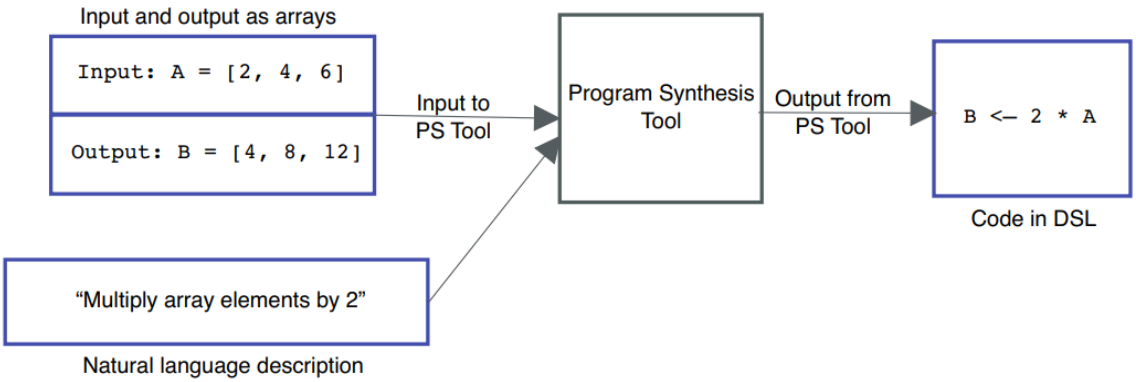


FIGURE 2

ABOVE IS A GENERIC PROGRAM SYNTHESIS TOOL. THE INPUT TO THE SYSTEM (OR TOOL) IS INPUT-OUTPUT EXAMPLES DEPICTING THE DESIRED MAPPING. WE CAN HAVE MULTIPLE EXAMPLES TO AVOID THE OVERFITTING OF THE SYSTEM TO THE EXAMPLE. ALONG WITH THIS, WE HAVE AN OPTIONAL *NATURAL LANGUAGE DESCRIPTION* OF THE TASK WE WANT THE PROGRAM TO ACCOMPLISH. THE OUTPUT IS THE CODE IN THE REQUIRED PROGRAMMING LANGUAGE THAT PERFORMS THE UNDERLYING TASK.

There are two significant motivations behind doing this work -

Enormous Program Synthesis applications: As mentioned earlier, one can find numerous applications of this automated approach. Following are a few exciting applications in the field of software engineering -

- Code repair: There have been recent improvements in this problem [4-6]. The problem goes as follows - Given a *buggy* program and a specification, the code repair problem requires computing modifications to the *buggy* program to obtain a new program that satisfies the specification. The general idea behind these techniques is to first insert alternate choices for the expressions present in the buggy program and then use synthesis techniques to find replacements or modifications of the expressions from the inserted choices such that the updated program satisfies the given specification.

- Code suggestions: This is a direct application of program synthesis to *software engineering*. Most modern code editors and IDEs include an “autocompletion” capability, which predicts the next likely token in the program based on the previously typed ones (e.g., IntelliSense in Microsoft Visual Studio or Content Assist in Eclipse).

- Program synthesis can enhance this capability by automatically completing whole snippets of code instead of single tokens. Such a tool can be handy for software engineers and developers to bring out as efficient implementation of the software design as possible. Besides, it can take into account optional annotations about the desired snippet properties from the programmer, such as expression types or keywords in the identifiers.

Following are some other applications -

- Probabilistic modeling: A probabilistic model incorporates random variables and probability distributions into an event or phenomenon model. Probabilistic modeling is a technique for the analysis of complex dependent systems based on empirical evidence. The most widespread tools for probabilistic modeling are various forms of machine learning: Markov models, neural networks, Bayesian networks, etc. Typically, a probabilistic model involves two components: a structure of the system (i.e., a description of dependencies between system components) and its parameters (i.e., particular statistical distributions for all components). Most machine learning forms focus on inferring parameters of the model, assuming a given structure such as linear regression. Program synthesis has been applied to learning the model structure in the form of a probabilistic program [7].

- Concurrent programming: Writing concurrent programs with efficient synchronization is a challenging and error-prone task even for proficient programmers. There have been several synthesis techniques designed to help programmers write such complex code, e.g., by automatically synthesizing placement of minimal synchronization constructs in a concurrent program [8-9] or by inferring the placement of memory fences in concurrent programs running on relaxed memory models [10].

Machine Learning is a subset of *Artificial Intelligence*. It is an excellent tool for making computers learn without explicitly being programmed to do so. This is commonly realized using function approximation. Machine learning can be employed to intelligently search over the search space of programs.

With the success and popularity of Deep Learning, powerful Deep Neural Networks have become a reliable tool for function approximation and developing learning algorithms. Program Synthesis is one of the applications where it can be used.

As mentioned before, a program synthesis tool generally takes as input, the examples of input and desired output of the function (or sequence of functions) along with an optional natural language/formal and logical description of what the function (or sequence of functions) is supposed to do. It outputs code in the desired programming language (depending on the development of the tool) which achieves the required function. Note that program synthesis does not try to predict the output for a new input (input here referring to the input to the function or sequence of functions), as is done in typical machine learning but rather tries to *generate code* which when applied on the input, gives the desired output. Achieving high accuracy and efficiency in generating DSLs is often seen as a precursor to generating full-fledged programs in popularly used languages. This modern view of program synthesis is shown in the figure below:

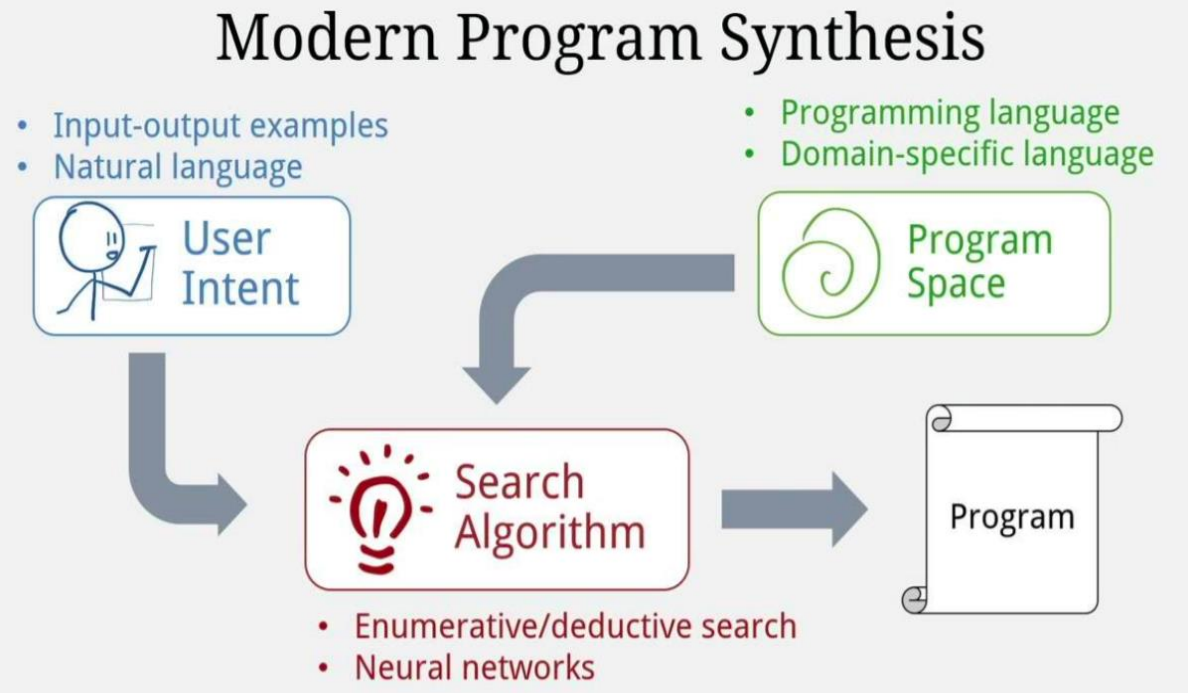


FIGURE 3

GENERAL RECIPE FOR MACHINE LEARNING-BASED PROGRAM SYNTHESIS. THE BLUE AND GREEN COLORED REGIONS SPECIFY THE INPUTS WHEN TRAINING THE MODEL. THE RED COLORED REGION IS THE PROGRAM SYNTHESIS ALGORITHM, WHICH INCLUDES LEARNING-BASED FUNCTION APPROXIMATORS LIKE NEURAL NETWORKS AND SEARCH ALGORITHMS LIKE ENUMERATIVE/DEDUCTIVE SEARCH OR TREE-BASED SEARCH ALGORITHMS. THE OUTPUT IS THE SYNTHESIZED PROGRAM. IMAGE CREDITS: [1]

In this work, we aim to study and improve Machine Learning techniques for synthesizing programs. We intend to use *Deep Learning* and *Deep Reinforcement Learning* techniques to solve this problem. We intend to develop and improve methods to synthesize programs, putting minimal human knowledge into the system. The existing methods have much scope for improvement, and we are indeed quite far from *automated programming*. This work aims to contribute in this direction.

However, this problem is extremely challenging, and the complexity of the synthesized programs by existing approaches is still limited. This work aims to generate programs with more complexity and better generalizability while maximizing the correctness. In this process, besides enabling real-world applications using program synthesis, we hope to contribute to addressing core challenges in Deep Learning, including generalization, search, abstraction, and representation. We believe that solving the program synthesis problem is a good step towards solving *Artificial General Intelligence.*

As a part of this project, we also study various learning-based methods in Deep Learning and Deep Reinforcement Learning. Our objective is to effectively utilize these methods and understand the core concepts of machine learning. We hope to address core challenges in deep learning, including generalization, search, abstraction, and representation. We would also like to address sample-efficiency and brittleness of reinforcement learning in real-world applications.

RELATED WORK

Our work has been highly inspired by the following papers:

1. Sumit Gulwani, Alex Polozov and Rishabh Singh, Program Synthesis [2]
   This paper presents a general overview of the state-of-the-art approaches to program synthesis, its applications, and subfields. It discusses the general principles common to all modern synthesis approaches such as syntactic bias, oracle-guided inductive search, and optimization techniques. It then presents a literature review covering the four most common state-of-the-art techniques in program synthesis: enumerative search, constraint solving, stochastic search, and deduction-based programming by examples and concludes with a brief list of future horizons for the field.

2. Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju et al. RobustFill: Neural Program Learning under Noisy I/O [11]
   In this paper, *neural program synthesis* and *neural program induction* are compared on a large-scale, real-world learning task. The paper describes a neural model which uses a modified attention RNN to allow encoding of variable-sized sets of I/O pairs and contrasts it with traditional rule-based approaches and also shows how a synthesis model outperforms a comparable induction model on this task, but more importantly demonstrate that the strength of each approach is highly dependent on the eval-uation metric and end-user application.

3. Matej Balog et al. DeepCoder: Learning to Write Programs [12]
   In this paper, a first line of attack for solving programming competition-style prob-lems from input-output examples using deep learning. The approach is to train a neural network to predict properties of the program that generated the outputs from the inputs. The neural network's predictions are used to augment search techniques from the programming languages community, including enumerative search and an SMT-based solver. Empirically, it is shown that their approach leads to an order of magnitude speedup over the strong non-augmented baselines and a Recurrent Neural Network approach, and is able to solve problems of difficulty comparable to the simplest problems on programming competition websites.

4. Illia Polosukhin and Alexander Skidanov, Neural Program Search: Solving Programming Tasks from Description and Examples [13]
   The paper presents a Neural Program Search, an algorithm to generate programs from natural language description and a small number of input/output examples. The algorithm combines methods from Deep Learning and Program Synthesis fields by designing rich domain-specific language (DSL) and defining an efficient search algorithm guided by a Seq2Tree model on it. The algorithm is shown to significantly outperform a sequence-to-sequence model with attention baseline.

5. Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, Sumit Gulwani, Neural-Guided Deductive Search for Real-Time Program Synthesis from Examples [14]
   This paper proposes Neural Guided Deductive Search (NGDS), a hybrid synthesis technique that combines the best of both symbolic logic techniques and statistical models. Thus, it produces programs that satisfy the provided specifications by construction and generalize well on unseen examples, similar to data-driven systems and claims to effectively utilize the deductive search framework to reduce the learning problem of the neural component to a simple supervised learning setup which allows for training on sparingly available real-world data and still leverage powerful recurrent neural network encoders. The effectiveness of the method is demonstrated by evaluating on real-world customer scenarios by synthesizing accurate programs with up to 12x speed-up compared to state-of-the-art systems.

6. Riley Simmons-Edler, Anders Miltner, Sebastian Seung, Program Synthesis Through Reinforcement Learning Guided Tree Search [15]
   This paper proposes representing the task of synthesizing a given program as a Markov decision process solvable via reinforcement learning (RL). From observations about the states of partial programs, an attempt is made to find a program that is optimal over a provided reward metric on pairs of programs and states. This approach is instantiated on a subset of the RISC-V assembly language operating on floating point numbers, and as an optimization inspired by search-based techniques from the PL community, and combines RL with a priority search tree. The effectiveness of the combined method is demonstrated by comparison to a variety of baselines, including a pure RL ablation and a state of the art Markov chain Monte Carlo search method on this task.

FROM TRADITIONAL APPROACHES TO SUPERVISED LEARNING

We shall first look at the problem and how it is traditionally looked upon and then look at modern approaches to the problem.

1. Background

Traditionally, program synthesis has generally been rule-based, where a program is generated from a well-defined specification using a formal grammar. Writing a formal grammar for a language is extremely difficult and involves a high level of creativity and ingenuity. Modern synthesis tools instead rely on input-output examples to act as an approximation to the specification. Statistical learning-based models are generally applied to these input/output examples to generate a (consistent) program.

Most work in statistical program learning falls into two general categories - (i) *Neural Program Induction*: here, the aim is to learn to generate the output directly from the input using a **latent** program representation. The latent representation (evident from its name) generally does not have an intuitive or programmatic interpretation. (ii) *Nural Program Synthesis:* here, a program is generated based on the input/output examples. This program has to produce correct outputs for the inputs in the training data (*consistency*) and produce the expected output for new input to the program (*generalizability*) [11].

In this (mini) project, we will consider Neural Program Synthesis as the primary problem.

The problem we aim to solve is the following: *Given Input-Output examples, generate a program that has a behavior consistent with the given examples.*

Given the complexity of modern and popular programming languages leading to large search space, most work in program synthesis focuses on generating *Domain-Specific Languages* (DSLs). DSLs are programming languages suitable for a specialized domain but are more restrictive than full-featured programming languages. Some aspects of a programming language like loops, certain data types may be excluded, and few primitive operations like 'add 1', concatenate, etc. may be allowed [12].

II. Natural Language Processor

In most program synthesis models, a natural language description of the desired program is provided. Processing this natural language description is generally a challenge due to the ambiguity in natural language. Strict rules are generally enforced to provide the natural language description. The natural language description may be preprocessed using standard NLP techniques like stripping of irrelevant characters and punctuations, stemming or lemmatization, etc. Training data consists of example pairs of English sentences and corresponding intended programs in the language to be generated (generally a DSL). A training phase infers a dictionary relation over pairs of English words and DSL terminals (in a semi-automated interactive manner), and optimal weights/classifiers (in a completely automated manner) for use by the generic synthesis algorithm. This approach can be seen as a meta-synthesis framework for constructing NL-to-DSL synthesizers. In the example we considered above, suppose the description is - "sort the elements of the array in O(nlogn) time complexity". The job of the natural language processor is to parse this statement and give this information to the learning and search algorithm [16].

III. Outline of the machine learning model

The general approach followed in most work on using machine learning in program synthesis has been outlined in this section.

In addition to the DSL mentioned above, an attribute function A is defined, which maps the programs P in the DSL to a finite attribute vector $a = A(P)$. The machine learning model predicts a distribution $d(a|IO)$, where IO is the set of input/output pairs:

$$IO = \{(I1, O1), (I2, O2), , (In, On)\}$$

The search procedure aims to search over programs P as ordered by $d(a|IO)$. The quality of a is determined by its ability to reduce the search space and its predictability from the input/output examples. The machine learning problem is to learn a distribution of attributes given input-output examples, $d(a|IO)$. This distribution is used to guide the search for programs in the program space of the DSL.

A generic algorithm may be described as follows:

Algorithm 1 - Generic Machine Learning for Program Synthesis

1: Obtain the input-output examples IO = $\{(I_1, O_1), (I_2, O_2), \dots, (I_n, O_n)\}$.

2: Specify the attribute function A to obtain the attribute vectors a = A(P), such that it is predictable from the input-output examples and conditioning on its value significantly reduces the effective size of the search space.

3: Generate the dataset: $((P^{(n)}, a^{(n)}, IO^{(n)}))_{n=1}^{N}$ of programs $P^{(n)}$ in the chosen DSL, their attributes $a^{(n)}$, and accompanying input-output examples $IO^{(n)}$.

4: Learn the distribution of attributes given input-output examples, $d(a|IO)$.

5: Search through the program space guided by $d(a|IO)$.

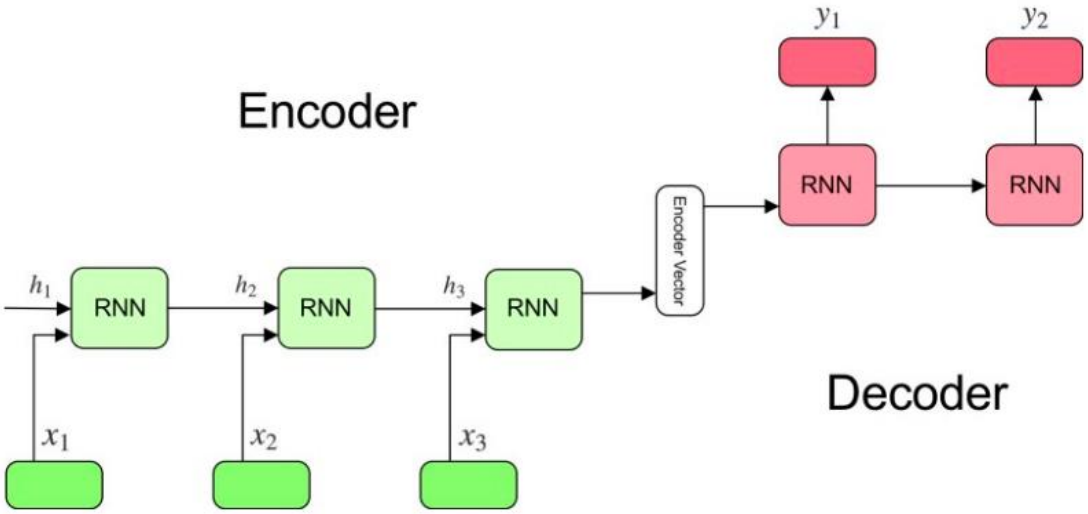


FIGURE 4
REPRESENTATIONAL ENCODER-DECODER MODEL.
IMAGE CREDITS: DEEPLEARNING.AI

Encoder-decoder models are commonly used to learn the mapping (A(P)) from input-output examples to attributes. They are described below.

IV. Encoder-Decoder Models:

Sequence-to-sequence is a relatively new paradigm introduced by Google in 2014, which incorporated the encoder-decoder model. At a high level, its an end-to-end model consisting of two RNNs:

- an *encoder* which encodes the model's input sequence into a fixed-size "context vector" and
- a *decoder*, which generates an output sequence using the context vector from above as a "seed."

The idea is to use one RNN to read the input sequence, one time step at a time, to obtain a sizeable fixed-dimensional vector representation, and then to use another RNN to extract the output sequence from that vector [17].

V. Improving encoder-decoder models: Attention

Attention was presented by Dzmitry Bahdanau et al. [18]. A major issue with traditional sequence-to-sequence models is what is referred to as the information bottleneck. Individual inputs begin stacking up in the encoder's final hidden state because the seq-to-seq uses a fixed-length memory, and longer sentences become an issue. Another shortcoming is that more importance is given to the inputs in later timesteps

Attention is popularly used in Natural Language Processing tasks like Neural Machine Translation (NMT). In NMT, the meaning of a sentence is mapped into a fixed-length vector, which then generates a translation based on that vector as a whole. The goal is not to translate the sentence word for word, but rather pay attention to the general, "high level" overall sentiment. Attention mechanisms, in addition to improving accuracy, also help to increase efficiency.

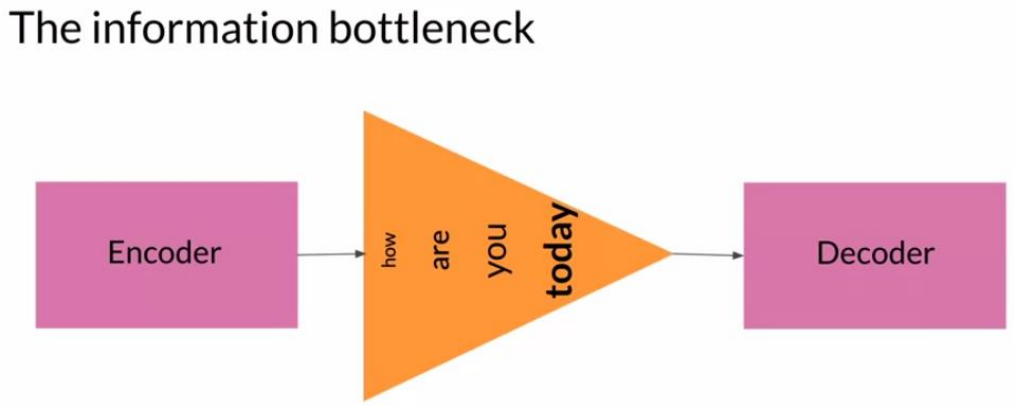


FIGURE 5
REPRESENTATION OF INFORMATION BOTTLENECK
IMAGE CREDITS: DEEPLEARNING.AI

Applications (of program synthesis) like RobustFill [11] use attention mechanisms for program synthesis as described below:

RobustFill takes in `(I, O)` as input and produces a program `P` as output in a single-example model. Four increasingly complex models have been described in RobustFill as follows:

- Basic sequence-to-sequence model: Each sequence is encoded with a non-attentional LSTM, and the final hidden state is used as the initial hidden state of the next LSTM.
- Attention-A: O and P are attentional LSTMs, with O attending to I and P attending to O.
- Attention-B: Same as Attention-A, but P uses a double attention architecture, attending to both O and I simultaneously.
- Attention-C: Same as Attention-B, but I and O are bidirectional LSTMs

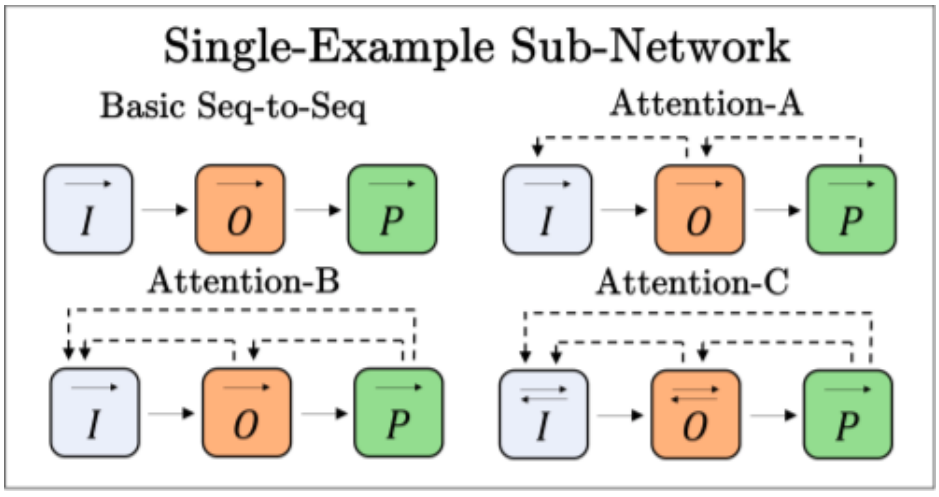


FIGURE 6
DIAGRAMATIC REPRESENTATION OF ATTENTION USED IN ROBUSTFILL [11]

Applications like RobustFill, FlashFill etc, are generally used for data transformation tasks and help users perform data transformations using examples. Sample input/output examples are given below:

Input	Output
I_1 = Monday	O_1 = mon
I_2 = Tuesday	O_2 = tue
I_3 = Wednesday	O_3 = wed
I^Y_1 = Thursday	O^Y_1 = thu
I^Y_2 = Friday	O^Y_2 = fri
Program : `P = ToCase(Lower,SubStr(1,3))`	

These methods rely on a small domain-specific language (DSL) and then develop algorithms to efficiently search the space of programs.

