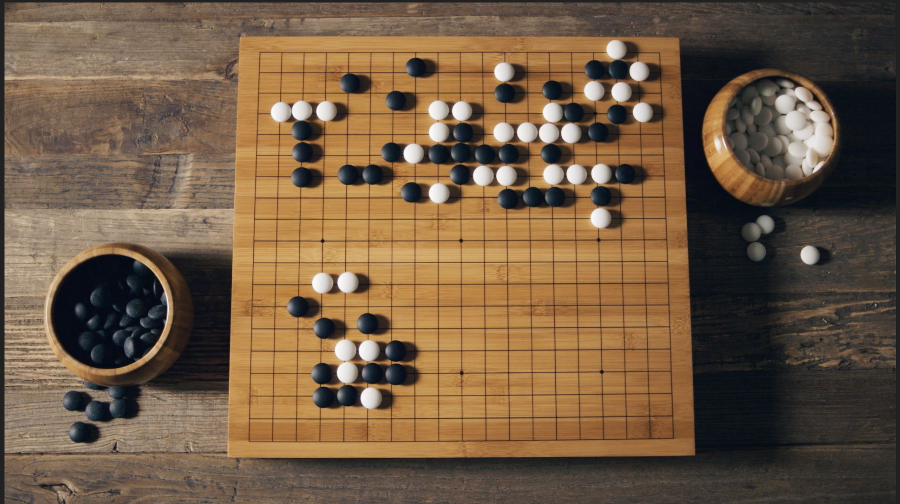
AlphaGo

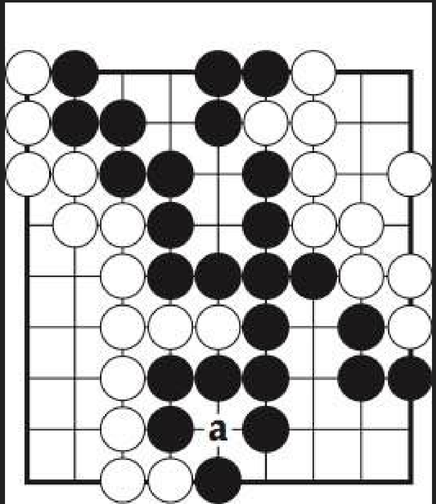
aka

[Mastering the Game of Go with Deep Neural Networks and Tree Search]

How Go works

- 3000 year old Chinese game
- 19x19 board
- Can move to *any* of the valid positions
- Your only goal is to enclose territory and capture other pieces

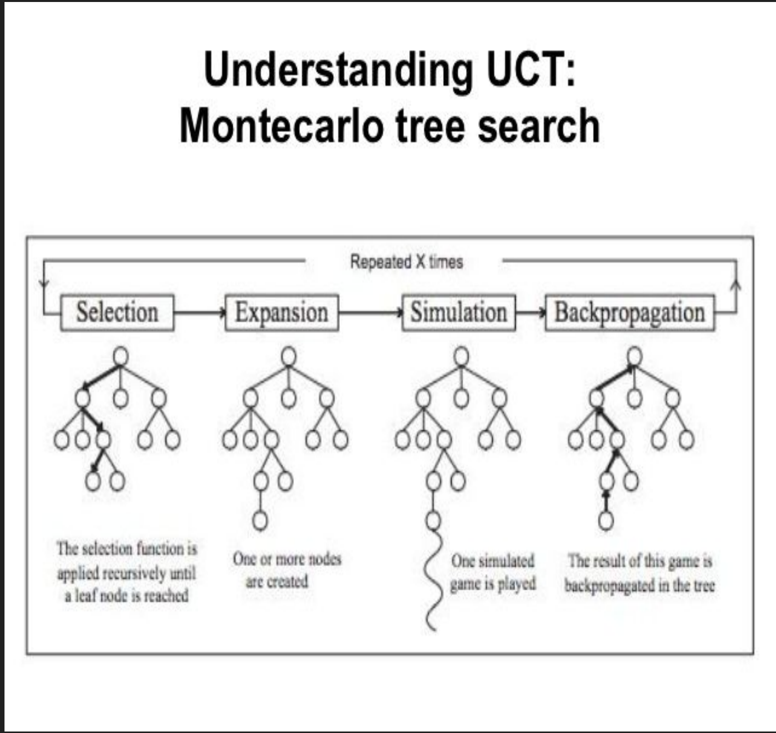




Why is this a hard game to solve for AI?

