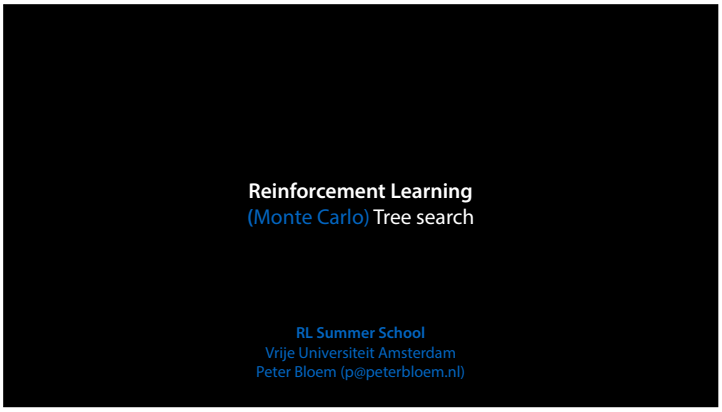


So far, we've assumed that we have no control over the environment we're learning in. All we can do is take an action, and observe the result.

This is not always true. In many cases, we have some, or even perfect access to the transition function and the reward. Consider, for instance the case of playing a game like tic-tac-toe or chess against a computer opponent. We don't have to play a single game from start to finish, never considering alternatives or trying different approaches. We can actually explore different paths and try different approaches to see what the consequences are.

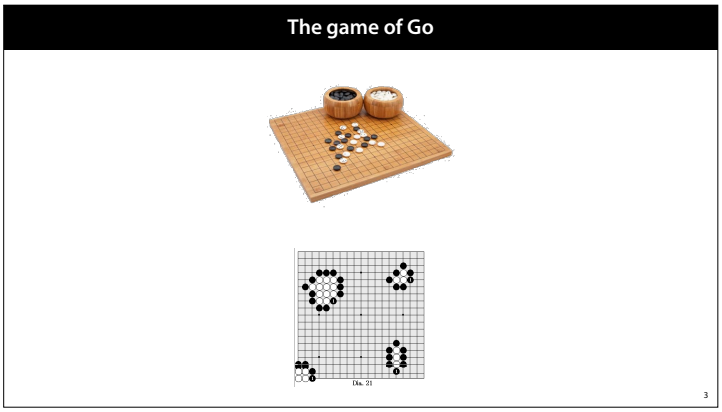
We can use this during training to try and explore the state space more efficiently. We can also use it during *in production* (for instance when we are playing a human opponent) to make our policy network more powerful: we try different moves observe what a computer player would do, and search a few moves ahead. In general, this is a good way to improve the judgements made by a policy network.

In general, we'll call such methods tree search. From the perspective of the agent, the space of possible future scenarios has the shape of a tree: all the actions we can take, all the states that can follow those actions, all the actions we can take in all of those states and so on. If we have access to the state transition function, or a good simulation of it, we can use that to explore the state space ahead of us a little bit before comitting to an action.

|section|Tree search|
|video|https://www.youtube.com/embed/R4souHAdRP4|

The combination of deep reinforcement learning and tree search has led to one of the most important breakthroughs in AI in recent years. In 2016 AlphaGo, a Go playing computer developed by the company DeepMind beat Lee Sedol, one of the best players in the world. Many AI researchers were convinced that this AI breakthrough was at least decades away.

image source: http://gadgets.ndtv.com/science/news/lee-sedol-scores-surprise-victory-over-googles-alphago-in-game-4-813248



Since we're using Go as a target for these methods, here is some intuition about how Go works. The rules are very simple: players (black and white) move, one after the other, placing stones on a 19 by 19 grid. The aim of the game is to have as many stones on the board, when no more stones can be placed. The only way to remove stones is to encircle your opponent.

The general structure of this game is the same as tic-tac-toe or chess: it's:

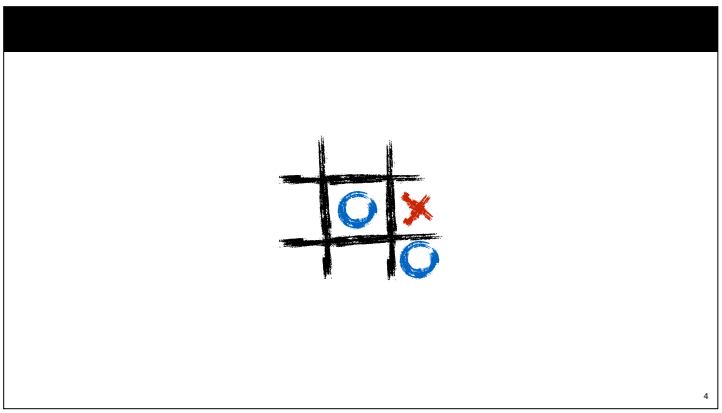
- **two player**, **turn-based**
- **perfect information**, both players can see all there is to know about the state of the game by looking at the board.
- **zero sum** if one player loses, the other wins and vice versa. If a state is good for one player it is precisely equally *bad* for the

