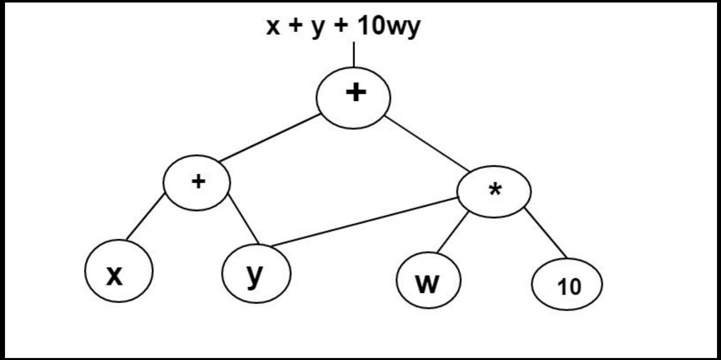
Reinforcement Learning of Polynomials

Hansel Lee, Kyle Zhang, Junbo Huang

Mentors: Jarod Alper, William Dudarov

Introduction and Goal

- Arithmetic circuits **compute a polynomial** using binary operations + and * where + is addition and * is multiplication.
- Use **reinforcement learning** to generate efficient arithmetic circuits representing polynomials with minimal complexity.
- Can the successes of **AlphaZero** be replicated for this task?

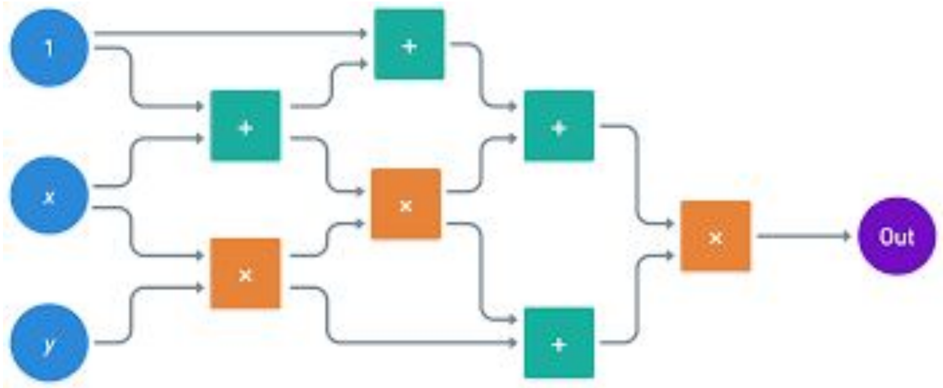


Arithmetic Circuits Example

For a polynomial: $x^2 + 2xy + y^2$

Efficiently computed:
(x) add gate (y) ⟶ A
(A) multiply gate (A)

Inefficiently computed:
(x) multiply gate (x) ⟶ A
(y) multiply gate (y) ⟶ B
(x) multiply gate (y) ⟶ C
(x) multiply gate (y) ⟶ D
(A) add gate (C) add gate (D) add gate (B)



Approaches

Frozen Lake Environment (Breadth First Search)

There is a reinforcement learning benchmark in **OpenAI Gymnasium** called FronzenLake Environment for navigating from start to goal.

We tried to apply search algorithms from **FrozenLake** to polynomial simplification, finding the shortest transformation path. Generating random polynomials using SymPy and apply search algorithms for simplification.

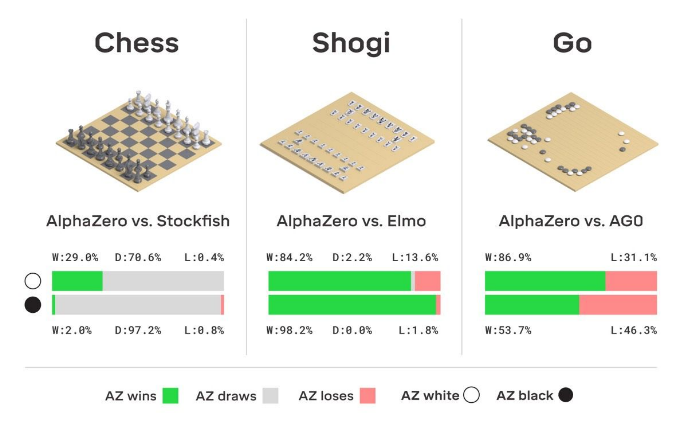
```python
import sympy as sp
import random

def generate_random_polynomial(variables=['x', 'y', 'z'], degree=5, terms=10):
    vars = [sp.Symbol(v) for v in variables]
    polynomial = sum(random.randint(1, 5) * sp.Mul(*random.choices(vars, k=random.randint(1, degree)))
                     for _ in range(terms))
    return sp.expand(polynomial)

random_poly = generate_random_polynomial()
print('randomly generated polynomial:')
print(random_poly)

def simplify_polynomial(poly):
    return sp.factor(poly)

simplified_poly = simplify_polynomial(random_poly)
print('simplify:')
print(simplified_poly)

from collections import deque

def find_shortest_simplification_path(start_poly):

    queue = deque([(start_poly, [])])
    visited = set()

    while queue:
        poly, path = queue.popleft()

        if poly == simplify_polynomial(start_poly):
            return path + [poly]

        next_states = [sp.factor(poly), sp.expand(poly)]

        for next_poly in next_states:
            if next_poly not in visited:
                visited.add(next_poly)
                queue.append((next_poly, path + [next_poly]))

    return None

path = find_shortest_simplification_path(random_poly)
print("simplified path")
for step in path:
    print("→", step)
```