5.6 More advanced search techniques and attention

Similar techniques are also used to synthesize programs from a given natural language description. One of the most important challenges faced in such applications is the ambiguity in natural language. Applications like in [13] use a Seq2Tree model[Alvarez-Melis & Jaakkola(2016)] .

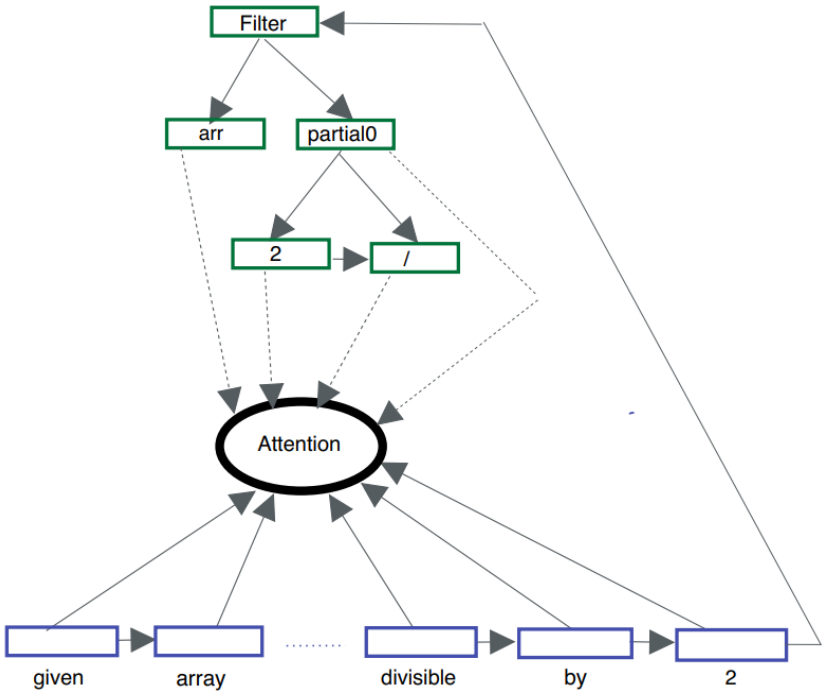


FIGURE 7

EXAMPLE OF SEQ2TREE ENCODER-DECODER MODEL FOR "GIVEN ARRAY ARR, RETURN VALUES DIVISIBLE BY TWO". IN THE BOTTOM IS AN ENCODER WITH EMBEDDINGS+GRU CELL, ON THE TOP IS A DOUBLY-RECURRENT DECODER WITH ATTENTION.

This model consists of a sequence encoder that reads the problem statement and a tree decoder augmented with attention that computes probabilities of each symbol in an Abstract Syntax Tree node one node at a time. Tree beam search is used to choose the tree that is consistent with the given input/output pairs from a set of most likely trees that is obtained using the probabilities computed above. Tree beam search may be represented algorithmically as follows:

Algorithm 2 - Tree Beam Search

```
queue ← HeapCreate()
model ← Seq2Tree(task description)
trees visited ← 0
HeapPush(queue, EMPTY_TREE)
while HeapLength(queue) > 0 and trees visited < MAX_VISITED do
  cur tree ← HeapPopMax(queue)
  empty node ← FindFirstEmptyNode(cur tree)
  if empty node = null then
        trees visited ← trees visited + 1
        if RunTests(cur tree, sample tests) = PASS then return cur tree
        else continue
for all (prob, symbol) in GetProbs(model, empty node) do . In decreasing order of probabilities
        if prob < THRESHOLD then break
        if SymbolMatchesNodeType(symbol, empty node) then
              new_tree ← CloneTreeAndSubstitute(cur tree, empty node, symbol)
              HeapPush(queue, new tree)
while HeapLength(queue) > QUEUE N do
        HeapPopMin(queue)
return null
```

Source [13]

Results and outputs using the above approaches are mentioned in RESULTS section

REINFORCEMENT LEARNING AND BEYOND

Next, we look at deep reinforcement learning-based methods for program synthesis. We specifically aim to study the methods which combine search and learning techniques for program synthesis, to get the best of both worlds (Programming languages community and Machine learning community). First, let us start with an overview of reinforcement learning.

I. Reinforcement Learning - an overview

Reinforcement learning (RL) is a branch of machine learning that aims to learn using rewards and/or punishments. To be a bit more technical, RL focuses on smart trial and error feedback-based learning bounded by a mathematical framework.

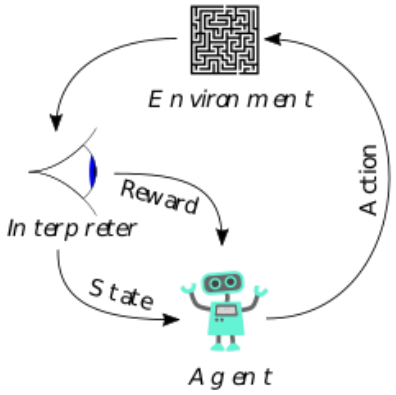


FIGURE 8
ABOVE SHOWN IS A GENERIC ILLUSTRATIVE DIAGRAM THAT SHOWS THE AGENT-ENVIRONMENTS LOOP.
IMAGE CREDITS [19]

In reinforcement learning, we typically have an agent(s) that receives some information about its surroundings (environment). Formally, this information is referred to as a state. Based on its policy (strategy) and the state, the agent takes certain actions that are fed into the environment. The environment, which is nothing but the outer world or the surroundings, returns feedback (or reward), based on the action, and the next state to the agent. This cycle continues until a terminal state is reached.

RL is one of the fastest-growing fields in machine learning and has gained a lot of traction in recent years. RL, in very simple words, is making the machines learn things just the way humans do. It is based on a potential positive or negative feedback that we get on accomplishing a particular task. RL, in many ways the closest to how humans think of a problem, out of all other machine learning paradigms.

There have been a lot of amazing applications of RL. It has been used extensively in games, robotics, recommender systems like Netflix, online advertising websites like Amazon, resource allocation in cloud computing, etc. We'll look at more such applications later.

Reinforcement learning, in many ways, is deeply connected to the way humans learn. Let us dive into a bit of history for this. RL is broadly a mixture of various theories and subjects that have traditionally been used by scientists and psychologists to better understand human beings, their behavior, evolution, and survival. It has deep connections with behavioral psychology and neuroscience. RL combines the study of these fields:

1. Trial and error psychology in *animal learning*.
2. Cognitive science, i.e., the scientific study of human reasoning, emotions, language, perception, attention, and memory.
3. Control theory, probability theory, dynamic programming (computer programming by breaking down a problem into smaller subproblems and solving them by recursion).
4. Temporal difference/disconnect: This is an important characteristic of RL. This means that a prediction that you make at time step $t+1$ is better than what you make at time step t. It is like going forward in the game and then updating your prediction according to what you saw in the future.

All these theories were intertwined with Computer Science and Artificial Intelligence to form a new mechanism for making machines learn to do some useful stuff, which we know as Reinforcement learning, where the "reinforcement" or the feedback is most important.

Let us now look at some terminologies that we'll be using:

- Agent: This realizes the learning algorithm that decides actions. Our problem is focussed around the agent and aimed to improve its performance.
- Environment: This comprises the outer world or the surroundings. In the problems that we consider here, it is everything apart from the agent.
- State: Denoted by s_t, a state is an observation that the agent receives from the environment. It acts as an interface between the agent and the environment. It is important to note that all information about the environment is not necessarily available to the agent at all times. For example - in the game of chess, a state may be the current position of the pieces on the board. Here, all the information from the environment is available. But now consider a card game where the players need to take cards from a shuffled deck. Here player 1 knows nothing either about player 2's cards or about the cards in the deck, even though they are part of the environment.
- Action: Denoted by a_t, this is a function that maps one state to another. It basically means the decisions that the agent takes.
- Reward: Denoted by r_t, the reward is the most important element in RL. It is basically the feedback signal that the agent gets from the environment when it executes an action in a state. Reward basically ties all the previous ideas together.
- State transition probability: The state transition probability matrix ($P(s_{t+1}|s_t, a_t)$). It basically denotes the probability of the agent reaching the next state, given the current state and the current action. These probabilities do not depend on the agent and are decided by the environment.
- Discount factor: This is a value between 0 (inclusive) and 1 (exclusive) and denotes the far-sightedness of the agent. It compares how much the agent cares about the rewards in the near and the far future. It is denoted by γ.
- Markov decision process (MDP): A Markov decision process or MDP is defined as follows -

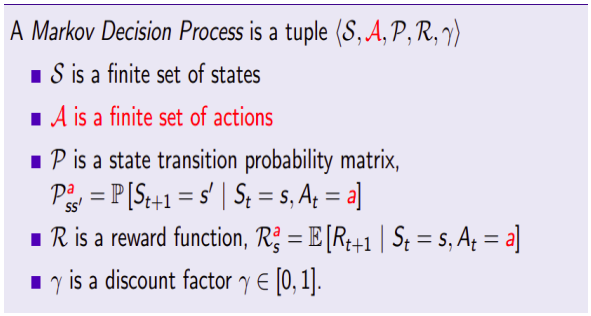


FIGURE 9
MARKOV DECISION PROCESS IN A TUPLE-FORMAT. IMAGE CREDITS: UCL RL COURSE BY DAVID SILVER

- Policy: Denoted by $\pi(a_t|s_t)$, a policy is what the agent tries to learn. It is a function mapping from states to actions (or distribution over actions). The objective of the agent is to find an optimal policy that maximizes its reward in the environment.
- Value function: This gives a notion of how good a certain state and action is. The state value-function ($V(s_t)$) denotes the expected return (discounted sum of all rewards in the future) from a certain state and then following the same policy afterward. The action value-function ($Q(s_t, a_t)$) denotes the expected return (discounted sum of all rewards in the future) from a certain state, upon taking an action and then following the same policy afterward. It is also called the Q-function.

In the last decade, there have been huge improvements in the field of Deep reinforcement learning, which is a combination of deep learning and reinforcement learning (thanks to powerful-capacity models like Neural Networks, that can behave as universal function approximators). To mention a few, deep RL has been applied in games (like Go [20], chess [21], Dota [22], Starcraft [23], atari [24], etc.), online advertising [25], applications in medicine like protein folding [26], robotics [27], etc.

To focus a bit more on deep RL, it is a way to bring reinforcement learning to real-world applications. Traditional tabular RL scales very poorly with the magnitude of the state and action space that we encounter for real-world agents. Deep learning provides a solution to this problem. The core idea comes from function approximators. In RL, the goal of the agent is to find an optimal policy, which is nothing but a function. Hence deep learning function approximators like Neural networks can be used to approximate the policy and the value function. This provides us an efficient and effective way to scale to nearly infinitely large state-action spaces.

