236606 - FSTMA Project - Settlers of Catan

Part 3 / 3

Aviv Cohen and Dan Navon

The Taub Faculty of Computer Science
Technion - Israel Institute of Technology

A description of the challenge we chose to tackle

The Settlers of Catan Domain: Description

Catan is a resource driven game in which players roll dice to collect resources, trade materials with other players, and compete to be the first one to achieve 10 points.
Our goal is to create an agent that improves his strategy to this game, while playing against himself.

Why is it interesting and why is it hard

Settlers of Catan consider as one of the most popular board games[1].
Catan makes a great Reinforcement Learning research environment because of its high strategic nature and a good mixture of competition and cooperation.
Why is it hard?

- Many optional actions combining with stochastic moves (Robber, dice rolling, development cards, etc.) lead to a great complexity.
- In oppose to a 2 players competition games (as Go, Chess) Catan may involve 4 players that needs to coordinate and compete, so it's hard to derive/adopt from current algorithms.
- Current AI agents still underperform against humans.

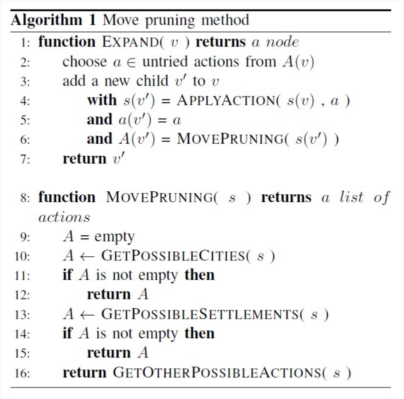
[1] `https://www.catan.com/about-us`

Previous works

- QSettlers[2], a DQN paradigm. working good for the player trading mechanism of the game but not a general gameplay.
- "Optimizing UCT for Catan"[3] uses MCTS with a combination of pruning strategy that uses domain knowledge and trade-optimistic search heuristic in order to reduce the algorithm's search space. - computation demanding, no learning process
- "AlphaZero GO"[4] is a well known algorithm which relay on MCTS but replacing the rollouts with a CNN in order to learn the value function and speed up the decision. Assumes full observability and only 2 agents.

[2] https://akrishna77.github.io/QSettlers/
[3] https://www.sbgames.org/sbgames2017/papers/ComputacaoFull/175405.pdf
[4] https://www.nature.com/articles/nature16961.pdf

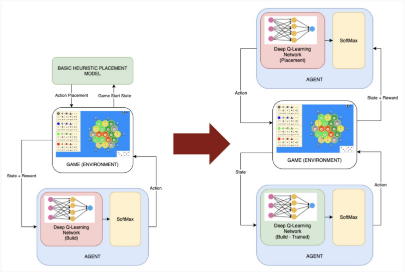
Previous works



Algorithm 1 Move pruning method

```
function EXPAND( $v$ ) returns $a$ $node$
    choose $a \in$ untried actions from $A(v)$
    add a new child $v'$ to $v$
        with $s(v')$ = APPLYACTION( $s(v)$ , $a$ )
        and $a(v') = a$
        and $A(v')$ = MOVEPRUNING( $s(v')$ )
    return $v'$

function MOVEPRUNING( $s$ ) returns $a$ $list$ $of$ $actions$
    $A$ = empty
    $A \leftarrow$ GETPOSSIBLECITIES( $s$ )
    if $A$ is not empty then
        return $A$
    $A \leftarrow$ GETPOSSIBLESETTLEMENTS( $s$ )
    if $A$ is not empty then
        return $A$
    return GETOTHERPOSSIBLEACTIONS( $s$ )
```

(a) MCTS with move-prunning



(b) DQN paradigm

DQN paradigm[5]

[5]https://cs230.stanford.edu/projects_fall_2021/reports/103176936.pdf

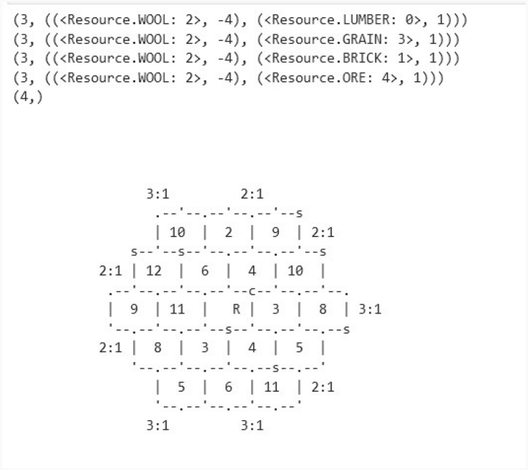
Formal model

Presentation: Formal Model

We model the environment as an MDP

- $\mathcal{X}$ - $\{current\_player, dice, longest\_road, intersection\_buildings, roads, resources, harbors\}$
- $\mathfrak{A} = \{\mathcal{A}^i\}_{i=1}^4$ where $\mathcal{A}^i(s) = \{$"possible trades","possible buildings(cities/settlements/roads)", "end turn"$\}$
  The trading and building possibilities depend on the resources availability.
- $\mathfrak{P}(s'|s,a)$- The probability of getting the state s', when we are in the state s, and make the action a. The stochastic transition happens only when $s.phase = "dice"$. The stochasticity is over $s'.resources$ for all players
- $\mathfrak{R} = \{\mathcal{R}^i\}_{i=1}^4$ - reward functions. $\mathcal{R}^i = \frac{agent^i\_score}{\sum scores}$
- Our objective is to find an optimal policy for our agent that maximizes his utility function.

