Alpha Toe

Applying deep learning to solve tic-tac-toe
https://github.com/DanielSlater/AlphaToe

Presented by Daniel Slater

What is this about?

- We are going to look at how Alpha Go was built.
- Apply some of those techniques to playing tic-tac-toe
- We can actually build Alpha Go in TensorFlow in only 6-7 files

Go

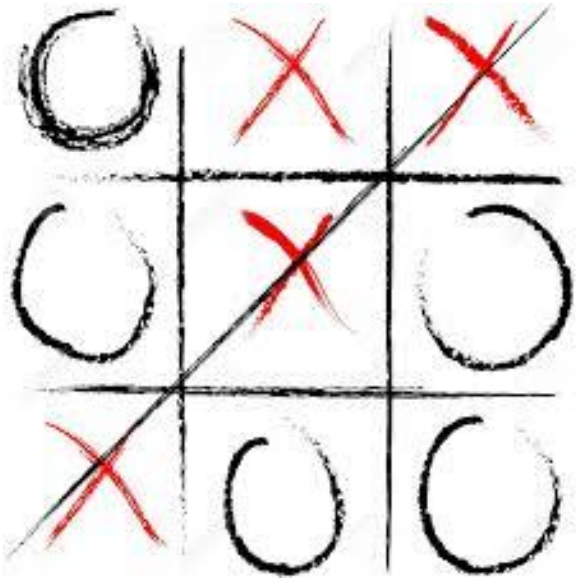
- Go, an ancient Chinese game whose origin goes back more than 5,500 years

- One of the most complex board games
  - Chess has 10^{123} possible games
  - Go has 10^{360}

- Alpha Go, by Google Deepmind recently became the first AI to beat a top level human Go player.

Toe - Tic-tac-toe

- Less than 360,000 possible games.
- Is solvable with quite simple techniques.
- But can also be solved using cutting edge deep neural networks, so why not.
- Also



Toe - Tic-tac-toe

- Game only takes about a page of code

```
def _new_board():
        return ((0, 0, 0), (0, 0, 0), (0, 0, 0))

def apply_move(board_state, move, side):
        move_x, move_y = move

        def get_tuples():
        for x in range(3):
        if move_x == x:
                temp = list(board_state[x])
                temp[move_y] = side
                yield tuple(temp)
        else:
                yield board_state[x]

        return tuple(get_tuples())

def available_moves(board_state):
        for x, y in itertools.product(range(3), range(3)):
        if board_state[x][y] == 0:
        yield (x, y)
```

```
def available_moves(board_state):
        for x, y in itertools.product(range(3), range(3)):
        if board_state[x][y] == 0:
        yield (x, y)

def _has_3_in_a_line(line):
        return all(x == -1 for x in line) | all(x == 1 for x in line)

def has_winner(board_state):
        for x in range(3):
        if _has_3_in_a_line(board_state[x]):
        return board_state[x][0]
        for y in range(3):
        if _has_3_in_a_line([i[y] for i in board_state]):
        return board_state[0][y]
        if _has_3_in_a_line([board_state[i][i] for i in range(3)]):
        return board_state[0][0]
        if _has_3_in_a_line([board_state[2 - i][i] for i in range(3)]):
        return board_state[0][2]

    return 0
```

Why do we care about any of this?

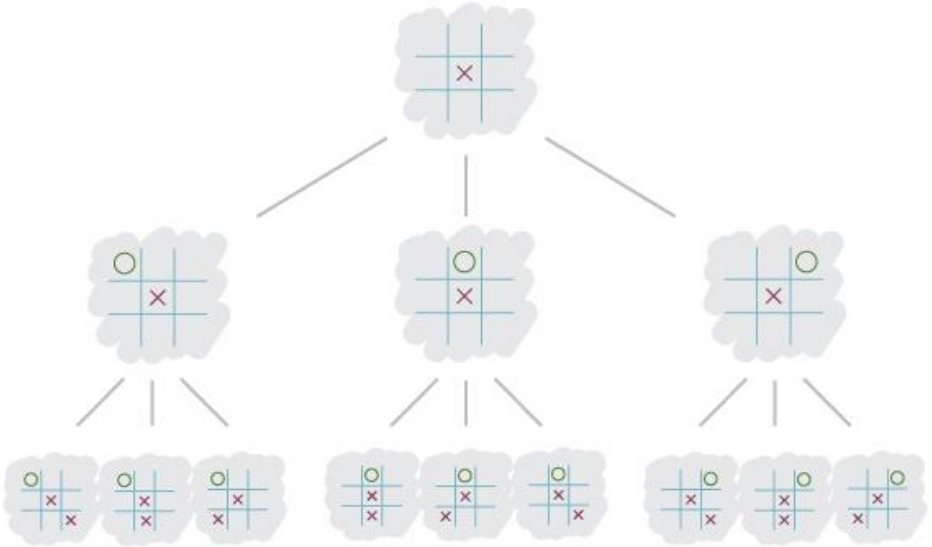
- We can use similar approaches to solve all kinds of real world problems
  - Autonomous robot agents
  - Trading systems
  - Business intelligence
  - Chat bots
- It’s fun

