Scallop and Neuro-Symbolic Programming

Lecture 1: Overview of Neuro-symbolic Method and Scallop Basics

About Us

Ziyang Li

Jiani Huang

Prof. Mayur Naik

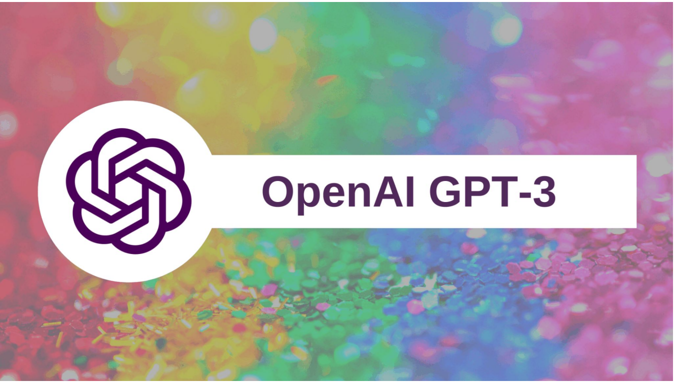
Agenda

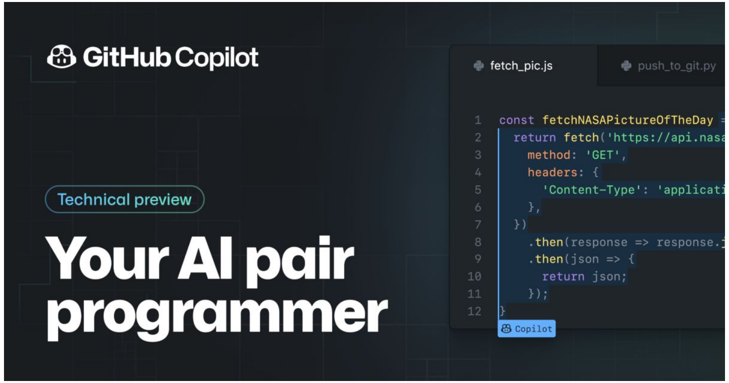
1. **Overview of Neuro-Symbolic Programming**
   Machine Learning vs. Formal Logic

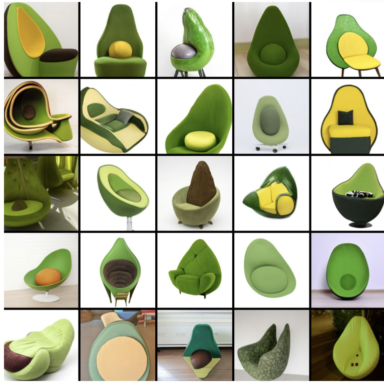
2. **Scallop Basics**
   Datalog and Relational Programming

Overview

Recent Advances in AI/ML...







However, there seems to be many problems…

Murray Shanahan
@mpshanahan

And this was the conversation we had about it:
Flamingo: This is a traffic light. It is red.
User: Can you see three circles?
Flamingo: Yes, I can see three circles.
User: What colours are they?
Flamingo: They are red, blue and green.
5/9