II. Combining search and learning

In our work of program synthesis, we aim to use ML methods to combine search and learning techniques to get the best of both worlds. This approach to program synthesis is described as follows:

Within the Programming Languages (PL) community, program synthesis is typically solved using *enumerative search*, finding correct programs for a given specification by naively enumerating candidates until a satisfying program is found. Various methods have been devised to do this efficiently and scalably. However, using traditional search-based techniques is often insufficient for achieving good performance in terms of computational complexity, resources, etc. The program search space for program synthesis is generally huge. The Machine learning community has treated the synthesis typically as a supervised-learning problem (described in the deep learning-based methods before). Using learning-based methods (from deep reinforcement learning), we can identify certain attributes or features from the input-output specification, which can be used to narrow down or rank different programs to choose the most suitable program.

Traditionally, program synthesis has been a task in the domain of programming languages. As mentioned before, the *enumerative search* has been a widely used method. In practice, this approach is made tractable by narrowing the search space by integrating deductive components into the search process [28] or modifying the language of interest into an equivalent language with a narrower search space and searching within that space [28].

Researchers in the machine learning community have approached this problem from a different direction - rather than searching naively through a restricted space, correct programs can be efficiently found within a more extensive search space by intelligently searching or sampling from that space using a learned model of how specifications map to programs. Most of these methods use a *supervised learning* approach - training on a large synthetic dataset of input/output examples and corresponding satisfying programs, then using Recurrent Neural Network (RNN)-based models to output each line in the target program successively given a corresponding set of input/output examples [11-12, 29-30].

However, most existing work in the machine learning community heavily relies on the domain-specific language (DSL) being used for synthesis. This hinders the program synthesis tool from generalizing to other languages. This is a legitimate problem as even if the syntax and rules of the programming language change, the underlying logic, data structures, and concepts of high-level languages don't. Such supervised learning methods often require language features such as full differentiability of the language [29-30] or a large training set of specification and satisfying program pairs [11-12]. These requirements make it challenging to combine existing machine learning-based methods with techniques developed by the PL community, which require very different properties in a DSL for effective synthesis. Further, most existing methods have heavily focused on solving programs in a "one-shot" fashion, either successfully outputting a satisfying program for a given specification in a single or small number of attempts or failing to do so, which scales poorly as program spaces become large. In contrast, as search-based methods enumerate all programs, they can eventually solve any synthesis task, but they may take infeasible amounts of time.

A solution to this problem is to combine both of these methods. *Search*, and *Learning* are the two most important components behind the recent successes in Artificial Intelligence and will continue to be [31]. They are complementary to each other. Progress in one can be used for developing better methods in the second. As a result of this, there has been little work in this area. [15] uses a priority-tree based search algorithm, augmented using reinforcement learning methods. The *searching* part can efficiently cut down the program search space by pruning irrelevant regions in the space. The *learning* part is used to assign priority and decide which part to omit and which to explore.

III. Program synthesis through Reinforcement Learning Guided Tree Search (RLGTS)

One of the works that discuss a combination of search and learning is by Simmons-Edler et al. [15]. They propose an approach, representing the task of synthesizing a given program as a Markov decision process solvable via reinforcement learning. From observations about the states of partial programs, they attempt to find a program that is optimal over a provided reward metric on pairs of programs and states. They instantiate this approach on a subset of the RISC-V assembly language operating on floating-point numbers, and as an optimization inspired by

search-based techniques from the PL community, they combine RL with a priority search tree. They evaluate this instantiation and demonstrate the effectiveness of our combined method compared to a variety of baselines, including a pure RL ablation and a state of the art Markov chain Monte Carlo search method on this task.

The problem of synthesizing programs is instantiated in the form of a Markov decision process (MDP) and reinforcement learning is used to learn to solve a program given only a set of input/output examples for that program, a language specification, and a reward function for the quality of a given program.

In this RL-based approach, the program state and the current partial program is interpreted as an environment, and lines of code in the language as actions seeking to maximize the reward function. Furthermore, the RL model is combined with a tree-based search technique which dramatically improves the performance of the method. This combination helps address issues of local minima and efficient sampling which arise in many RL applications.

RLGTS does not depend on the availability of training data for a given programming language and makes no assumptions about the structure of the language other than that the language allows for partial programs to be executed and evaluated. Further, this RL-based approach can be combined with other program synthesis methods easily and naturally, allowing for users to benefit from the extensive work on search-based synthesis available for some domains.

1. The synthesis model**:** Program synthesis is represented as follows - A program of length T is a set of actions $P_T = \{a_1, a_2, , a_T\}$. Each action $a_t \epsilon \{1, 2, , T\}$ represents a single line of code applied to the state s_t, which is the memory state, i.e., the values of all the variables after the execution of all previous lines of code $\{a_1, a_2, , a_{T-1}\}$, applied to a set of initial variable assignments. The agent's objective is to output the next line of code a_t given s_t, such that the final program P_T will maximize a user-provided cumulative reward function R_T on a given small set of input and output examples (typically 5-10 examples). Note that the number of input-output examples for each task is very less as compared to the examples required in supervised deep learning techniques. This is one of the benefits of combining search and learning.

The reward function combines correctness (distance from the current state to the output state) and the program length as a metric for computational complexity, but this could easily be extended to include terms optimizing for properties like power consumption, memory usage, network/filesystem IO, or any other desired attribute.

2. Reward function**:** The reward function has two main components:

(i) Correctness: This measures the correctness of the next state s_{t+1}, given a set of N input-output pairs. It is mathematically defined as follows -

$$r_{\text{correctness}}(s_t, a_t) = \frac{\lambda_{\text{correctness}}}{NM} \sum_{j=0}^{N} \sum_{k=0}^{M} \frac{|O[j][k] - s_{t+1}[j][k]|}{|O[j][k]|}$$

FIGURE 10.
COMPONENT OF THE REWARD DENOTING THE DISTANCE FROM THE TARGET EXAMPLES.
IMAGE CREDITS [15]

N denotes the number of input-output examples, M denotes the number of variables used in each of the program examples. O is a matrix, each element of which is the state of the k^{th} variable in the j^{th} program example. $s_{t+1}[j][k]$ denotes the next state output by the RL algorithm. $\lambda_{correctness}$ is a hyperparameter weight on this component of the reward function. This component, $r_{correctness}$ is basically the distance between the correct desired program and the output of our algorithm. Hence it should be as little as possible.

(ii) Efficiency: This term penalizes encouraging the agent to learn shorter programs as an approximation of program computational efficiency.

$$r_{\text{efficiency}}(s_t, a_t) = |P_t| + 1$$

FIGURE 11
COMPONENT OF THE REWARD DENOTING THE EFFICIENCY DETERMINED BY THE NUMBERS OF LINES OF CODE. IMAGE CREDITS: [15]

$|P_t|$ here is the length of the program. One is added to ensure that the denominator does not become 0.

Finally, we define the reward function as follows:

$$r(s_t, a_t) = \frac{\lambda_{\text{scale}}}{r_{\text{correctness}}(s_t, a_t) + r_{\text{efficiency}}(s_t, a_t)}$$

FIGURE 12.
FINAL REWARD FUNCTION EXPRESSED AS A COMBINATION OF THE CORRECTNESS AND EFFICIENCY REWARD. NOTE THAT THE FINAL REWARD IS INVERSELY PROPORTIONAL TO THE DISTANCE AND THE LENGTH OF THE PROGRAM. THIS IS IN ORDER TO PRODUCE PROGRAMS THAT PRODUCE OUTPUTS CLOSER TO THE GIVEN OUTPUT EXAMPLES AND PROGRAMS SHORT IN LENGTH. IMAGE CREDITS: [15]

λ_{scale} is a hyperparameter. We combine all the terms above to get the reward function, which is inversely proportional to the correctness and efficiency rewards.

The reward function described above uses *reward shaping* and can be modified according to the user requirements to include other terms to measure correctness and efficiency. Also, note that our MDP formulation is generic and can be applied to a wide range of reward functions, such as the one above.

3. Main Reinforcement learning model: Here we describe the function approximator (neural network) architecture and the reinforcement learning algorithm used in RLGTS.

We use the Deep double Q-learning algorithm [32] as our RL algorithm. It minimizes the following loss function:

$$\delta_t = \|r(s_t, a_t) + \gamma Q_{\theta'}(s_{t+1}, \text{argmax}_{a'} Q_\theta(s_{t+1}, a')) - Q_\theta(s_t, a_t)\|^2$$

FIGURE 13.
STANDARD BELLMAN ERROR [33] CLUBBED WITH A DOUBLING STRATEGY (USING THE BASE NETWORK TO FIND THE MAXIMIZING ACTION AND THE TARGET NETWORK TO EVALUATE ITS VALUE). WE AIM TO MINIMIZE THIS OBJECTIVE. IMAGE CREDITS: [15]

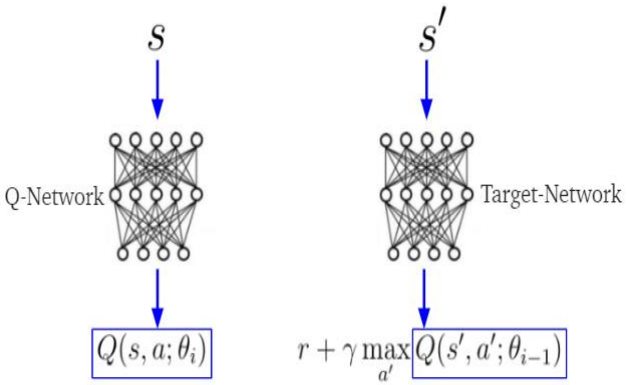


FIGURE 14
BASE Q-NETWORK (LEFT) AND TARGET Q-NETWORK (RIGHT). IMAGE CREDITS: TOWARDSDATASCIENCE.COM

Algorithm 3: Deep double Q-learning

1: Initialize replay memory D to capacity N
2: Base action-value function Q with weights θ
3: Target action-value function Q' with weights θ'
4: **for** episode = 1 to M **do**
5: Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
6: **for** t = 1, N **do**
7: With probability ϵ select a random action a_t
8: otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
9: Execute action a_t in emulator and observe reward r_t and image x_{t+1}
10: Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
11: Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
12: Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
13: if ϕ_{j+1} **is** terminal
14: $y_j = r_j$
15: else
16: $y_j = r_j + \gamma\, Q'(\phi_{j+1}, a'\, Q(\phi_{j+1}, a'; \theta); \theta')$

17: Perform a gradient descent (or Adam) step on $(y_i - Q(\phi_j, a_j; \theta))^2$ with respect to the
18: network parameters θ
19: Every C reset $Q' = Q$
20: **End for**
21: **End for**

Source [32]

We use an ϵ-greedy policy to balance exploration and exploitation. The objective function trained using the above equation predicts the expected future reward that will result from taking action $a\epsilon\{a_1, a_2, , a_n\}$ from state s_t.

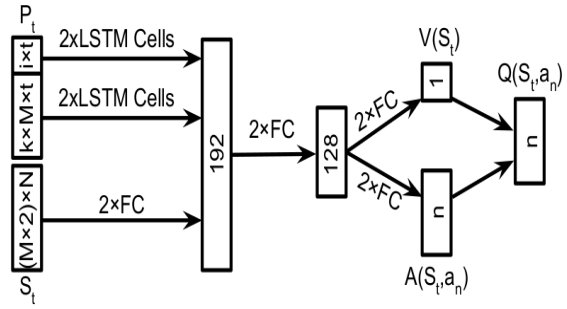


FIGURE 15.
NEURAL NETWORK ARCHITECTURE TO BE USED. IMAGE CREDITS: [15]

Above is the Neural network architecture to be used. It consists of the state input s_t, which encodes the current and target values for each variable and a sequence of one-hot vectors encoding previous lines of the program that produced s_t, P_t. The s_t module is a two-layer fully-connected (FC) neural network, while the P_t module is a two-layer LSTM [34]. These modules are concatenated and fed to more FC-layers followed by a two-layer value stream computing $V_\theta(s_t)$, the expected future reward value from being in the current state, and a separate two-layer advantage stream computing $A_\theta(s_t, a_t)$, the advantage of each action $a\epsilon\{1, ..., n\}$. These modules are combined to give the action-value function $Q_\theta(s_t, a_t)$. We use ReLU non-linearities in the FC-layers and sigmoid and tanh activation functions in the LSTM-layers.