Why board games as opposed to other games

- Unlike Atari games we are not learning from the images.
- We know the rules
- We know the exact state
- We can simulate moves forwards

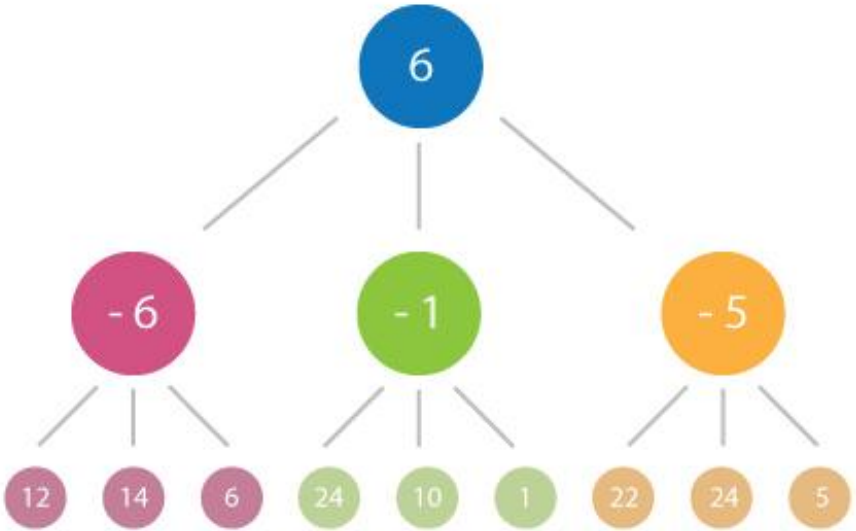
What is the best move in a board game?

- Series of states
- Construct a tree of states based on possible moves for each player



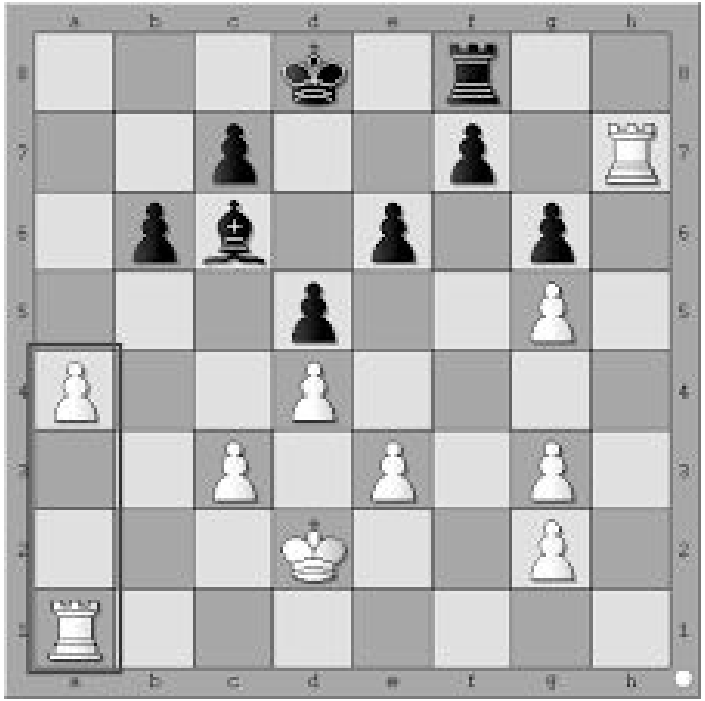
Min-Max algorithm

- Worst best moves for opponents
- Simulate through to terminal states



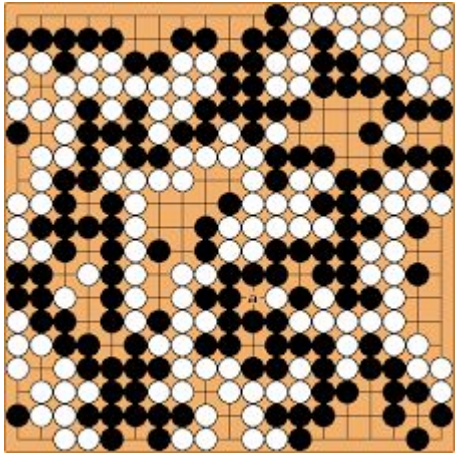
Min-Max algorithm

- Use an evaluation function to estimate the value of the position at max depth
- Here white is up a rook for a bishop
- Which equals +2 pawn advantage to white