May 10th 2022

6 Likes

Murray Shanahan
@mpshanahan

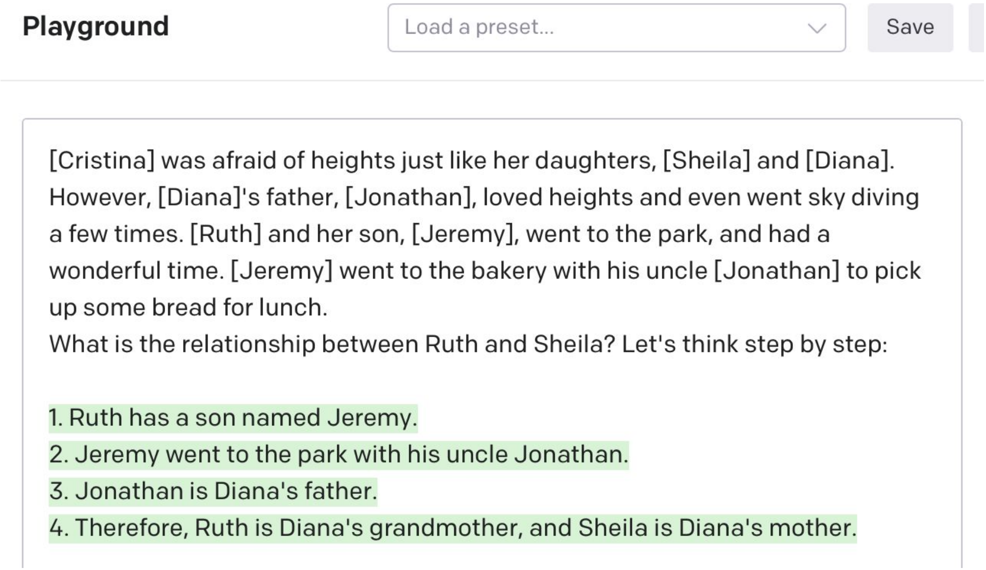
User: Where is the green circle?
Flamingo: The green circle is on the right.
User: Really? On the right of what?
Flamingo: The green circle is on the right of the blue circle.
6/9

May 10th 2022

4 Likes

https://garymarcus.substack.com/p/the-new-science-of-alt-intelligence

However, there seems to be many problems...



GPT-3 reasoning about kinships...

However, there seems to be many problems...

Playground

Load a preset...

Save

[Cristina] was afraid of heights just like her daughters, [Sheila] and [Diana]. However, [Diana]'s father, [Jonathan], loved heights and even went sky diving a few times. [Ruth] and her son, [Jeremy], went to the park, and had a wonderful time. [Jeremy] went to the bakery with his uncle [Jonathan] to pick up some bread for lunch.
What is the relationship between Ruth and Sheila? Let's think step by step:

1. Ruth has a son named Jeremy.
2. Jeremy went to the park with his uncle Jonathan.
3. Jonathan is Diana's father.
4. Therefore, Ruth is Diana's grandmother, and Sheila is Diana's mother.

We are asking Ruth and Sheila...

Spoiler Alert: Sheila is Ruth's **niece**

oning about kinships...

What Could be a Missing Piece?

- Deep Neural Networks are very good at perceiving the world. But they are very poor at consistently and logically reason about it
- We can reason with logical and relational symbols
- A logical and relational reasoning system could bridge the gap between logical symbols and deep neural networks
- If such a system is probabilistic and differentiable, it could be tightly integrated with machine learning models to perform the reasoning that the neural networks nowadays cannot do

What Could be a Missing Piece?

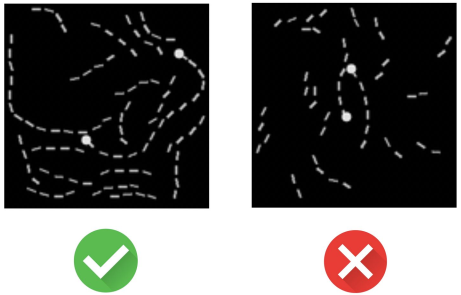
- Deep Neural Networks are very good at perceiving the world. But they are very poor at consistently and logically reason about it
- We can reason with logical and relational symbols
- A logical and relational reasoning system could bridge the gap between logical symbols and deep neural networks
- If such a system is probabilistic and differentiable, it could be tightly integrated with machine learning models to perform the reasoning that the neural networks nowadays cannot do

