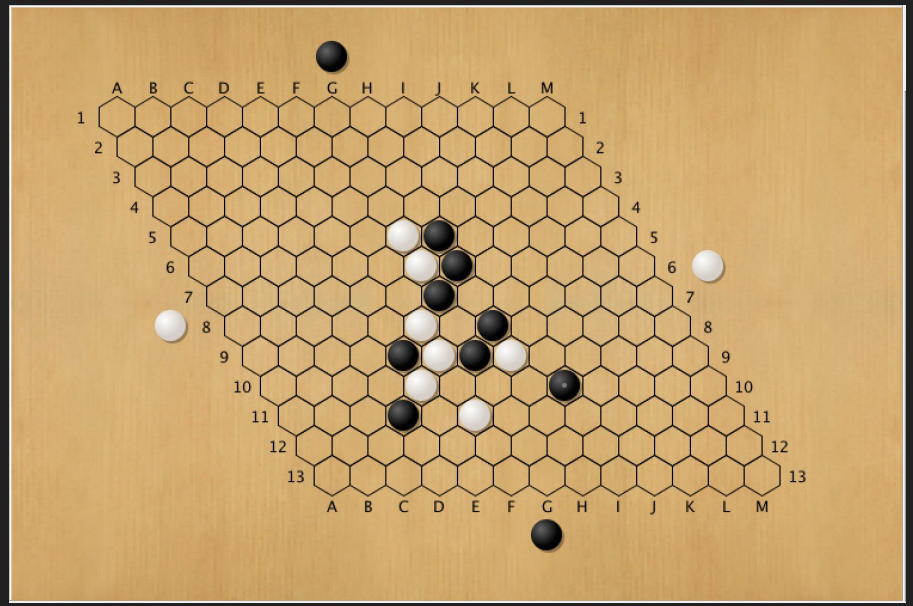
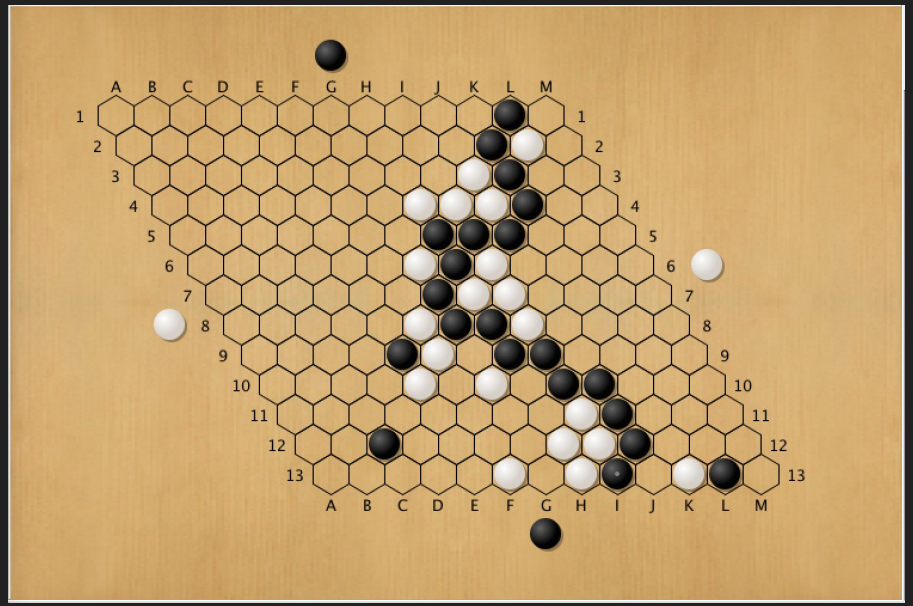
Learning to play Hex

(with deep Q-learning)

Kenny Young, Gautham Vasan and Ryan Hayward

Hex





- White and Black alternate placing stones
- Winner is the player who connects their two sides by an unbroken group of stones
- On the order of 10^80 states for 13x13

Reinforcement learning

- In supervised learning we want to learn to approximate a function from a large amount of input-output examples
- By contrast Reinforcement learning seeks to build algorithms which allow agents to learn to choose actions to maximize some cumulative reward in an initially unknown environment
- Despite the difference in goals we can often apply many of the same techniques

Basic Reinforcement learning Definition

State (S): Immediate information available to the agent at the current time-step

Action (A): Choice made by the agent from some set, based on current state, which in general affects the next state and reward received

Reward (r): Scalar value supplied by the environment after each action, the agent tries to maximize the cumulative reward (also called return) in each episode

Policy ($\pi(A,S)$): Probability distribution defining probability of taking each action from each state

