Neural-Guided Learning of Integer Sequences

Jaden Long, Tony Wu, Dennis Xu

Duke University

November 30, 2022

Outline

This slide is just for reference. We will hide it on final presentation.

1. Introduction: Why does this problem matter? An example problem. (Jaden)
2. Previous works: See the report latex file. (Jaden)
3. Methods: show the results from each method following that method
   3.1 Experimental setup: Just explain all the code we have (Dennis)
   3.2 Human (To-Do, we'll include this if we actually do it)
   3.3 Brute force search (Dennis)
   3.4 Symbolic regression with genetic algorithm (Jaden)
   3.5 seq2seq (Jaden)
   3.6 GPT3 (Dennis)
   3.7 Our proposed method: MCTS (Tony)
4. Discussion and future directions (Tony)

0, 4, 1, 10, 2, 16, 3, 22, 4, __, __, __

What are the next few numbers?

0, 4, 1, 10, 2, 16, 3, 22, 4, __, __, __

The Collatz Map[1]

$$a_n = \begin{cases} n/2 & \text{if } n \text{ is even} \\ 3 * n + 1 & \text{if } n \text{ is odd} \end{cases}$$

Collatz Conjecture (unsolved)
Every integer eventually map to 1 through repeated application of this map.

[1] Animation from Veritasium on YouTube

Introduction

Problem setup

- $\{a_n\}_{n=1}^N$ is generated by a function $f(n) = a_n$, with $f : n, a_{n-s}, a_{n-s+1}, ..., a_{n-1} \to a_n$ and initial condition $a_1, ..., a_s \in \mathbb{Z}$
- **Goal**: given an observed sequence $\{a_n\}_{n=1}^N$, find f^* such that $f^*(n) = f(n)$ for all $1 \leq n \leq N$
- *In this project*, we restrict f to be a polynomial with integer coefficients. We also add restrictions on s and complexity of f to prevent overfitting

Introduction

Why do we want to solve integer sequences?

- Many integer sequences can have interesting structures behind them (e.g. Fibonacci sequence).
- Plus, quantitative trading firms like to ask about them on interviews...

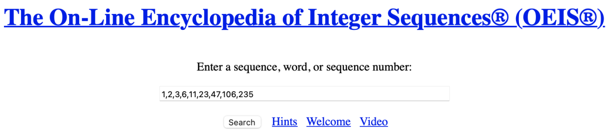
Difficulties

- The functional relation that generates the sequence is often hard to infer directly.
- Integer constraint is not guaranteed by standard seq2seq models.

Introduction

Neil Sloane and OEIS

Neil Sloane (1939 -)



How About Brute Force Search?

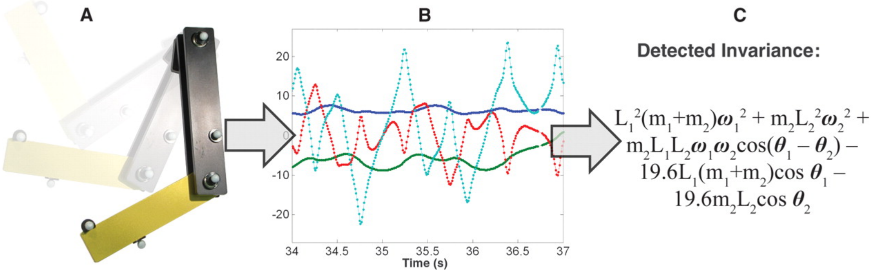
Given lists of allowed terms $\boldsymbol{t}$ ($|\boldsymbol{t}| = n$) and coefficients $\boldsymbol{c}$ ($|\boldsymbol{c}| = m$), some $1 < k \leq n$, and a target sequence $\boldsymbol{s}$ ($|\boldsymbol{s}| = h$),

1. Generate all possible k-combinations of the terms, $O({}_nC_k)$
2. For each combination, generate all possible functions by applying different coefficient permutations, $O(m^k)$
3. For each function, compute the respective sequence and compare with target, $O(h)$

In the worst case, the runtime is $O\left({}_nC_k \cdot m^k \cdot h\right) \approx O\left(\left(\frac{m \cdot n}{k}\right)^k \cdot h\right)$, which grows exponentially fast as we increase n and quickly becomes infeasible.

Previous Work

- **Schmidt and Lipson (2009)**: Symbolic regression with genetic algorithm to distil equations from experimental data



  - Led to the proprietary software Eureqa and startup Nutonian.

Symbolic Regression with Evolutionary Algorithm

- Algorithm used in Schmidt and Lipson (2009), reimplemented in package PySR (Cranmer, 2020)
- **Genetic algorithm**: Define a "fitness" for each function. "Mate" individuals with high fitness to produce better individuals. **Effectively searches non-convex space.**
- However, PySR does not support self-referencing terms[2] like $f(n) = 2f(n-1)$.

[2] Theoretically, we can implement it, but do we want to?