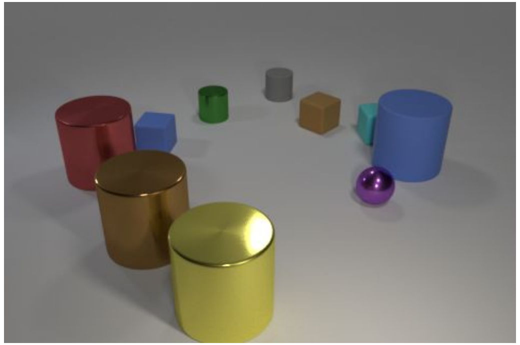
Neuro-Symbolic Methods

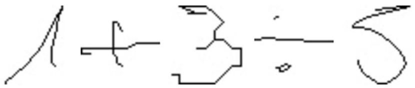
What is Scallop

- Scallop is a language and framework for Neuro-symbolic programming, that aims to bridge the gap between perception and reasoning
- It is based on the language of Datalog, and additionally supports (stratified) negation and aggregation
- Users can instrument customizable provenance, allowing for discrete, probabilistic, and even differentiable reasoning
- Scallop can be easily integrated with modern days machine learning frameworks, such as PyTorch

What can Scallop do?







Learning Objectives

1. Learn the language of Scallop
2. Learn the concept of provenance, tagging and instrumentation
3. Probabilistic Reasoning and Differentiable Reasoning
4. Running simple Machine Learning experiments, and Program with Scallop on Machine Learning tasks that involve both perception and logical reasoning

Scallop Basics: Datalog and Relations

What is a Relation?

Alice is the mother of Bob

What is a Relation?

Alice is the mother of Bob

Entities

What is a Relation?

Alice is the mother of Bob

Relation **Entities**

What is a Relation?

Alice is the mother of Bob

```
mother("Alice", "Bob")
```

What is a Relation?

Alice is the mother of Bob

```
mother("Alice", "Bob")
```

Relation / Predicate

What is a Relation?

Alice is the mother of Bob

```
mother("Alice", "Bob")
```

2-Tuple

What is a Relation?

Alice is the mother of Bob

```
mother("Alice", "Bob")
```

Arity-2 Fact

Many concepts can be encoded as relations

$3 + 4 = 7$

↓

```
add(3, 4, 7)
```

Alice is the mother of Bob

↓

```
mother("Alice", "Bob")
```

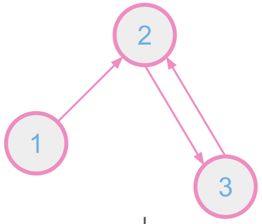
Many concepts can be encoded as relations

$3 + 4 = 7$

```
add(3, 4, 7)
```

Alice is the mother of Bob

```
mother("Alice", "Bob")
```




```
node(1)   edge(1, 2)
node(2)   edge(2, 3)
node(3)   edge(3, 2)
```

Specifying a relation in Scallop

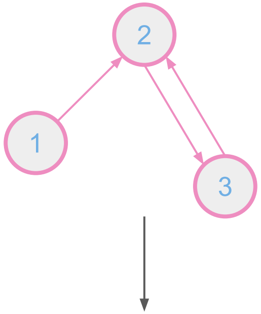
$$3 + 4 = 7$$

↓

```
add(3, 4, 7)
```

```
// Written in Scallop
rel add(3, 4, 7)
```

Specifying a relation in Scallop




```
node(1)   edge(1, 2)
node(2)   edge(2, 3)
node(3)   edge(3, 2)
```

```
// Written in Scallop
rel node = {(1), (2), (3)}
rel edge = {(1, 2), (2, 3), (3, 2)}
```

We can derive new facts by combining facts

```
father("John", "Alice")        mother("Alice", "Bob")
```

We can derive new facts by combining facts

```
father("John", "Alice")        mother("Alice", "Bob")
------------------------------------------------------
             grandfather("John", "Bob")
```

We can derive new facts by combining facts

```
father("John", "Alice")        mother("Alice", "Bob")
---------------------------------------------------------
              grandfather("John", "Bob")
```

Proof Tree

We can generalize the derivation into Rules

father(“John”, “Alice”)   mother(“Alice”, “Bob”)

---

grandfather(“John”, “Bob”)

```
// Written in Scallop
rel father = {("John", "Alice")}
rel mother = {("Alice", "Bob")}
rel grandfather(a, b) = father(a, c) and mother(c, b)
```