Action Value ($Q_{\pi}(S,A)$): Expected sum of rewards received by the agent if they take action A from state S and then follow policy π

(Deep) Q-Learning

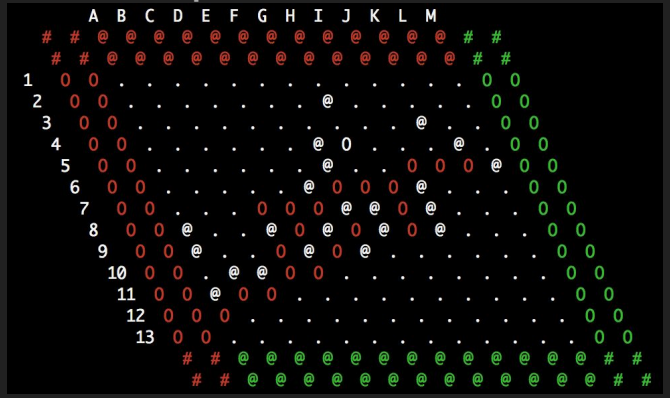
- In Q-learning the agent tries to learn the action values Q(S,A) for each state-action pair and then follows the policy which picks the optimal action in each state
- In each time step the estimated value of the action chosen is updated according to:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right].$$

- In Deep Q-learning a neural network is used to estimate Q(S,A) which takes an encoding of the state S as input and outputs a vector of values for all available actions. The update above is performed as a gradient step.

Our State Representation

- State of hex board is encoded by a tensor with 2 spatial dimensions and 6 channels as follows:
  - White stone present
  - Black stone present
  - Black stone group connected to North edge
  - Black stone group connected to South edge
  - White stone group connected to East edge
  - White stone group connected to West edge



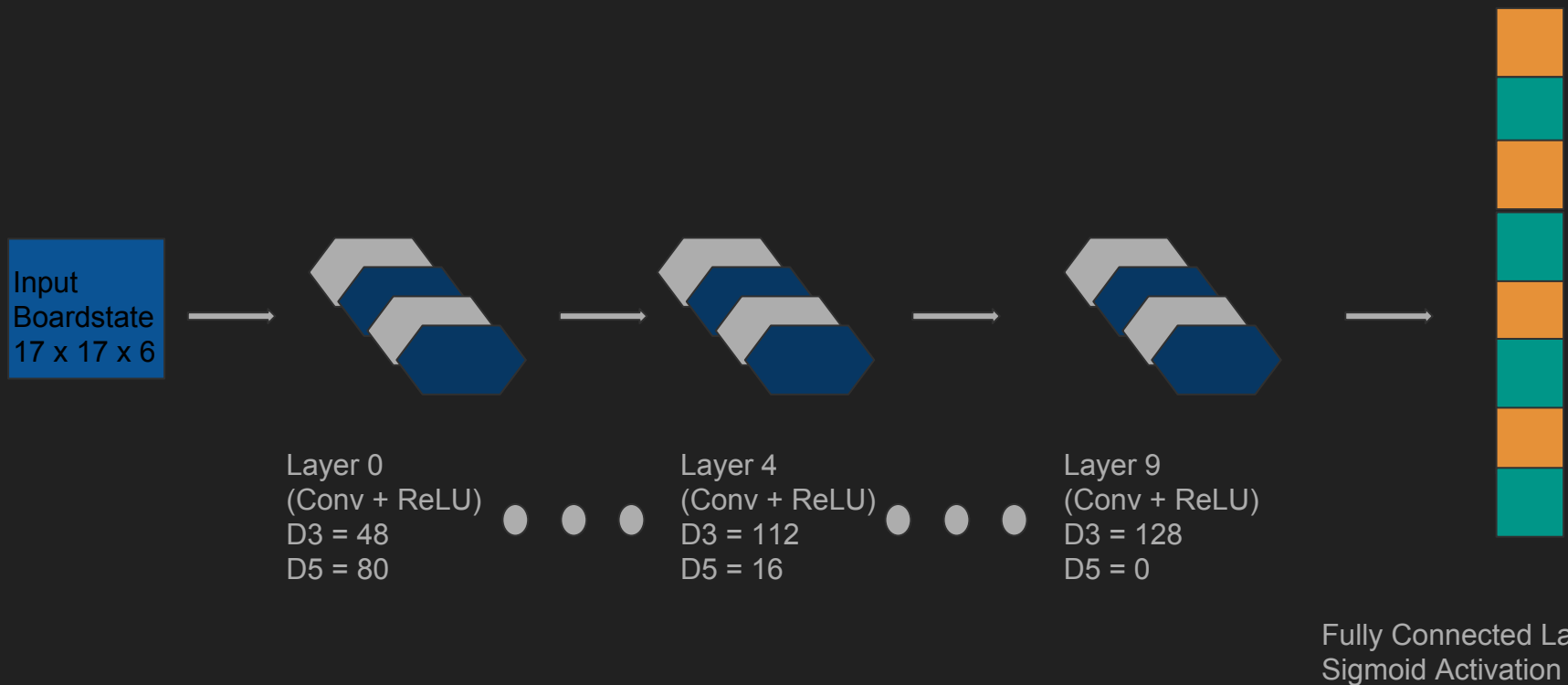
- In addition to the actual board there are 2 cells of padding on each side which are edge connected by default and belong to the player trying to connect that edge.