4. Tree search: The action-value function ($Q(s_t, a_t)$) is obtained from the RL model described above. This function gives the quality estimate of a state-action pair and subsequently following the same policy. Hence this is the learning part of RLGTS.

To combine this with search-based techniques (in order to cut down to the program search space), we combine the Q-function learning mechanism with a simple prioritized tree search algorithm to aid in an efficient exploration of the program space.

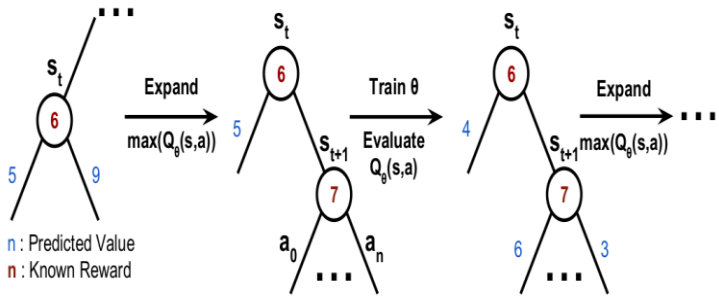


FIGURE 16.
PRIORITY-BASED TREE SEARCH USING THE Q-FUNCTION. IMAGE CREDITS: [15]

The tree starts from the initial state represented by the root of the tree. The root is therefore the initial values of the variables in the program. The rest of the nodes represent other possible states. The edges from a node represent the n possible actions that can be taken from that state. Hence the objective of the search algorithm is to travel from the root to the node corresponding to a terminal state (example - return or exit statement in C-language) by taking a sequence of actions that denote the lines in the code.

But how to decide which edge to pick from a node? In other words, how to decide what action to take from a state? We use our Q-function here. The task of this Q-function is to predict a Q-value (or score) for each unexplored edge (action) of the tree. The search algorithm then simply expands the edge corresponding to the maximal Q-value out of the n values available. The Q-

value is determined by the Neural network we described above. The action (line of code) corresponding to the expanded edge is then added to the program P_t. This process is done by alternating between sampling the unexplored edges (action) via ϵ-greedy strategy and training the Q-network.

The worst-case performance of this *search+learning* approach is guaranteed to be that of traditional *enumerative search*, given the memory and computational requirements. This is because this method encourages exploration by evaluating each unique state-action pair only once. This approach also allows reinforcement learning-based synthesis to be combined easily with other search-based synthesis methods using deductive reasoning and search tree pruning for domain-specific improvements.

Algorithm 4: Priority-based tree search

1: **Input**: Parameters of the Q-function θ
2: Initialize root node of the search tree as the initial values of the variables in the program
3: intermediate_program = []
4: **for** 1, N **do**
5: With probability ϵ select a random action a_t (means select a random line of code out of the
6: available)
7: otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$ (line of code maximizing the Q-value)
8: intermediate_program.add(a_t)
9: **return** intermediate_program

Algorithm 5: Reinforcement learning with guided priority tree search algorithm

1: **Input**: M pairs of input-output examples
2: **Input**: Natural language description of the task
3: **Input**: Synthesis language
4: Initialize base action-value function Q with weights θ
5: Initialize target action-value function Q' with weights θ'
6: output_nlp = **Invoke** Natural language processor
7: **while True**
8: program_list = []
9: θ = **Invoke** Algorithm 3 (output_nlp, M examples)
10: **for** 1, K **do**
11: intermediate_program = **Invoke** Algorithm 4 (θ)
12: program_list.**add**(intermediate_program)
13: r_{net} = Evaluate program_list based on reward structure
14: if $r_{net} > r_{threshold}$
15: **break**
16: synthesized_program = **Invoke** Algorithm 4 (θ)
17: **return** synthesized_program

IV. Combining Monte Carlo Tree Search (MCTS) with reinforcement learning

Here we present the combination of reinforcement learning-based program synthesis methods with Monte Carlo Tree Search (MCTS) [35]. MCTS is a popular heuristic search tree-based algorithm, used widely for the various sequential decision-making processes, making it a good fit for RL. We keep the learning-process the same as described before and just change the search algorithm. MCTS is a more efficient and effective algorithm as compared to the simple priority tree search algorithm described before.

MCTS has also been successfully used in several significant works in AI. One of the classic applications of MCTS is in AlphaGo [36], the Go-playing agent that defeated the World champion in the Game of Go in 2016 [20]. Similar to program synthesis, the search space of Go is huge. There can be enormous possible strategies that a player can adopt. Like us, AlphaGo also uses the idea of combining search and learning, where they use MCTS as the search algorithm and supervised and reinforcement learning-based policy and value function

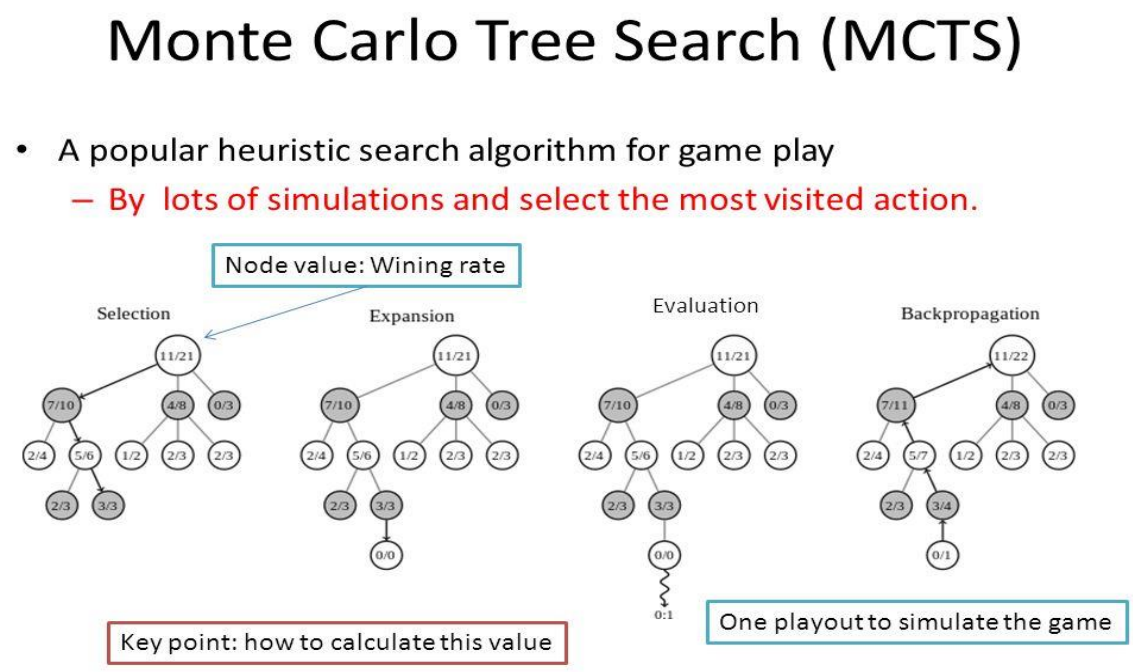
The search algorithm is described below.

Here we use the term "game" for the task of synthesizing a program.

MCTS consists of 4 phases:

1. Selection: MCTS simulates many games to find out how good are the moves (lines of code). The purpose of this selection step is to prioritize the moves for future simulations. If our priority is correct, we can pick good and appropriate lines of code in fewer games. Let us assume the tree at some point in time (how to build and expand the tree will be discussed in the Expansion section). We start from the root and select a path until a leaf node is encountered. This path involves choosing the nodes of the tree via some method that balances the exploration and exploitation of different parts of the program space. One of the good methods used for selecting the nodes to build a path is the Upper Confidence Bound 1 (UCB1) [37]. When used with MCTS, it is known as UCT (Upper Confidence applied to Trees). The UCB1 selection function is -

FIGURE 17. MONTE CARLO TREE SEARCH ALGORITHM.



Neural networks.

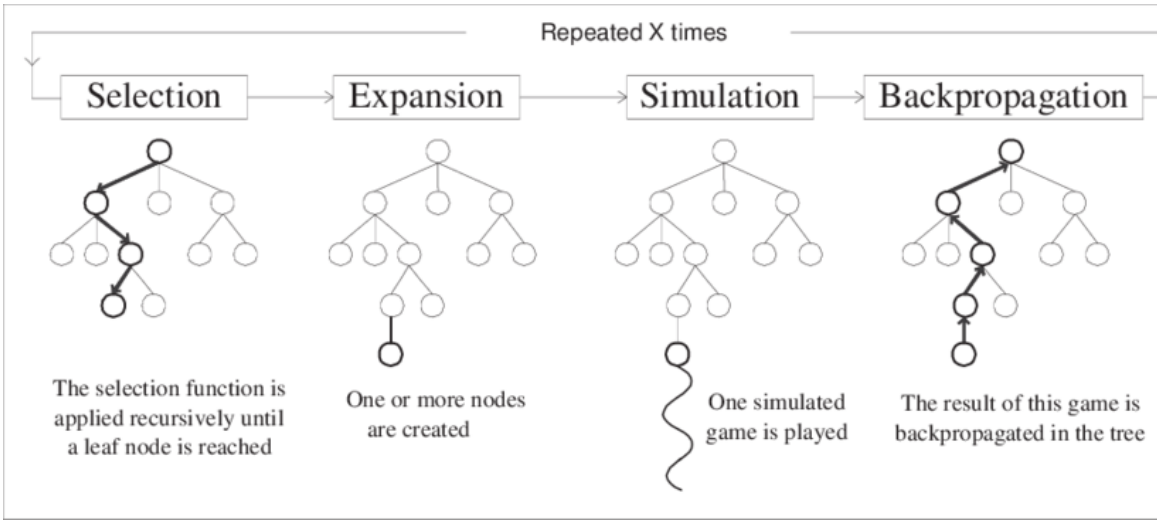


FIGURE 18
FORCE PHASES OF MCTS ALGORITHM. IMAGE CREDITS: RESEARCHGATE.NET

$$a_t = \underset{a}{\operatorname{argmax}}(Q(s_t,a) + u(s_t,a))$$

$$u(s,a) \propto \frac{P(s,a)}{1+N(s,a)}$$

$$P(s,a) = p_\sigma(a|s)$$

$$N(s,a) = \sum_{i=1}^{n} 1(s,a,i)$$

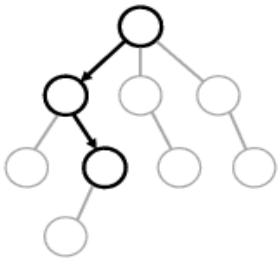
$$V(s_L) = (1-\lambda)v_\theta(s_L) + \lambda z_L$$

$$Q(s,a) = \frac{1}{N(s,a)} \sum_{i=1}^{n} 1(s,a,i)V(s_L^i)$$

$p_\sigma(a|s)$ - From the policy network: how good to take action a.
$v_\theta(s_L)$ - From the value network: how good to be in positions s_l
$N(s,a)$ - How many times have we select action a so far.
z_L - the previous simulated game result.

FIGURE 19.
UCB1 ALGORITHM ADAPTED TO MCTS WITH REINFORCEMENT LEARNING.
IMAGE CREDITS: towardsdatascience.com

This function uses Q-term for exploitation and the u-term for exploration. The Q-term is determined by the learned policy and value networks. The u-term denotes the win-rate (means how often our algorithm is able to synthesize the desired program).

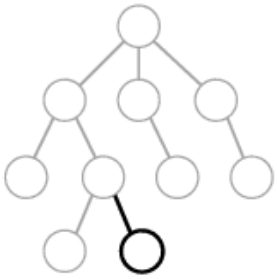


(a) Selection

FIGURE 20.
SELECTION STEP IN MCTS
IMAGE CREDITS: towardsdatascience.com

We use this UCB1 function at each node to pick the edge which maximizes the value of Q+u. This is done until a leaf node is reached.