Min-Max for Go

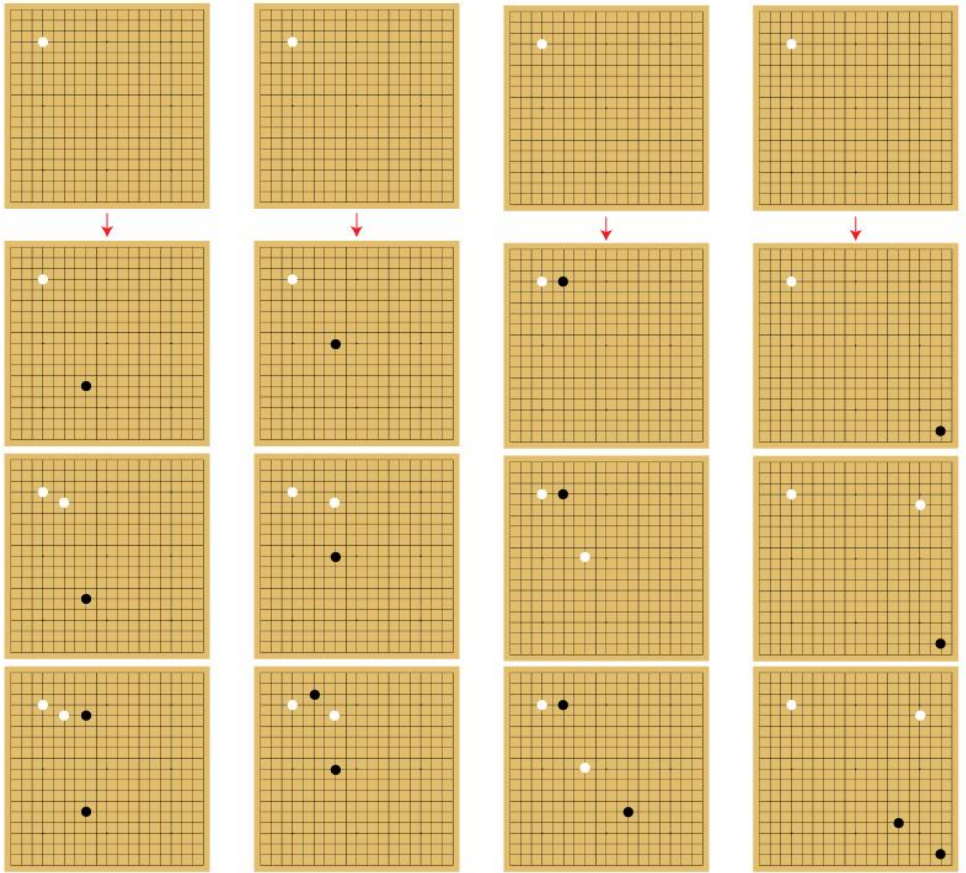
- State space is too vast
- No good evaluation functions
- Go is hard for humans to learn from



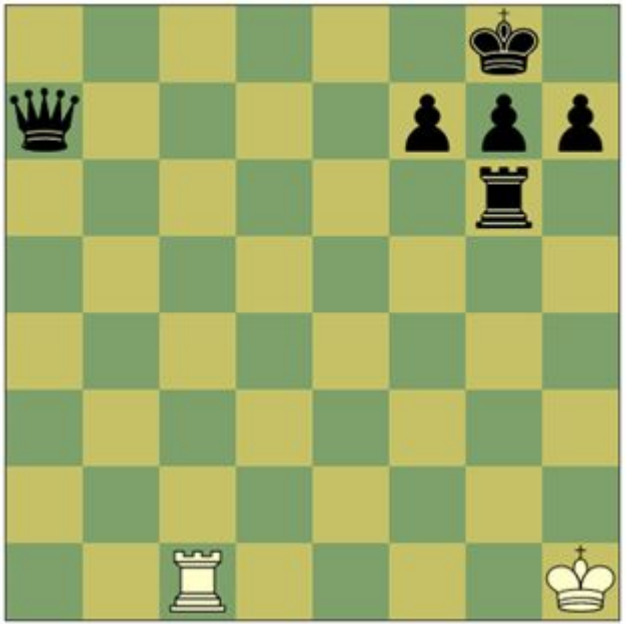
Monte-Carlo sampling

How else might we evaluate a position?