We can generalize the derivation into Rules

```
// Written in Scallop
rel father = {("John", "Alice")}
rel mother = {("Alice", "Bob")}
rel grandfather(a, b) = father(a, c) and mother(c, b)
```

We can generalize the derivation into Rules

```
// Written in Scallop
rel father = {("John", "Alice")}
rel mother = {("Alice", "Bob")}
rel grandfather(a, b) = father(a, c) and mother(c, b)
```

Variables

We can generalize the derivation into Rules

```
// Written in Scallop
rel father = {("John", "Alice")}
rel mother = {("Alice", "Bob")}
rel grandfather(a, b) = father(a, c) and mother(c, b)
```

Head

We can generalize the derivation into Rules

```
// Written in Scallop
rel father = {("John", "Alice")}
rel mother = {("Alice", "Bob")}
rel grandfather(a, b) = father(a, c) and mother(c, b)
```

Body

Rules can be Recursive

```
// Written in Scallop
rel ancestor(a, b) = father(a, b) or mother(a, b)
rel ancestor(a, b) = ancestor(a, c) and ancestor(c, b)
```

Rules can be Recursive

```
// Written in Scallop
rel ancestor(a, b) = father(a, b) or mother(a, b)
rel ancestor(a, b) = ancestor(a, c) and ancestor(c, b)
```

Ancestor appears in both body and the head

Rules can be Recursive

```
// Written in Scallop
rel ancestor(a, b) = father(a, b) or mother(a, b) // Rule 1
rel ancestor(a, b) = ancestor(a, c) and ancestor(c, b) // Rule 2
```

father("John", "Alice") ——— [Rule 1] ——— ancestor("John", "Alice")

mother("Alice", "Bob") ——— [Rule 1] ——— ancestor("Alice", "Bob")

ancestor("John", "Alice"), ancestor("Alice", "Bob") ——— [Rule 2] ——— ancestor("John", "Bob")

Rules can be Recursive

```
// Written in Scallop
rel ancestor(a, b) = father(a, b) or mother(a, b) // Rule 1
rel ancestor(a, b) = ancestor(a, c) and ancestor(c, b) // Rule 2
```

Base Facts

```
father("John", "Alice")                    mother("Alice", "Bob")
---------------------------- [Rule 1]      ---------------------------- [Rule 1]
ancestor("John", "Alice")                  ancestor("Alice", "Bob")
------------------------------------------------------------------------- [Rule 2]
                        ancestor("John", "Bob")
```

Rules can be Recursive

```
// Written in Scallop
rel ancestor(a, b) = father(a, b) or mother(a, b) // Rule 1
rel ancestor(a, b) = ancestor(a, c) and ancestor(c, b) // Rule 2
```

Facts derived in the 1st iteration

John”, “Alice”) mother(“Alice”, “Bob”)

[Rule 1] [Rule 1]

ancestor(“John”, “Alice”) ancestor(“Alice”, “Bob”)

[Rule 2]

ancestor(“John”, “Bob”)

Rules can be Recursive

```
// Written in Scallop
rel ancestor(a, b) = father(a, b) or mother(a, b) // Rule 1
rel ancestor(a, b) = ancestor(a, c) and ancestor(c, b) // Rule 2
```

father("John", "Alice")

[Rule 1]

Alice")

mother("Alice", "Bob")

[Rule 1]

ancestor("Alice", "Bob")

[Rule 2]

ancestor("John", "Bob")

Facts derived in the 2nd iteration

How do we know when to stop?

$$\dfrac{\dfrac{\texttt{father("John", "Alice")}}{\texttt{ancestor("John", "Alice")}}\,[\text{Rule 1}] \qquad \dfrac{\texttt{mother("Alice", "Bob")}}{\texttt{ancestor("Alice", "Bob")}}\,[\text{Rule 1}]}{\texttt{ancestor("John", "Bob")}}\,[\text{Rule 2}]$$