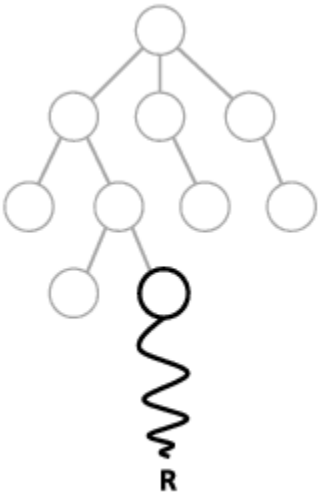
2. Expansion: This phase involves expanding the tree by initializing a new node below the leaf node selected in the previous phase. The edge used for expansion is randomly generated. The value of the added node is computed using our value function network. The statistics of this node (as in the UCB1 function) are initialized to 0.



(b) Expansion

FIGURE 21
EXPANSION STEP IN MCTS
IMAGE CREDITS: towardsdatascience.com

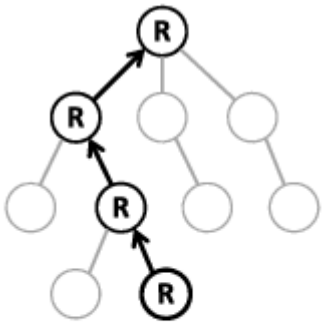
3. Simulation: In the simulation phase, we simulate the rest of the game using **Monte Carlo rollout** starting from the leaf node (generated in the Expansion phase). It means finish playing a game using a policy and find out whether you win or lose. Winning corresponds to successfully synthesizing a program and losing corresponds to failing to do so. This can be realized by some sort of a threshold on the total reward received in the process of simulation.



(c) Simulation

FIGURE 22.
SIMULATION STEP IN MCTS
IMAGE CREDITS: towardsdatascience.com

4. Backpropagation: Once the outcome of the simulation is decided, we back-propagate the path to reach the starting root node. On the way, all the statistics of all the nodes visited are updated based on the final outcome of the simulation.



(d) Backpropagation

FIGURE 24
BACKPROPAGATION STEP IN MCTS
IMAGE CREDITS: towardsdatascience.com

This completes 1 round of MCTS. The search tree generated after this round is then re-used for the next round. After many rounds, finally, we get a robust search+learning algorithm that can efficiently synthesize a program, given the input specifications.

RESULTS

The following are some implementation examples of program synthesis. This is based on the work by Illia Polosukhin, Maksym Zavershynskyi and Richard Shin, nearai/program_synthesis, June 2018.

Experimental details:
The model trains on a set of around 80000 data samples (from https://github.com/nearai/program_synthesis) where each data sample has the natural language description of the problem in hand, 5-10 input-output examples where the input is a dictionary of key-value pairs where each key refers to the variable (an integer or an array) and its value refers to its numeric value/s. The output may be an integer or an array. Some additional inputs like return type and argument types are also given. The output is code obtained from a syntax tree developed

using the Seq2Tree model. This tree is evaluated against the actual tree for the problem during training.

I. Example 1:

Input-Output Examples, given as input to the program synthesis model

Input	{"a": [7, 16, 1, 4, 21, 6, 6, 14, 23, 5, 2, 13, 4, 19, 22, 22, 6, 27, 12, 14, 6, 9, 10], "c": 15, "b": 8}
Output	4
Input	{"a": [22, 25, 14, 26, 21, 8, 28, 7, 19, 30, 15, 27, 11, 22, 26, 8, 27, 19, 2, 7, 19, 6, 7, 6, 3, 27, 16, 16, 20, 30, 30, 17, 29, 7, 30], "c": 16, "b": 29}},
Output	1
Input	{"a": [9, 4, 28, 4, 25, 8, 8, 14, 26, 13, 1, 5, 2, 10, 28, 26, 12, 10, 20, 3], "c": 1, "b": 29}
Output	1
Input	{"a": [12, 16, 4, 21, 19, 19, 11, 6, 6, 18, 4, 20, 29, 29, 7, 6, 30, 14, 11, 20, 12, 19], "c": 9, "b": 16}
Output	1
Input	{"a": [22, 17, 27, 2, 17, 27, 10, 16, 5, 6, 16, 17, 7, 25, 21, 12, 1, 15, 23, 17, 1, 12], "c": 25, "b": 13}},
Output	21
Input	{"a": [11, 26, 27, 8, 18, 11, 1, 26, 17, 16, 17, 4, 5, 29, 19, 15, 20, 17, 29, 19, 22, 6, 1, 15, 28, 27, 1, 15, 9, 6], "c": 17, "b": 2}},
Output	4
Input	{"a": [2, 30, 23, 26, 14, 17, 25, 3, 17, 12, 12, 11, 16, 13, 13, 27, 6, 2, 25, 3, 23, 23, 20, 19, 3], "c": 23, "b": 14}
Output	3
Input	{"a": [28, 4, 14, 2, 12, 13, 9, 27, 2, 16, 15, 20, 28, 30, 23, 22, 13, 15, 11, 25, 25, 4, 15, 25, 17, 29, 29], "c": 18, "b": 20}
Output	1
Input	{"a": [28, 4, 14, 2, 12, 13, 9, 27, 2, 16, 15, 20, 28, 30, 23, 22, 13, 15, 11, 25, 25, 4, 15, 25, 17, 29, 29], "c": 18, "b": 20}
Output	1
Input	{"a": [6, 24, 6, 4, 20, 26, 29, 22, 8, 25, 25, 10, 2, 29, 1, 22, 9, 16, 28, 28, 9, 27, 11, 10, 29, 18, 14, 25, 7, 17, 13, 5, 6, 3, 25, 13, 16, 18, 28], "c": 23, "b": 12}
Output	2

Natural Language Description:

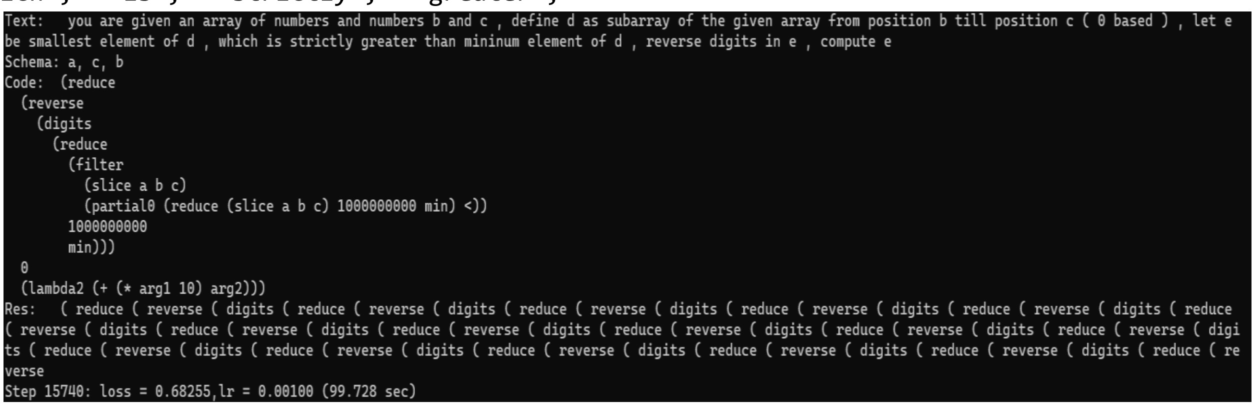
"*you are given an array of numbers and numbers b and c , define d as subarray of the given array from position b till position c (0 based) , let e be smallest element of d , which is strictly greater than minimum element of d , reverse digits in e , compute e*"

This description is tokenized as:

"text": ["you", "are", "given", "an", "array", "of", "numbers", "and", "numbers", "b", "and", "c", ",", "define", "d", "as", "subarray", "of", "the", "given", "array", "from", "position", "b", "till", "position", "c", "(", "0", "based", ")", ",", "let", "e", "be", "smallest", "element", "of", "d", ",", "which", "is", "strictly", "greater", "than", "minimum", "element", "of", "d", ",", "reverse", "digits", "in", "e", ",", "compute", "e"]

Other Inputs:

"return_type": "int",
"args": {"a": "int[]", "c": "int", "b": "int"},
"nodes": ["l1_reverse_number"]



```
Text:   you are given an array of numbers and numbers b and c , define d as subarray of the given array from position b till position c ( 0 based ) , let e
be smallest element of d , which is strictly greater than mininum element of d , reverse digits in e , compute e
Schema: a, c, b
Code:  (reduce
  (reverse
    (digits
      (reduce
        (filter
          (slice a b c)
          (partial0 (reduce (slice a b c) 1000000000 min) <))
        1000000000
        min)))
  0
  (lambda2 (+ (* arg1 10) arg2)))
Res:   ( reduce ( reverse ( digits ( reduce ( reverse ( digits ( reduce ( reverse ( digits ( reduce ( reverse ( digits ( reduce ( reverse ( digits ( reduce
( reverse ( digits ( reduce ( reverse ( digits ( reduce ( reverse ( digits ( reduce ( reverse ( digits ( reduce ( reverse ( digits ( reduce ( reverse ( digi
ts ( reduce ( reverse ( digits ( reduce ( reverse ( digits ( reduce ( reverse ( digits ( reduce ( reverse ( digits ( reduce ( reverse ( digits ( reduce ( re
verse
Step 15740: loss = 0.68255,lr = 0.00100 (99.728 sec)
```

FIGURE 24

OUTPUT SCREENSHOT

II. Example 2

Input	{"a": [7, 14, 2, 9, 20, 2, 24, 27, 13, 29, 26, 1, 26, 24, 12, 3, 9, 11, 18, 20, 4, 7, 15, 3, 4, 6, 2, 18, 27, 8, 6, 30, 12, 10, 13, 25], "c": [14, 21, 29, 21, 14, 26, 29, 6, 27, 12, 27, 9, 18], "b": [24, 8, 29, 22, 4, 1, 1, 3, 1, 27, 13, 27, 30, 9], "d": 8}
Output	[]
Input	{"a": [16, 16, 7, 14, 10, 19, 23, 19, 17, 13, 22, 17, 30, 25, 30, 5, 5, 30, 15, 12, 20, 17], "c": [23, 26, 19, 28, 18, 12, 28, 28, 10, 4, 29, 2], "b": [10, 23, 19, 22, 11, 3, 14, 2], "d": 27}
Output	[16, 16, 7, 14]
Input	{"a": [19, 2, 7, 21, 5, 14, 27, 8, 28, 14, 30, 28, 22, 30, 9, 7, 28, 15, 10, 24, 17, 11, 3, 24, 23, 20, 2, 6], "c": [3, 10, 29, 26, 6, 24, 16, 3, 29, 21, 15, 23, 26, 8, 24], "b": [8, 3, 1, 6, 13, 19], "d": 27}
Output	[7, 21]
Input	{"a": [9, 4, 29, 3, 4, 12, 18, 14, 5, 14, 10, 12, 20, 4, 24, 18, 23, 2, 21, 16, 26, 15,