Take the average result from simulating it



Monte carlo sampling can be very inaccurate



Monte-carlo tree search with Upper Confidence bounds for Trees

- Allows monte-carlo to converge towards min-max
- Make a decision between
  - Exploitation
  - Exploration
- This was the Go-to technique for Go for many years

Upper confidence bound: $v_i + C \times \sqrt{\frac{\ln N}{n_i}}$

For more info, David Silver lecture: http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/XX.pdf

Code: https://github.com/DanielSlater/AlphaToe/blob/master/techniques/monte_carlo.py

How are we going to improve at Go?

- Deep neural networks are hip right now
- Used to map an input to an output
- Look at who has the current best performance and learn from them - Humans

Supervised training to learn human moves

- Neural network
  - Input is the 19x19 board, informing it which pieces are in each square
  - Output is a 19x19 board, each neuron representing the probability that a human player would make a move in that position.
- Trained on a database of human go games to predict the human move in each position.
- After training predicted the move chosen by a human with 57% accuracy.
- We now have a trained supervised network.

Applications to Alpha Toe

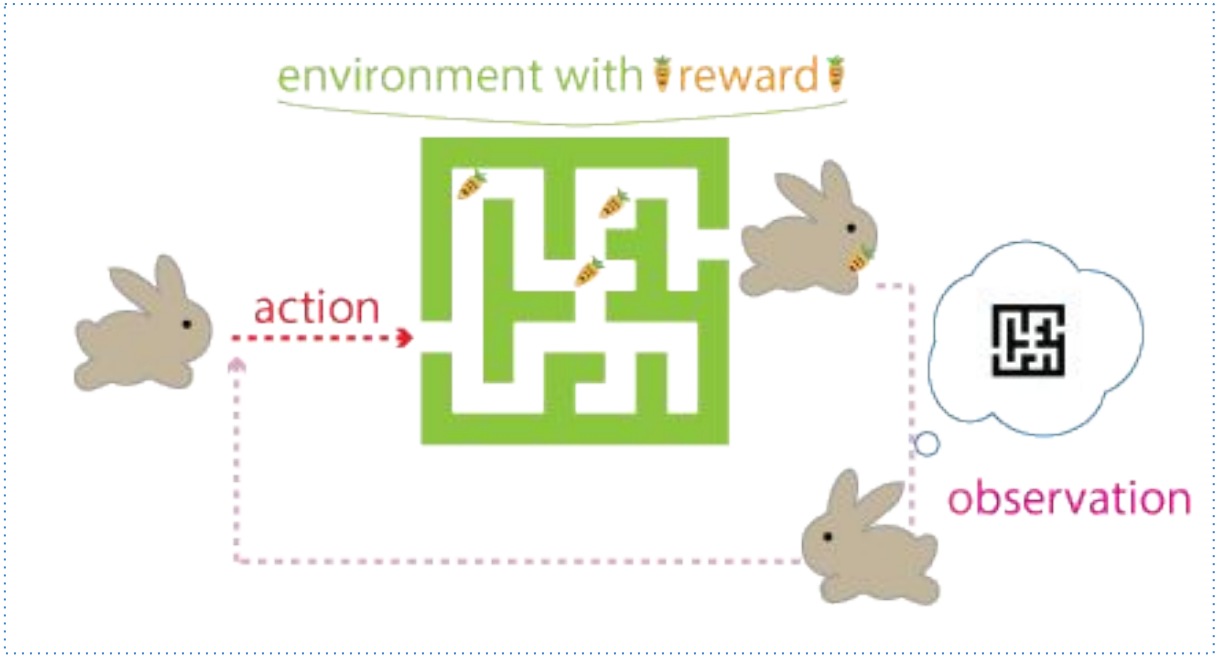
- Limited, I'm yet to find a data set…
- Code still in the AlphaToe project, it looks just like a training off MNIST example