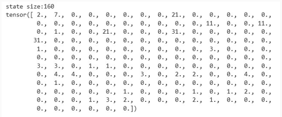
Model representation example



```
(3, ((<Resource.WOOL: 2>, -4), (<Resource.LUMBER: 0>, 1)))
(3, ((<Resource.WOOL: 2>, -4), (<Resource.GRAIN: 3>, 1)))
(3, ((<Resource.WOOL: 2>, -4), (<Resource.BRICK: 1>, 1)))
(3, ((<Resource.WOOL: 2>, -4), (<Resource.ORE: 4>, 1)))
(4,)
```

```
      3:1         2:1
       .--'--.--'--.--'--s
       | 10  |  2  |  9  | 2:1
    s--'--s--'--.--'--.--'--s
2:1 | 12  |  6  |  4  | 10  |
 .--'--.--'--.--'--c--'--.--'--.
 |  9  | 11  |   R |  3  |  8  | 3:1
 '--.--'--.--'--s--'--.--'--.--s
2:1 |  8  |  3  |  4  |  5  |
    '--.--'--.--'--.--s--.--'
       |  5  |  6  | 11  | 2:1
       '--.--'--.--'--.--'
      3:1         3:1
```

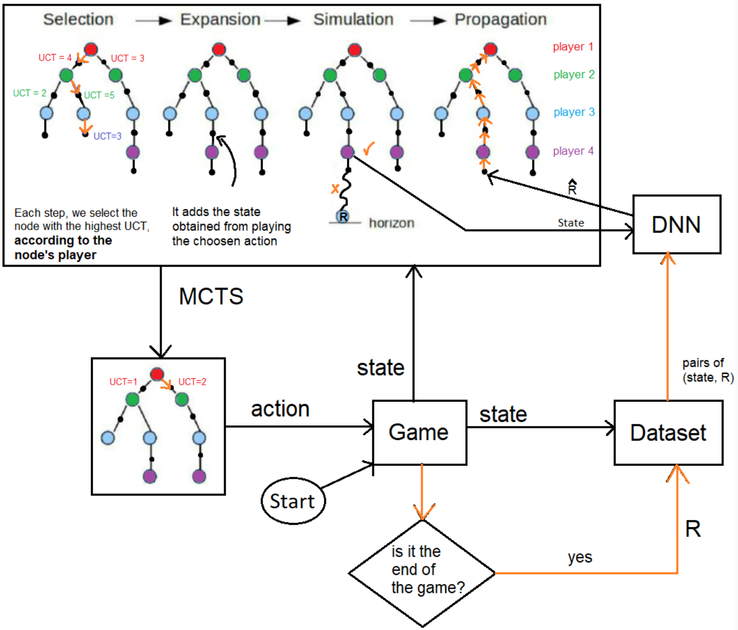
(a) actions: (4)- end turn and roll dice



```
state size:160
tensor([ 2.,  7.,  0.,  0.,  0.,  0.,  0.,  0., 21.,  0.,  0.,  0.,  0.,  0.,
         0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0., 11.,  0.,  0., 11.,
         0.,  1.,  0.,  0., 21.,  0.,  0.,  0., 31.,  0.,  0.,  0.,  0.,  0.,
        31.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
         1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  3.,  0.,  0.,  0.,
         0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
         3.,  3.,  0.,  1.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
         0.,  4.,  4.,  0.,  0.,  0.,  3.,  0.,  2.,  2.,  0.,  0.,  4.,  0.,
         0.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
         0.,  0.,  0.,  0.,  0.,  1.,  0.,  0.,  0.,  1.,  0.,  1.,  2.,  0.,
         0.,  0.,  0.,  1.,  3.,  2.,  0.,  0.,  0.,  2.,  1.,  0.,  0.,  0.,
         0.,  0.,  0.,  0.,  0.,  0.])
```

(b) Encoded state vector

Suggested solution



Method

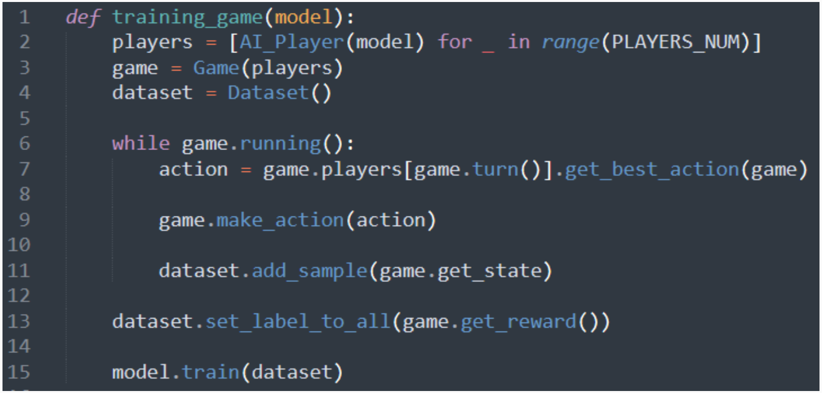
Intuitive Description

The advantages of using DNN on the regular MCTS:

- The DNN saves information between games, while regular MCTS saves information between trains in the same game only.
- In long played games (like Catan), the simulation phase of MCTS takes a lot of time, while forward pass in DNN is fast.
- In games with a lot of randomness, each MCTS iteration gives little information. So even after many iterations, the MCTS will have a lack of information, while DNN has an inclusion property.
- It's possible to use only the DNN in the production phase, and give up on the MCTS, to make the agent play very fast.

Full Description

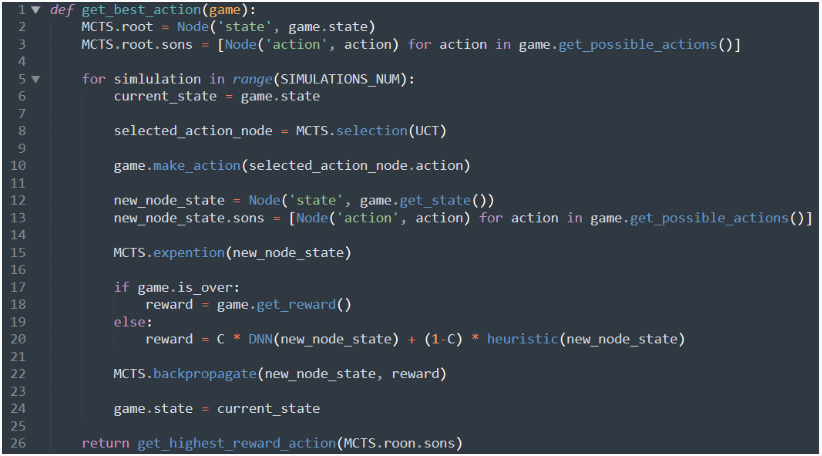
The function that describes how to implement one training game:



```python
def training_game(model):
    players = [AI_Player(model) for _ in range(PLAYERS_NUM)]
    game = Game(players)
    dataset = Dataset()

    while game.running():
        action = game.players[game.turn()].get_best_action(game)

        game.make_action(action)

        dataset.add_sample(game.get_state)

    dataset.set_label_to_all(game.get_reward())

    model.train(dataset)
```

Full Description

The function of the agent that describes how he chooses an action:



```python
def get_best_action(game):
    MCTS.root = Node('state', game.state)
    MCTS.root.sons = [Node('action', action) for action in game.get_possible_actions()]

    for simlulation in range(SIMULATIONS_NUM):
        current_state = game.state

        selected_action_node = MCTS.selection(UCT)

        game.make_action(selected_action_node.action)

        new_node_state = Node('state', game.get_state())
        new_node_state.sons = [Node('action', action) for action in game.get_possible_actions()]

        MCTS.expention(new_node_state)

        if game.is_over:
            reward = game.get_reward()
        else:
            reward = C * DNN(new_node_state) + (1-C) * heuristic(new_node_state)

        MCTS.backpropagate(new_node_state, reward)

        game.state = current_state

    return get_highest_reward_action(MCTS.roon.sons)
```

Evaluation

- After playing a number of games, we trained our neural network and then run multiple games(20) in order to collect statistics over agents' winning rate.
- The evaluation has been done by comparing different types of agents.
    - *Agent*1 - Trained NN with pruning
    - *Agent*2 - Trained NN without pruning
    - *Agent*3 - No NN with pruning
    - *Agent*4 - No NN without pruning
    - *Agent*5 - Random actions
- Agents 1-4 are based on the MCTS, with 300 iterations, using heuristic.
- Since we use different rules and a different framework comparing to previous works, it's impossible to compare our methods with others.

Results

- We expected that the winning rate will be higher while using a trained NN against non NN, and action pruning should be better than no pruning.
- We also expected that the NN should have a greater impact over winning rate comparing the pruning.
- Obviously the random agent should perform the worst which was anticipated

Results

- Agent 1 got the best results as expected.
- Agent 3 has second-best scores, while agents 2 and 4 have similarly lower scores.
- Agent 3 is that it uses the MCTS wisely. Iterations will go deeper in the tree in contrast to agents 2 and 4 which will search more widely
- It seems that the NN is not strong enough as well (i.e. not trained enough or bad architecture)

	$Agent1$	$Agent2$	$Agent3$	$Agent4$	$Agent5$
Winning rate	40%	6.6%	33%	13.3%	0%

Conclusion

- AlphaZero algorithm needs adaptation for games with more than 2 players, stochastic moves and partially observable states.
- Creating and training a deeper NN requires more computational power to achieve AlphaZero-like performance, otherwise it's impractical.
- Replacing heuristics with DQN on parts of the game (trading, placement, etc.)