	13, 25, 30, 11, 12, 18, 23, 28, 25, 24, 12, 8, 22, 22], "c": [23, 10, 19, 4, 24, 5, 17, 19, 11], "b": [23, 18, 18, 20, 2, 20, 26, 4, 2], "d": 30}
Output	[]
Input	{"a": [2, 7, 28, 2, 24, 18, 18, 17, 12, 6, 24, 28, 25, 23, 23, 8, 4, 28, 30, 10, 6, 10, 3, 16, 18, 20, 2, 20, 16, 11, 22, 15, 16, 8, 14, 20, 27, 24], "c": [1, 29, 5, 10, 3, 6, 14], "b": [11, 22, 23, 29, 17, 3, 4, 10, 3, 27, 1, 5, 19], "d": 10}
Output	[]
Input	{"a": [28, 7, 15, 6, 12, 12, 5, 20, 21, 15, 29, 20, 10, 23, 11, 26, 17, 10, 10, 13, 4, 14, 24], "c": [6, 21, 18, 13, 12, 16, 30, 4, 2, 28], "b": [2, 21, 21, 11, 3], "d": 19}
Output	[7, 15, 6, 12, 12, 5, 20, 21, 15]
Input	{"a": [18, 16, 7, 22, 30, 6, 9, 9, 15, 6, 20, 5, 10, 26, 4, 15, 6, 2, 3, 26, 12, 22, 9, 5, 18, 17, 8, 16, 16, 21, 7, 13, 3, 29, 2, 2, 15, 3, 16, 15], "c": [26, 2, 9, 21, 10, 23, 14, 30, 15], "b": [7, 14, 20, 10, 23, 22, 21, 12, 8, 30, 18, 14, 24, 18, 30],
	"d": 25}
Output	[]
Input	{"a": [27, 26, 9, 29, 5, 6, 2, 6, 29, 27, 25, 1, 5, 27, 11, 20, 16, 4, 25, 11, 19, 1, 5, 24, 29, 24, 8, 16, 2, 18, 13, 30, 1, 27, 7, 10, 2], "c": [26, 7, 5, 15, 21, 9, 5, 23, 5, 1], "b": [29, 15, 22, 18, 25, 2], "d": 13}
Output	[]
Input	{"a": [17, 2, 27, 23, 27, 3, 8, 30, 26, 20, 8, 23, 21, 19, 7, 7, 14, 13, 5, 12, 11, 3, 15, 9, 12, 24, 16, 6, 4, 22], "c": [28, 9, 30, 28, 27, 30, 9], "b": [18, 28, 3, 18, 21, 7, 17, 10, 27, 16], "d": 7}
Output	[27, 23]
Input	{"a": [15, 28, 30, 4, 4, 18, 6, 12, 10, 18, 15, 20, 8, 30, 23, 11, 28, 8, 8, 24, 16, 18, 17, 20, 9, 24, 28, 26, 8, 15, 3, 19, 28, 16, 18], "c": [25, 4, 2, 21, 17, 22, 26, 10, 1, 3, 18, 30], "b": [4, 26, 23, 26, 10, 10, 22, 21, 29, 24, 5, 11], "d": 29}}
Output	[]

Natural Language Description:

"*Consider arrays of numbers a , c and b and a number d , define e as values of a from position how many indexes there are on which the value in c is smaller than the value in b till position such number that the sum of one and that number multiplied by d is prime (the first number has index 0) , your task is to find e*"

This description is tokenized as:

```
"text": ["consider", "arrays", "of", "numbers", "a", ",", "c", "and", "b", "and", "a", "number", "d", ",", "define", "e", "as", "values", "of", "a", "from", "position", "how", "many", "indexes", "there", "are", "on", "which", "the", "value", "in", "c", "is", "smaller", "than", "the", "value", "in", "b", "till", "position", "such", "number", "that", "the", "sum", "of", "one", "and", "that", "number", "multiplied", "by", "d", "is", "prime", "(", "the", "first", "number", "has", "index", "0", ")", ",", "your", "task", "is", "to", "find", "e"]
```

Other inputs:

```
"return_type": "int[]",
"args": {"a": "int[]", "c": "int[]", "b": "int[]", "d": "int"},
"nodes": ["basic_slice_int_array"]
```

```
Text:   consider arrays of numbers a , c and b and a number d , define e as values of a from position how many indexes there are on which the value in c is
smaller than the value in b till position such number that the sum of one and that number multiplied by d is prime ( the first number has index 0 ) , your t
ask is to find e
Schema: a, c, b, d
Code:  (slice
  a
  (reduce
    (map
      (range 0 (min (len c) (len b)))
      (lambda1 (if (< (deref c arg1) (deref b arg1)) 1 0)))
    0
    +)
  (head
    (filter
      (range 1 40)
      (combine is_prime (combine (partial0 1 +) (partial0 d *))))))
Res:   ( filter ( slice a ( lambda1 ( reduce ( map b ( partial0 arg1 == ) ) false || ) ) ) ( partial1 c * ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) )
) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) )
Step 15840: loss = 0.66120,lr = 0.00100 (105.263 sec)
```

FIGURE. 25.
Output screenshot-2

MODELLING PROGRAM SYNTHESIS

Below we describe a few ways in which program synthesis can be modelled.

I. MVC-architecture

We can propose a Model-View-Controller (MVC) architecture for the program synthesis tool. MVC is a popular architecture in the field of software engineering and is increasingly adopted as a design for the world wide web applications in the major programming languages. The MVC architecture can largely speed up the time required to develop the software and also decrease the latency between the interactions at the server end. This is because the model and the view components are not directly dependent on each other. The functionality of each component is independent and the controller is used for interaction between them. This decreases the maintenance time as we can update, upgrade, and repair each component independent of all the others.

There are three components in the MVC design pattern:

1. Model: This part of the design pattern is the primary part and contains purely application information. It doesn't contain any information on how to show the data to the user. It is independent of the user interface. It controls the logic and rules of application. It also interacts with the database (if required in the application) to fetch relevant data to cater to requests from the controller.

2. View: This part helps the user to see the model's data. The main concern of this part is to access the model's data. The view section uses a chart, table, or diagrams to represent the information. It can also show similar data and use bar graphs and tables for different purposes. It is a visualization of information that the application contains. Multiple views of the same information are possible, such as a bar chart for management and a tabular view for accountants. The view module directly interacts with the user.

3. Controller: The controller establishes the communication between the model and the view components. Most of the dataflow and request flow work is done by the controller. It provides support for input and converts the input to commands for the application. The model and the view are indirectly connected, and the execution is reflected in the view part. The controller can also be used optionally to validate the incoming input from the user.

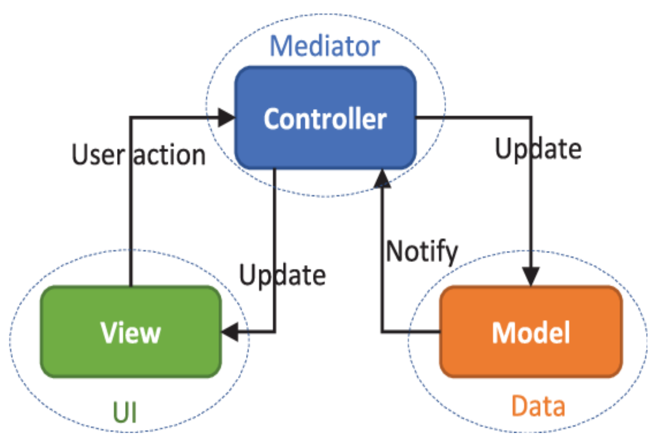


FIGURE. 26.
THE FIGURE SHOWS A TYPICAL MVC DESIGN. THE USER INTERACTS WITH THE VIEW (UI). THE VIEW INFORMS THE CONTROLLER ABOUT THE USER ACTION. THE CONTROLLER

updates or sends the required information to the model, which does all the algorithmic and data logic. The model then notifies and responds to the controller with the required output. The controller then updates the view with the output, which is then shown to the user. In this whole process, the controller acts as a mediator. Image credits: medium.com

In the above figure, the controller acts as a mediator. Following is an end-to-end instance of the complete MVC process starting from input by a user to the response from the software for our program synthesis task:

1. The user/client gives as input the data structure, natural language description, and the programming or domain-specific language. This would correspond to an input array, sorting description, and a programming language (say Python), respectively in the example above. These are given to the view, which passes them to the controller. The controller validates the request and sends it to the model.

2. First of all, (optionally) the model authenticates the user by looking into the database and also makes relevant changes to the database if the user requested so. If the authentication fails, it could invoke the controller, which then invokes the view to render the appropriate error message to the user. After user-verification, the model handles the task of synthesizing the program in the given language and applying the program to the data structure. It does this by first invoking the natural language processor to parse the input. The learning algorithm then formulates the problem as a reinforcement learning problem and outputs the Q-functions. These are then passed to the search algorithm, which searches for the appropriate actions (lines of code) by pruning the search space [15]. This toggling between learning the Q-function and searching through space (environment) is repeated for a few steps, until the cumulative reward for the program exceeds a threshold value. The final program is then taken. This completes the synthesis task.

3. The model then sends the response back to the controller. The controller then interacts with the view component to render the program and the output data structure to the user/client in a desirable format.

II SaaS Model

SaaS is a cloud computing model where software delivery and maintenance is done via a subscription model. Such software is directly hosted on the cloud and requires no installation, complicated configuration, or updates and is generally not sold with lifetime access. SaaS is generally seen at the top of the pyramid of cloud service models [38] which also include Platform as a Service (PaaS) and Infrastructure as a Service (IaaS), each of which have their own (possibly overlapping) target audiences.

SaaS applications can be used anywhere in the world, from any device and web browser sometimes even without an internet connection. Users also have access to the latest version of the software and don't have to bother about updates. Benefits like security, cost efficiency, reliability, scalability, portability and the "try before buy" paradigm have made SaaS applications extremely popular. SaaS applications have gained a lot of popularity in recent times due to the *win-win situation* for both the customer and vendor.

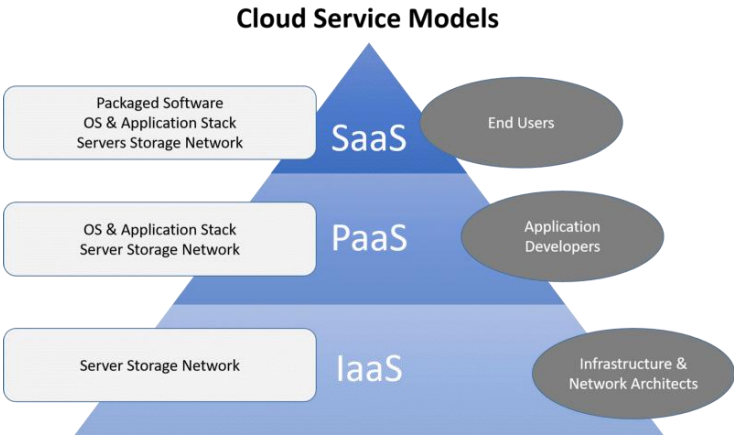


FIGURE 27.
CLOUD SERVICE MODELS PYRAMID
IMAGE CREDITS: IDM BLOG [38]

A multi-tenant cloud architecture with subscriptions based on the number of queries and the complexity of the model being used (and the associated computing resources) may be employed to deploy SaaS applications. A program synthesis tool deployed as a SaaS application helps both expert and non-expert programmers alike. Programmers increasingly work across libraries and programming languages, creating more complex systems

than ever before, and cannot memorize every detail of all the systems that must be used due to the mere complexity of these systems and the frequent changes made to these systems and frameworks in the interest of efficiency and security. Researchers and engineers working on program synthesis should consider deploying their service as an SaaS application to gain momentum for their products.

III. Model Driven

Model-driven software engineering (MDSE) is a software engineering paradigm that uses models to improve productivity, efficiency, interpretability, and communication. Models are basically more abstract representations of the system to be built. MDSE highlights and aims at abstract representations of the knowledge and activities that govern a particular application domain, rather than the computing (i.e. algorithmic) concepts. The MDE approach is meant to increase productivity by maximizing compatibility between systems, simplifying the process of design, and promoting communication between individuals and teams working on the system. A modeling paradigm for MDE is considered effective if its models make sense from the point of view of a user that is familiar with the domain, and if they can serve as a basis for implementing systems [39].

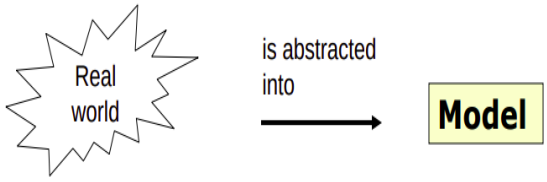


FIGURE 28.
ABSTRACTION OF THE REAL WORLD INTO A MODEL. IMAGE CREDITS: [39]

A model is an abstraction of something for the purpose of understanding it before building it [39]. Models are widely used in various engineering disciplines. For instance, modeling electrical circuit systems and the internal components represented in the form of a logic gate circuit.

Some important properties of models are:

- Models are abstractions from certain aspects of the real-world.
- They focus on certain aspects of the real-world that are more relevant, leaving the other open.
- Abstraction into models promotes the ability to analyze the properties of the system and understand the system better.