Code: https://github.com/DanielSlater/AlphaToe/blob/master/supervised_training.py

```
actual_move_placeholder = tf.placeholder("float", (None, game_spec.outputs()))

error = tf.reduce_sum(tf.square(actual_move_placeholder - output_layer))

train_step = tf.train.RMSPropOptimizer(LEARN_RATE).minimize(error)
```

We want to beat Humans

- Use reinforcement learning to improve on our previously trained supervised network



Policy gradients

- An approach that aims to optimize a policy given a function
- Function = The reward we get from the game we are playing given the actions we take
- Policy = The choice of actions playing the game
- TensorFlow network outputs the probability of a move in a given board position
- Moves are chosen randomly based on the output of the network.
- Better moves will tend to get more reward

Policy gradients - training

```
policy_gradient = reward * move_we_actually_made * output_of_the_network
train_step = tf.train.RMSPropOptimizer(LEARN_RATE).minimize(-policy_gradient)
```

- Move_we_actually_made = [0, 0, 0, 0, 1, 0, 0, 0, 0]
- Output_of_the_network = [0.2, 0.001, 0.2, 0.09, 0.5, 0.0001, 0.0001, 0.0001, 0.0001]
- Reward = 1, -1 or 0

Over many games we start to increase the probability of selecting moves that lead to rewards

Policy gradients for Alpha Go

- Build a network with the same structure and initial weights as the supervised network
- It plays endless games against itself and historical versions of itself
- If we wins a game the weights that caused us to choose the moves we did are increased
- If we lose the game the weights that caused us to choose the moves we did are decreased