Our Reward Structure

- We use the convention that a win is worth +1 reward and a loss is worth -1 reward, rewards for all time-steps which do not end the game are 0
- Since we are in the “episodic case” we use a discount factor of 1 meaning a win one move ahead is just as valuable as a win 20 moves ahead
- Thus all action-values lie between 1 and -1 and correspond to the network's estimate of P(win)-P(loss)

Our Model

- Our model was inspired by that used by Deep-Mind's alphaGo
- Model consists of a convolutional neural network with 10 convolutional layers and one fully connected layer at the output
- Each convolutional layer includes some 5 by 5 and some 3 by 3 convolutions for a total of 128 filters in each layer, the ratio of 3 by 3 to 5 by 5 increases toward the higher layers
- Convolutions are hexagonal rather than square, to hopefully better capture the games notion of locality


Q(a,s) over all 169 cells
Input
Boardstate
17 x 17 x 6
Layer 0
(Conv + ReLU)
D3 = 48
D5 = 80
Layer 4
(Conv + ReLU)
D3 = 112
D5 = 16
Layer 9
(Conv + ReLU)
D3 = 128
D5 = 0
Fully Connected Layer +
Sigmoid Activation

Our Model

- Each layer outputs over the whole board in space and is zero padded to maintain the same size as the input
- All activations are relu except the output which uses a sigmoid (necessary since Q-values must lie between 1 and -1)
- No pooling layers are used

Our Q-learning Implementation

- Our implementation was inspired by Deep Mind's DQN for Atari:

initialize replay memory M, Q-network Q, and state set D
for *desired number of games* **do**
  s = position drawn from D
  randomly choose who moves first
  randomly flip s with 50% probability
  while *game is not over* **do**
    a = epsilon_greedy_policy(s, Q)
    s_{next} = s.play(a)
    if *game is over* **then**
      r=1
    else
      r=0
    end
    randomly flip s_{next} with 50% probability
    M.add_entry((s,a,r,s_{next}))
    (s_t,a_t,r_{t+1},s_{t+1}) = M.sample_batch()
    $target_t = r_{t+1} - \max_a Q(s_{t+1})[a]$
    Perform gradient descent step on Q to reduce $(Q(s_t)[a_t] - target_t)^2$
    s = s.play(a)
  end
end

Our Q-learning Implementation

- Like that work we use experience replay, wherein each State-Action-Reward-State pair is stored in a large memory from which random elements are drawn each step to perform a batch update (batch size 64 in our case)
- Our network only learns to play as white and the board is transformed such to the equivalent black state when it plays as black
- Our target differs from the one in the DQN algorithm in that the target is given as the negation of our opponents best available move in the next state rather than our own best available move (we seek to make our opponent's next state as bad as possible, as opposed to making our own as good as possible)