Previous AI implementations used Monte-Carlo Search

- Play out *n* iterations for position you want to find move for
- Whichever next move leads to highest success of winning the game is the choice you make
- Add one leaf node each iteration
- Each node stores its estimated value and how many times its been visited
- Selection done by some ratio of estimated value to # of visits



We can do better than basic Monte-Carlo Search

- First, they have a 13-layer convolutional network with ReLU activations predict the best move in 30 million situations
- How'd they get the data?
- What is happening start-end in the convolutional network?
- Softmax layer at the end gives probability distribution over all legal moves *a*
- σ represents the weights in this first convolutional network
- What does $\propto$ mean?
- What does this equation represent?

$$\Delta\sigma \propto \frac{\partial \log p_\sigma(a|s)}{\partial \sigma}.$$

The next stage is Reinforcement Learning

- That previous policy network was only as good as the players it emulated, we want it to be better
- How would you make it better?

The next stage is Reinforcement Learning

- That previous policy network was only as good as the players it emulated, we want it to be better
- How would you make it better?
  - Play it against random past versions of itself
    - Why random past versions?
    - How do we initialize this new network?
- How do we take error/derivatives on this?

The next stage is Reinforcement Learning

- That previous policy network was only as good as the players it emulated, we want it to be better
- How would you make it better?
  - Play it against random past versions of itself
    - Why random past versions?
    - How do we initialize this new network?
- How do we take error/derivatives on this?
  - ϱ is the weights, z_t is the +1/-1 for the win/loss at the end

$$\Delta\rho \propto \frac{\partial \log p_\rho(a_t|s_t)}{\partial\rho} z_t \,.$$

Results thus far

- Just predicting the next move to make resulted in a state of the art Go AI
    - Beat all other AI’s consistently
    - Still not better than world champions
- What’s the problem with simply picking the best move at any given moment?

We need to be able to evaluate consequences

- The third neural network was trained to evaluate the strength of a given position
- It is learning $v^p(s)$

$$v^p(s) = \mathbb{E}\left[z_t \mid s_t = s, a_{t \ldots T} \sim p\right].$$

- Our error function is MSE with an unspecified learning rate

$$\Delta\theta \propto \frac{\partial v_\theta(s)}{\partial\theta}(z - v_\theta(s)).$$

What do we predict game outcomes on?

- Can we predict the game outcomes on every move stored in our bank?
  - The ones we trained the next-move policy network on

What do we predict game outcomes on?

- Can we predict the game outcomes on every move stored in our bank?
  - The ones we trained the next-move policy network on
- No, because it leads to overfitting!
  - 0.37 MSE on test set compared to 0.19 MSE on training set
  - So how do we get enough data so that the game states are always different?

What do we predict game outcomes on?

- Can we predict the game outcomes on every move stored in our bank?
  - The ones we trained the next-move policy network on
- No, because it leads to overfitting!
  - 0.37 MSE on test set compared to 0.19 MSE on training set
  - So how do we get enough data so that the game states are always different?
- We play our next-move policy network against itself millions of times
  - Trains the policy network more too

How does it all come together

- We do Monte-Carlo Tree Search with these neural networks
  - The *value network* was used at leaf nodes to reduce the depth of the tree search
  - The *policy network* was used at all nodes to reduce the breadth of the tree search
- Each *edge* of the tree stores an
  - *Action value* Q(s, a)
  - *Visit count* N(s, a)
  - *Prior Probability* P(s, a)
- Get ready

Traversing the Monte-Carlo Tree

- Each *edge* of the tree stores an
  - *Action value* Q(s, a), *Visit count* N(s, a), P*rior Probability* P(s, a)
- a_t is the next node we choose in the current Monte-Carlo simulation (which we run thousands of!)
- Choose a_t using value network and policy network
- $V(s_L)$ is the value of a leaf node we have not explored
  - $v(s_L)$ is the value network's prediction of how good the position is
  - z_L is the result of playing a full game using our policy network
- N(s, a) is the number of times we've visited that edge
- Q(s, a) is basically a number for how good the average node in this sub-tree is

$$a_t = \operatorname*{argmax}_a \left(Q(s_t, a) + u(s_t, a) \right),$$

$$u(s, a) \propto \frac{P(s,a)}{1+N(s,a)}$$

$$V(s_L) = (1 - \lambda) v_\theta(s_L) + \lambda z_L .$$

$$N(s, a) = \sum_{i=1}^{n} \mathbf{1}(s, a, i)$$

$$Q(s, a) = \frac{1}{N(s, a)} \sum_{i=1}^{n} \mathbf{1}(s, a, i) V(s_L^i)$$

After doing this thousands of times, pick the most visited node as your next move

This process was split along 48 CPUs and 8 GPUs

- GPUs for doing the convolutional application
- CPUs for doing Monte-Carlo Simulation

After training this exhaustively...

- It beat the world champion 4-1 in a million dollar match
- Made some crazy moves along the way

Next week: GANs