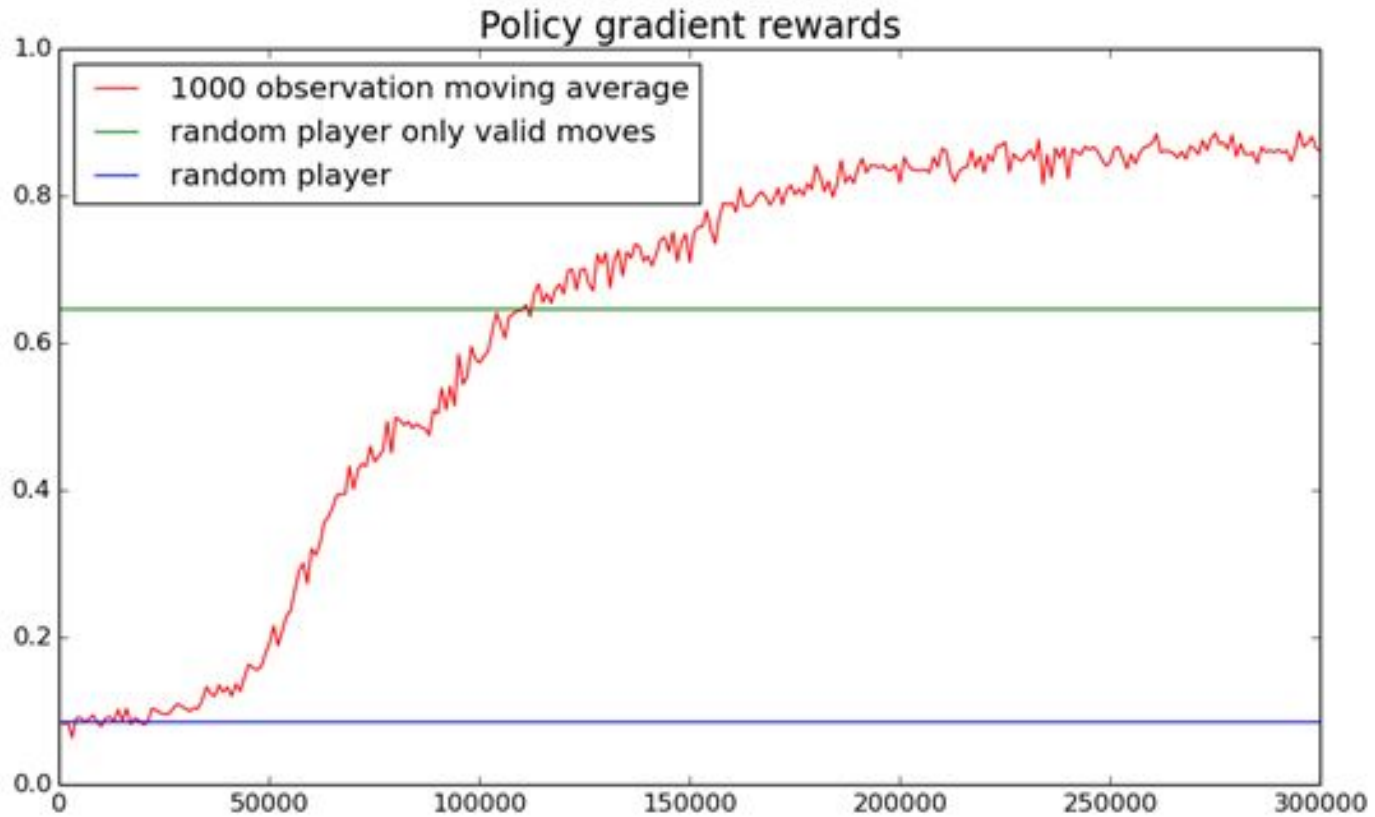
Policy gradients for Alpha Toe

Here we can start to play tic-tac-toe, selecting moves stochastically, playing against a completely random opponent

```
def get_stochastic_network_move(session, input_layer, output_layer, board_state, side):
        board_state_flat = np.ravel(board_state)
        if side == -1:
        board_state_flat = -board_state_flat

        probability_of_actions = session.run(output_layer,
                                             feed_dict={input_layer: [board_state_flat.ravel()]})[0]

        try:
                move = np.random.multinomial(1, probability_of_actions)
        except ValueError:
                move = np.random.multinomial(1, probability_of_actions / (sum(probability_of_actions) + 1e-7))

        return move
```

Code: https://github.com/DanielSlater/AlphaToe/blob/master/policy_gradient.py

Alpha Toe - Policy gradient training



Policy gradients vs historical

- Tensor flow not as well set up for this, so had to write some custom saver and loader functions
- Seemed to show improvement vs historical self

```
def save_network(session, tf_variables, file_path):
        variable_values = session.run(tf_variables)
        with open(file_path, mode='w') as f:
                pickle.dump(variable_values, f)


def load_network(session, tf_variables, file_path):
        with open(file_path, mode='r') as f:
                variable_values = pickle.load(f)
        for value, tf_variable in zip(variable_values, tf_variables):
```

Code: https://github.com/DanielSlater/AlphaToe/blob/master/policy_gradient_historical_competition.py

How Alpha Go Improves further

- Monte carlo UCT is really good, we would still like to use that to evaluate a range of possible moves
- Our reinforcement network is too slow to use to sample every move in a monte carlo rollout
- Train to predict what will be the result of the network

Value network

- Given positions from the game, learns to the result obtained by the reinforcement learning network.
- Uses Supervised learning
- Board as input => Value of the position as output
- For alpha Go the accuracy of this value network was: 0.234 - Mean Squared Error
- We can now do monte-carlo rollout using a single evaluation of the neural network as opposed to a single rollout requiring an evaluation per move (200 times improvement)

Alpha Toe - Value network

- We don’t have a database of moves still… so just generate random positions and see how our reinforcement learning does

Code: https://github.com/DanielSlater/AlphaToe/blob/master/value_network.py

Combine the value network with Monte-carlo UCT

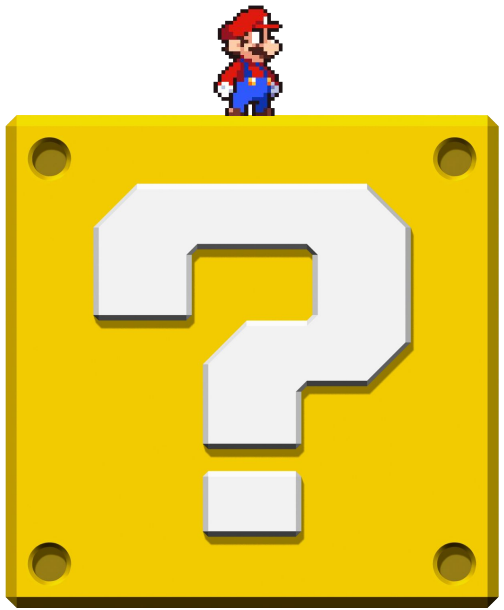
- Monte carlo UCT trees constructed using the value network as a prior
- It is then further improved by successive rollouts using a fast policy
- This resulted in best performance overall

Code: https://github.com/DanielSlater/AlphaToe/blob/master/techniques/monte_carlo_uct_with_value.py

AlphaToe

- Samples of all the different steps are now in AlphaToe Github
- Includes other examples games, larger tic-tac-toe and connect-4

Questions



Submissions welcome :)

https://github.com/DanielSlater/AlphaToe