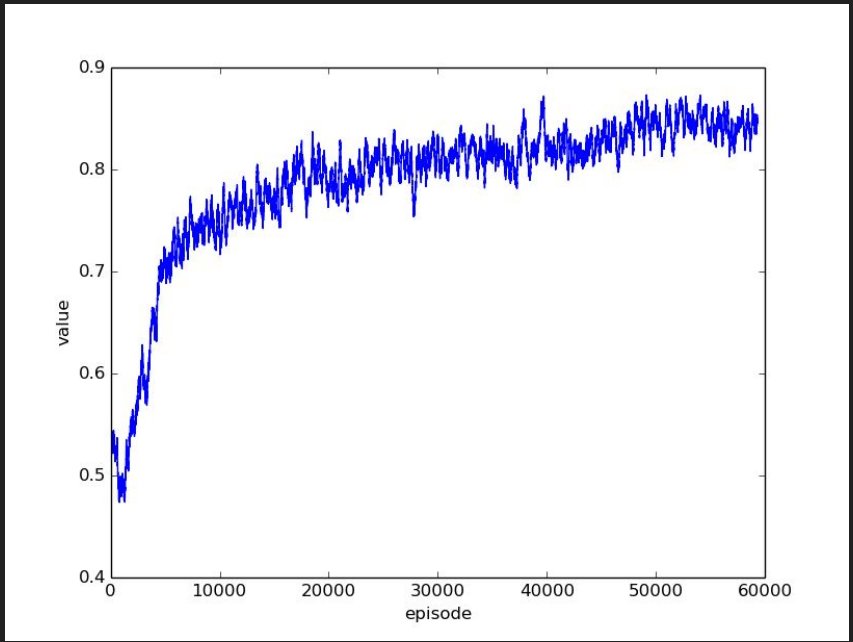
Our Q-learning Implementation

- Training proceeds by drawing a start state from a database of games played by another strong agent (called wolve)
- It then proceeds by self play until the game ends making one batch update after each move
- During self play the agent follows the “epsilon greedy policy”, playing the move with the highest Q value most of the time but occasionally playing randomly to facilitate exploration

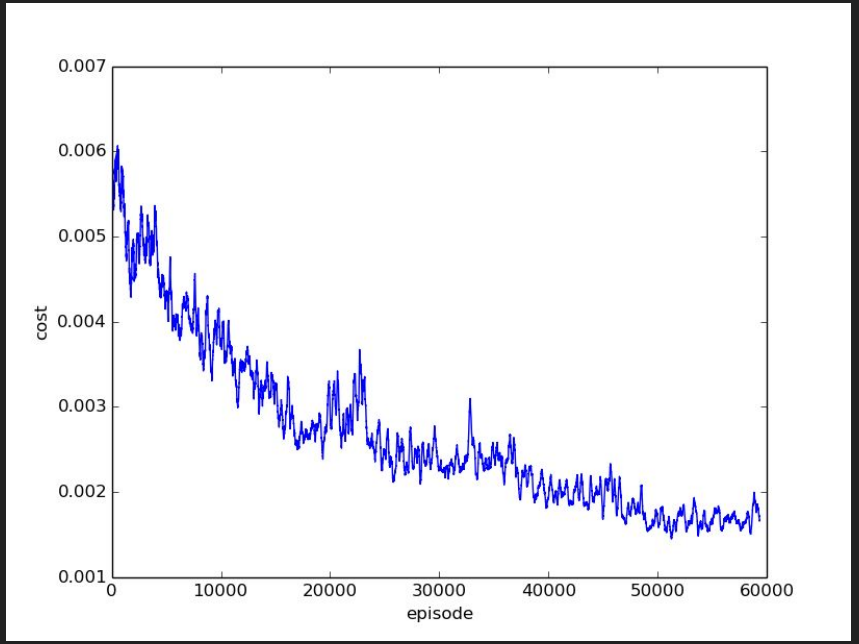
Mentoring

- Deep Q-learning is relatively slow since each step updates the evaluation of only one move and discovering strategies by random exploration takes time
- Also in our case reward is given only at the end of a game so the network takes a long time to propagate this reward back to mid/early game positions
- Thus to speed up initial training we first train the network to replicate a common hex heuristic based on electrical resistance
- This serves the dual purpose of making mid-game update steps somewhat meaningful from the very start, and providing the network with some trained filters that give some useful information about the game

Results



Average magnitude of value assigned to each position encountered during training. Roughly measures how certain the network is about the value of positions.



Average magnitude of cost (difference between target value for action and current assigned value). Indicates how rapidly the network is learning.

Results

- Win-rate of final network v.s. Mohex, a strong search-based players with 1s search time over 1000 games as white and black
  - As white (2nd move): 2.1%
  - As black (1st move): 20.4%
- Note that mohex is a very complex and highly optimized Monte Carlo tree search based player, making use of many theorems for pruning moves and detecting wins early. Though the obtained winrate is quite low we consider it a reasonable success and an indication that the network could probably be incorporated into a search routine (perhaps even into mohex itself) to produce a strong player.

Future Work

- Augment the input space with more features to give tactical information specific to the game of hex such as virtual connections, dead cells and capture patterns
- Build a search based player using the trained network or incorporate it into an existing search player like mohex
- Use an actor-critic type method to train a policy network in addition to the Q network and see how this affects training time and effectiveness
- Continue training for a longer period of time to test the limits of the learning process
- Test the necessity of drawing initial positions from a state database, as well as heuristic supervised mentoring

Thank You!