Limitations of Frozen Lake

- This approach works well for polynomials that can be directly factored into **simple components**.
- For polynomials that cannot be easily factored, **the method struggles** to find an efficient simplification path.
- For example, it can directly factorize x^2 + 4*x + 3 into (x + 1)(x + 3), but cannot deal with more complex case like x^2 + 4*x + 4.

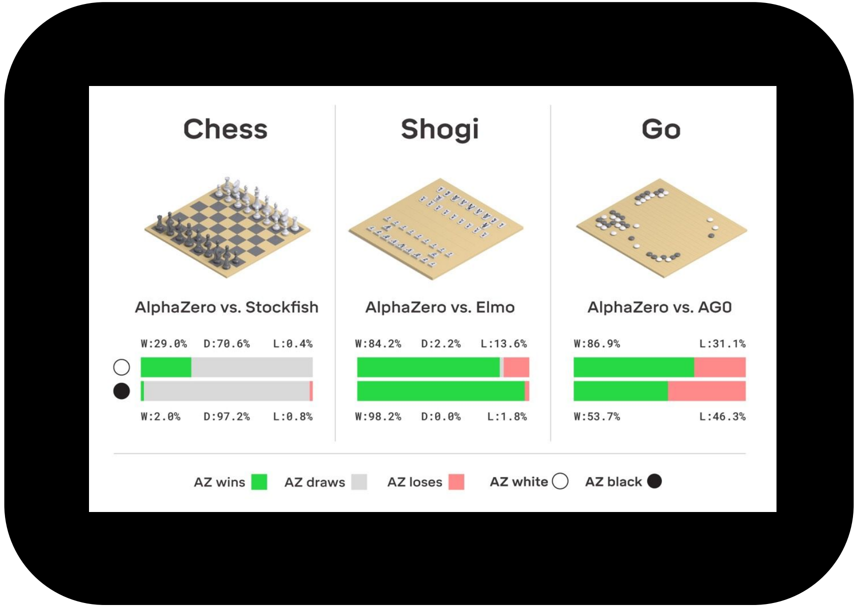
Using AlphaZero

- Using **Monte Carlo Tree Search** (MCTS) to explore action space of creating arithmetic circuits.
- Train a neural network to learn a **policy from MCTS**, enabling efficient polynomial computation.
- Direct MCTS computation is slow, so we develop a model for efficient predictions.



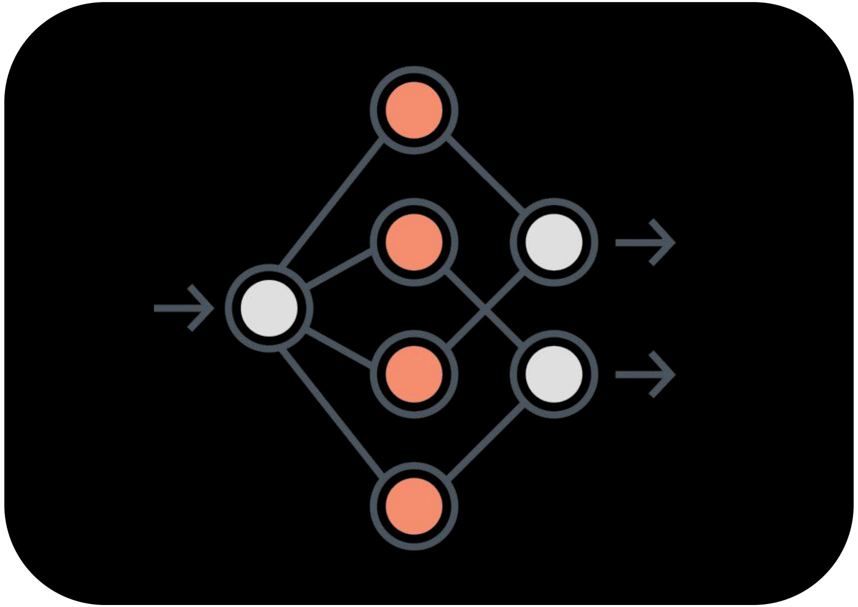
Obstacles in AlphaZero

- Converting AlphaZero algorithm to a **single player game**
  - Identifying win/lose conditions
- Representing the circuit as a **fixed-size tensor**
  - Our action space is continuous due to the ability to add constants



Next Steps

- Try other reinforcement learning algorithms, such as **Proximal Policy Optimization** (PPO)
- Generate a **dataset** with efficiently computable polynomials
- Experiment with different state and action **representations**.
  - Learned embeddings?
  - Textual representation?



Questions?

Hansel Lee, Kyle Zhang, Junbo Huang (UW XLL WI 2025)