Experiment Setup: Dataset Generation

- **Terms**: constant, index-based (n, n^2, n^3), self-referencing ($f[n-i]^j$, $1 \leq i \leq 3$, $1 \leq j \leq 2$)
- **Functions**: $f = \sum_{i=1}^{k} c_i \cdot \text{term}_i$, where $\{\text{term}_i\}_{i=1}^{k}$ is a random k-combination of the terms (represented as a boolean mask) and c_i is chosen randomly in $(-5, 5)$
- **Sequences**: generated based on f, with initial terms chosen from $(1, 3)$ as needed
- Given an absolute value bound b on the sequence and k, the dataset consists of tuples $(sequence, boolean\_mask)$ where $\max(|sequence|) < b$ and exactly k values in *boolean_mask* are True
- With $b = 1000$, we generated 1000 tuples for each of $k = 2, 3$ and randomly split them into size 800 and 200 sets for training and test

Experiment Setup: Evaluation

Note that we cannot simply check for equality of the boolean masks

- Consider the sequence 1,3,6,10,15,21 where both $f(n) = n + f(n-1)$ and $f(n) = n^2 - f(n-1)$ are valid

Hence, we primarily evaluate the models based on two metrics:

1. **Average RMSE** (based on element-wise loss)
   - Given the predicted boolean mask, grid search over all valid term coefficients and generate the corresponding sequences
   - Compute the minimum RMSE achieved between the target sequence and all generated sequences, and average over all test sequences
   - Recall for sequences $\{a_i\}_{i=1}^n$, $\{b_i\}_{i=1}^n$, RMSE $:= \sqrt{\frac{1}{n}\sum_{i=1}^n (a_i - b_i)^2}$
2. **% Correct**
   - Percentage of predictions for which the best sequence generated from grid search matches the target perfectly (RMSE $= 0$)

Simple seq2seq

- Takes sequence of integers, outputs boolean mask of terms
- A simple encoder-decoder network with attention[3]
- Training: 20 epochs, 1000 functions, evaluated every 500 steps. 200 functions not in the training set evaluated randomly at each time
- **Results:**
  - Taking the **best** result from each trial across evaluations

% terms	% correct
3	17%

  - Just taking the **last** results:

Trial number	% correct
3	9%

[3] Taken from a tutorial on PyTorch

GPT3

Training

- Further split the 800 training sequences into 640 for training and 160 for validation
- Fine-tuned four base models, listed in increasing number of parameters: `ada`, `babbage`, `curie`, `davinci`
- Default hyperparameters: n_epochs=4, batch_size=1, learning_rate_multiplier=0.05

Results

- % correctly solved test sequences

# terms in f	ada	babbage	curie	davinci
2	18.0%	22.5%	25.5%	30.5%
3	9.0%	11.5%	11.0%	9.0%

- Mean RMSE on test sequences

# terms in f	ada	babbage	curie	davinci
2	40.1	25.0	28.1	23.5
3	334.7	215.0	119.6	284.0

Neural Network with MCTS

MCTS Recap

- Selection: starting at current node, pick the most promising leaf node L (a node that has unexplored child) based on a "tree policy"
- Expansion: if L doesn't end the game, append a child node C of L to the tree
- Simulation (rollout): play the game randomly starting from C, until game ends
- Backpropagation: Update score of nodes from u to C based on simulation outcome
- Repeat these steps for many times, and choose the best move. Go to the best move node and repeat.

Neural Network with MCTS

Methodology (inspired by AlphaGo)

- Search tree
  - Each node is a list of terms in a function expression (coefficients excluded). Root is $f(n) = 0$
  - Child nodes can be obtained by appending a new term.
  - Each node has a reward and a policy.
  - Evaluate reward at terminal nodes: when the number of terms in node reaches a depth limit, or when MCTS chooses to append an <EOS> token.
  - Reward: $10 - depth$ if $\exists$ a solution using exactly the node's terms, otherwise $-RMSE$
- Neural network: (sequence, current node's terms) $\rightarrow$ (reward, policy).
- Idea: Neural network guides MCTS, generating reward and policy estimates; then, neural network is trained to learn the updated reward and policy (critic); repeat.

Neural Network with MCTS

Training

- Training schedule: for each training sequence, run 3 MCTS iterations. Reinitialize an empty search tree at the start of every iteration.
- At the end of every iteration, neural network is trained on training examples collected from the node reward and policy in the search tree.
- Loss function at node u is
  $L(u) = [reward_{NN}(u) - reward_{MCTS}(u)]^2 + \sum_{a \in A} -policy_{MCTS}(u)_a \log \left[policy_{NN}(u)_a\right]$
- Neural network architecture: GRU or MLP backbone, with two projection heads (for reward and policy).

Neural Network with MCTS

Results

- % correctly solved test sequences

# terms in f	GRU	MLP
2	5.5%	13.0%
3	0.0%	0.5%
3 (with hint)	4.5%	3.5%

- Mean RMSE on test sequence

# terms in f	GRU	MLP
2	25.0	33.4
3	107.8	48.6
3 (with hint)	96.0	72.6

Comments

Concerns/Difficulties

- For PySR, maybe just experiments on functions without self-referencing terms?
- MCTS-driven training is slower than other methods

Plans Going Forward

- Finish symbolic regression experiments
- Use more sophisticated neural network architecture for MCTS and tune hyperparameters to improve performance
- Test the generalization ability of our models (test sequence expression has more terms than training sequence)
- Add human baseline
- Use more terms (e.g. interaction term)

Thank you!