We will stop when there is **NO new facts** that can be derived from the rules

How do we know when to stop?

```
father("John", "Alice")                    mother("Alice", "Bob")
------------------------------- [Rule 1]   ------------------------------- [Rule 1]
ancestor("John", "Alice")                  ancestor("Alice", "Bob")
------------------------------------------------------------------------- [Rule 2]
                     ancestor("John", "Bob")
```

We will stop when there is **NO new facts** that can be derived from the rules

Execute until **fixpoint**

Rules can be Recursive... and Infinite

```
// Written in Scallop
rel natural_number(0)
rel natural_number(n + 1) = natural_number(n)
```

Rules can be Recursive… and Infinite

```
// Written in Scallop
rel natural_number(0)
rel natural_number(n + 1) = natural_number(n)
```

We can write simple expressions to create new value

Rules can be Recursive… and Infinite

```
// Written in Scallop
rel natural_number(0)
rel natural_number(n + 1) = natural_number(n)
```

natural_number(0)
natural_number(1)
natural_number(2)
natural_number(3)
natural_number(4)

…

Rules can be Recursive... and Infinite

```
// Written in Scallop
rel natural_number(0)
rel natural_number(n + 1) = natural_number(n)
```

```
natural_number(0)
natural_number(1)
natural_number(2)
natural_numbe
natural_numbe
...
```

New value is created in every iteration
i.e. There is **NO FIXPOINT**

Rules can be Recursive… and Infinite

```
// Written in Scallop
rel natural_number(0)
rel natural_number(n + 1) = natural_number(n), n < 100
```

A not so elegant fix: setting a bound on n

Rules can contain Negations

```
// Written in Scallop
rel person = {"John", "Alice", "Bob"}
rel has_no_children(a) = person(a) and ~father(a, _) and ~mother(a, _)
```

Rules can contain Negations

```
// Written in Scallop
rel person = {"John", "Alice", "Bob"}
rel has_no_children(a) = person(a) and ~father(a, _) and ~mother(a, _)
```

"a" is neither a father nor a mother

Rules can contain Negations

```
// Written in Scallop
rel person = {"John", "Alice", "Bob"}
rel has_no_children(a) = person(a) and ~father(a, _) and ~mother(a, _)
```

We need to use a whole set (person) to limit the search for such a person

Rules can contain Negations

```
// Written in Scallop
rel person = {"John", "Alice", "Bob"}
rel father = {("John", "Alice")}
rel mother = {("Alice", "Bob")}
rel has_no_children(a) = person(a) and ~father(a, _) and ~mother(a, _)
```

↓

has_no_children(“Bob”)

Rules can contain Negations

```
// Written in Scallop
rel person = {"John", "Alice", "Bob"}
rel father = {("John", "Alice")}
rel mother = {("Alice", "Bob")}
rel has_no_children(a) = person(a) and ~father(a, _) and ~mother(a, _)
```

↓

`has_no_children("Bob")` 🥲

But not arbitrary Negation...

```
// This rule will be rejected by the compiler
rel this_is_true() = ~this_is_true()
```

A predicate cannot be used negatively to prove itself

We can use Aggregations

```
rel person = {"John", "Alice", "Bob"}
rel num_people(n) = n = count(p: person(p)) // 3
```

We can use Aggregations

```
rel person = {"John", "Alice", "Bob"}
rel num_people(n) = n = count(p: person(p)) // 3
```

Count is an aggregator

There are many kinds of Aggregators

```
rel person_age = {("John", 65), ("Alice", 40), ("Bob", 15)}
// Max and Argmax
rel maximum_age(x) = x = max(a: person_age(_, a))
rel oldest_person(p) = _ = max[p](a: person_age(p, a))
// Exists
rel exists_person_over_50(v) = v = exists(p: person_age(p, a) and a > 50)
```

Conclusion

- We have talked about
  - Overview of neuro-symbolic methods
  - Language syntax and basics of Scallop
  - Concepts of proof tree and iterative execution until fixpoint
- Exercises during the Lab
  - Writing interesting Scallop programs about graph algorithms and scene graphs
- What's next
  - Using Scallop for probabilistic reasoning and differentiable reasoning, with tags, instrumentation, and provenance