Model-driven development software is often referred to as a high-productivity platform as a service, given the unprecedented speed at which the users build and deploy new applications. This speed is derived from the use of models and other pre-built components that business and technical teams use to visually construct applications.

There are three main modules in fig. 2. of which we can construct a model:

1. Input-output examples and the corresponding probability distributions: In machine learning terminology, this corresponds to constructing $p(x)$ and $p(y|x;\theta)$. $p(x)$ is the input distribution from which the input examples are sampled. $p(y|x;\theta)$ is the distribution over labels, given the input x and the network parameters θ.

2. Natural language processing tool: This module converts the human-understandable description of the program to be synthesized into a form that a machine can understand. This is done via various natural language processing (NLP) techniques available.

3. Program synthesis tool: This is the main module that generates the program via machine learning algorithms. We have two types of approaches - deep learning and deep reinforcement learning-based. We will discuss them in detail in future sections.

Following is the model diagram for the reinforcement learning-based tree search algorithm:

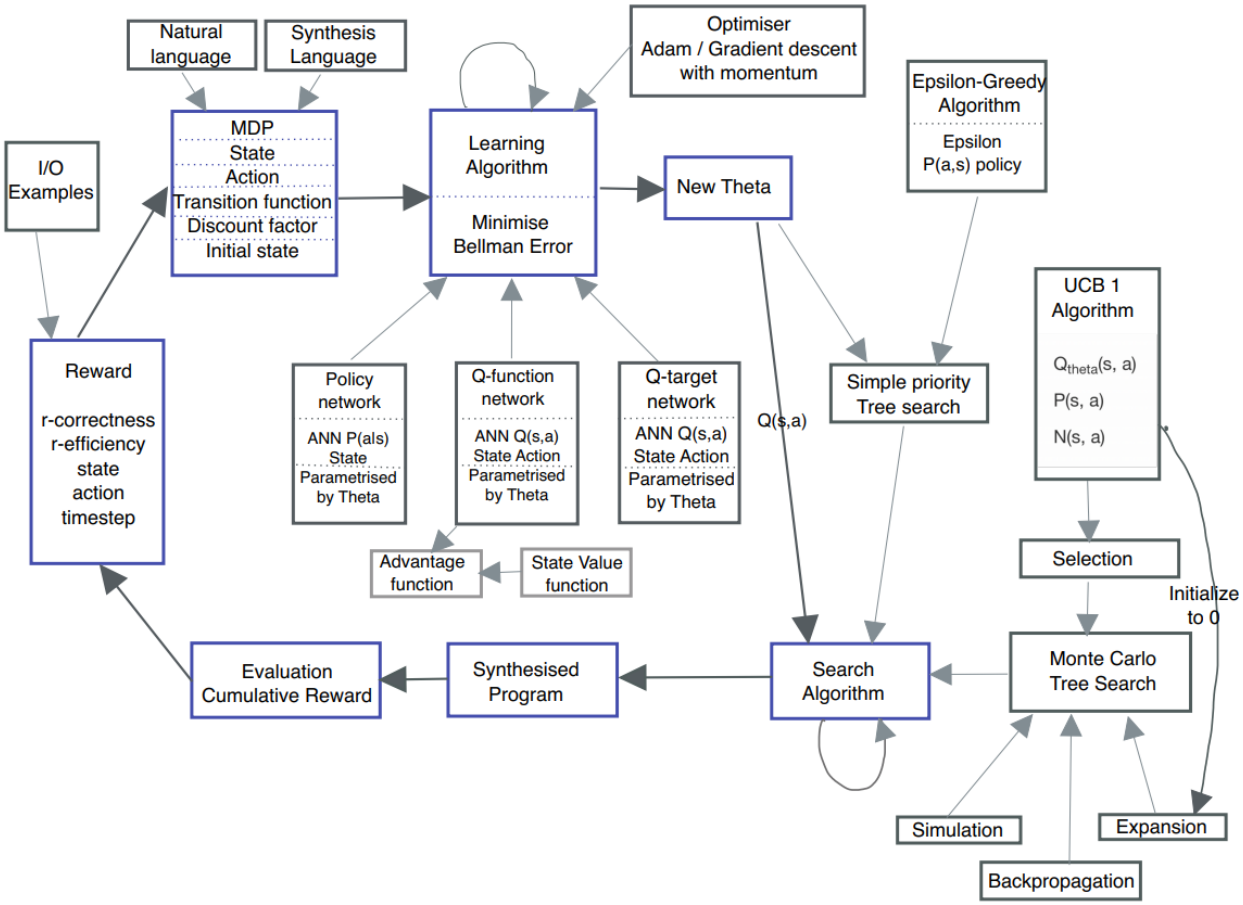


FIGURE 29.

MODEL DIAGRAM FOR REINFORCEMENT LEARNING-BASED TREE SEARCH

CONCLUSION

In this paper, we have incrementally gone over different ways to automate the task of program synthesis, ranging from traditional supervised learning to algorithms which power some of the most powerful AI programs written till date. We have seen how some of the most popular topics in deep learning like sequence-to-sequence models and attention, and in reinforcement learning like Markov Decision Processes and Monte-Carlo tree search are used in the context of program synthesis. We have also looked at an example of a program synthesis tool in actions. Finally we present different views on program synthesis including a Model-View-Controller approach and a Model driven approach to understand and represent program synthesis and propose a few arguments for deploying a program synthesis tool as Software as a Service (SaaS) application

.

REFERENCES

1. "Program Synthesis", slides by Alex Polozov, Microsoft Research AI. Link: https://www.microsoft.com/en-us/research/uploads/prod/2018/12/Program-Synthesis-SLIDES.pdf
2. Sumit Gulwani, Alex Polozov, Rishabh Singh, "Program Synthesis", *NOW*, Vol 7, August, 2017.
3. Shi, K., Bieber, D., and Singh, R. "Tf-coder: Program synthesis for tensor manipulations", arXiv:2003.09040v3 [cs.PL] Jul 2020.
4. Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. Qlose: Program Repair with Quantitative Objectives. In Computer Aided Verification (CAV). Springer-Verlag, 2016.
5. Barbara Jobstmann, Andreas Griesmayer, and Roderick Bloem. Program repair as a game. In Computer Aided Verification (CAV), pages 226–238. Springer-Verlag, 2005.
6. Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. SemFix: program repair via semantic analysis. In 35th International Conference on Software Engineering, ICSE, pages 772–781, 2013.
7. Aditya V. Nori, Sherjil Ozair, Sriram K. Rajamani, and Deepak Vijaykeerthy. Efficient synthesis of probabilistic programs. In ACM SIGPLAN Notices, volume 50, pages 208–217. ACM, 2015.
8. Pavol Cerný, Krishnendu Chatterjee, Thomas A. Henzinger, Arjun Radhakrishna, and Rohit Singh. Quantitative synthesis for concurrent programs. In Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings, pages 243–259, 2011.
9. Martin T. Vechev, Eran Yahav, and Greta Yorsh. Abstraction-guided synthesis of synchronization. In Proceedings of the 37th ACM SIGPLANSIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010, pages 327–338, 2010.
10. Michael Kuperstein, Martin T. Vechev, and Eran Yahav. Automatic inference of memory fences. In Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, October 20-23, pages 111–119, 2010.
11. J. Devlin, J. Uesato, et al. "RobustFill: Neural Program Learning Under Noisy I/O". In: *International Conference on Machine Learning (ICML)*, 2017.
12. M. Balog, A. Gaunt, et al. "Deepcoder: Learning to Write Programs". In: *International Conference on Learning Representations (ICLR)*, 2017.
13. Illia Polosukhin and Alexander Skidanov, "Neural Program Search: Solving Programming Tasks from Description and Examples", arXiv:1802.04335v1 [cs.AI], Feb 2018
14. Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain and Sumit Gulwani, "Neural-Guided Deductive Search for Real-Time Program Synthesis from Examples", In: *International Conference on Machine Learning (ICML)*, 2018.
15. Riley Simmons-Edler, Anders Miltner, and Sebastian Seung. Program synthesis through reinforcement learning guided tree search. arXiv preprint arXiv:1806.02932, 2018.
16. Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, Sailesh R, Subhajit Roy, Program Synthesis using Natural Language, In: Proceedings of the 38th International Conference on Software Engineering, May 2016..
17. Ilya Sutskever, Oriol Vinyals and Quoc V.Le Sequence to Sequence Learning with Neural Networks. In Proceedings of 27th International Conference on Neural Information Processing Systems - Volume 2 (NIPS), December 2014
18. Dzmitry Bahdanau et al. Neural Machine Translation by Jointly Learning to Align and Tranlate. In Proceedings of International Conference on Learning Representations (ICLR), May 2015.
19. Wikipedia: Reinforcement learning
20. https://deepmind.com/research/case-studies/alphago-the-story-so-far
21. https://deepmind.com/blog/article/alphago-zero-starting-scratch
22. https://openai.com/projects/five/
23. https://deepmind.com/blog/article/AlphaStar-Grandmaster-level-in-StarCraft-II-using-multi-agent-reinforcement-learning
24. https://deepmind.com/blog/article/Agent57-Outperforming-the-human-Atari-benchmark
25. Xiangyu Zhao, Changsheng Gu, Haoshenglun Zhang, Xiaobing Liu, Xiwang Yang, Jiliang Tang, "Deep Reinforcement Learning for Online Advertising in Recommender Systems", arXiv:1909.03602 [cs.IR], Sep, 2019.
26. https://deepmind.com/blog/article/AlphaFold-Using-AI-for-scientific-discovery
27. https://ai.googleblog.com/2018/06/scalable-deep-reinforcement-learning.html
28. Vu Le and Sumit Gulwani. FlashExtract: A framework for data extraction by examples. In Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14. ACM, 2014.
29. Rudy Bunel, Alban Desmaison, Pushmeet Kohli, Philip H.S. Torr, and M. Pawan Kumar. Adaptive neural compilation. In Proceedings of the 30th International Conference on Neural Information Processing Systems, NIPS'16, pages 1452–1460, USA, 2016. Curran Associates Inc. ISBN 978-1-5108-3881-9.
30. Alexander Gaunt, Marc Brockschmidt, Rishabh Singh, Nate Kushman, Pushmeet Kohli, Jonathan Taylor, and Daniel Tarlow. Terpret: A probabilistic programming language for program induction. 08 2016.
31. http://www.incompleteideas.net/IncIdeas/BitterLesson.html: The Bitter Lesson by Richard Sutton
32. Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In AAAI, volume 16, pages 2094–2100, 2016.
33. Mnih, Volodymyr, Kavukcuoglu, Koray, Silver, David, Graves, Alex, Antonoglou, Ioannis, Wierstra, Daan, and Riedmiller, Martin. Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602, 2013.
34. Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. Neural Computation, 9(8), 1735–1780.
35. Wikipedia MCTS: https://en.wikipedia.org/wiki/Monte_Carlo_tree_search

36. Silver, David, Huang, Aja, Maddison, Chris J., Guez, Arthur, Sifre, Laurent, van den Driessche, George, Schrittwieser, Julian, Antonoglou, Ioannis, Panneershelvam, Veda, Lanctot, Marc, Dieleman, Sander, Grewe, Dominik, Nham, John, Kalchbrenner, Nal, Sutskever, Ilya, Lillicrap, Timothy, Leach, Madeleine, Kavukcuoglu, Koray, Graepel, Thore, and Hassabis, Demis. Mastering the game of go with deep neural networks and tree search. Nature, 529(7587):484–489, Jan 2016. Article.
37. Multi-armed bandits UCB1: https://en.wikipedia.org/wiki/Multi-armed_bandit
38. "SaaS. How it Works and Which Problems it has", blog by IDM, link: https://medium.com/@IDMdatasecurity/saas-how-it-works-and-which-problems-it-has-a05495cbfca
39. https://researcher.watson.ibm.com/researcher/files/zurich-jku/mdse-01.pdf