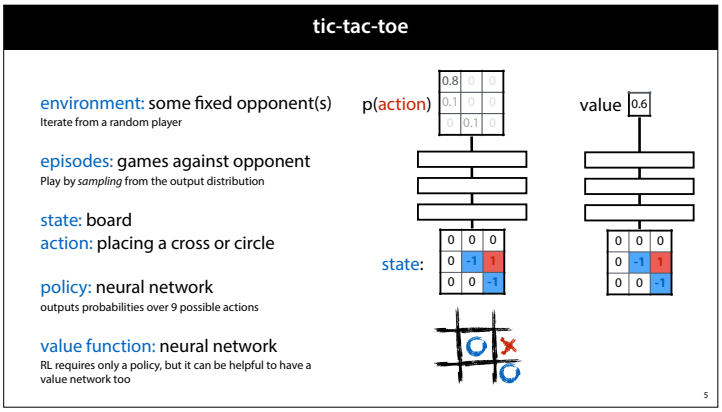
other player.

The reason Go was considered so difficult to solve compared to chess was simply that the game tree was so broad and deep: at any given point a player must choose between hundred of possible moves and and a game has 211 turns on average. Compare this to chess, which has about 20 possible moves at any one point and lasts about 40 moves on average.

It's sometimes said that the sheer number of possible positions in Go is larger than the number of atoms in the universe and that that is what. makes it so difficult. This is partly misleading and partly false. The number of possible chess positions is also vast (10^{46} possible distinct positions and 10^{120} nodes in the game tree, with the number of atoms in the universe somewhere in between) and we managed to solve that just fine without any learning at all.

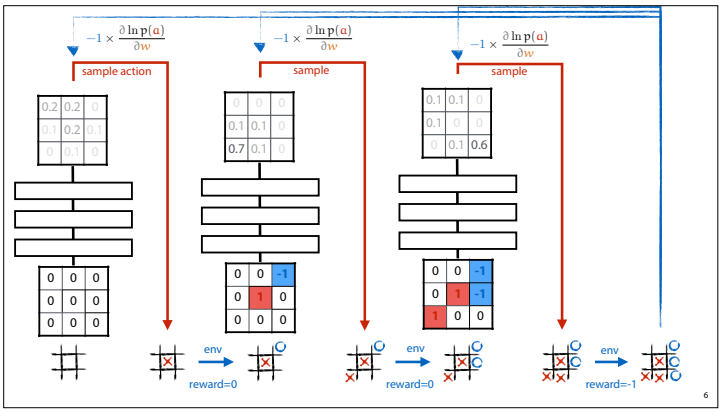
What makes Go so difficult is partly it breadth. The number of possible moves per turn is what makes it impossible to search the full tree even two moves ahead moves ahead. Moreover, humans seem to use a kind of visual intuition to break through this complexity which is very hard to capture in simple rules, which suggests that learning may be a worthwhile approach.





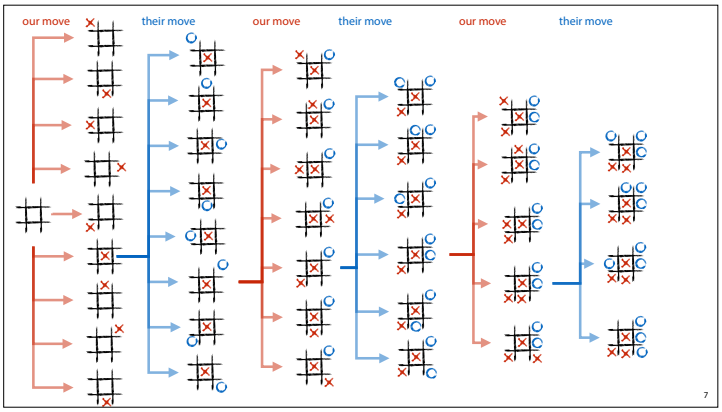
To finish up, let's see what this looks like in our our tic-tac-toe example.

In principle the only requirement for the value network is that the better the state is (according to the network), the higher the value, but in practice, a good way to define the value is to make it an estimate of the expected reward from the given state, *using the current policy*.



This is what we would do in a regular RL setting, where we don't know anything about the environment.

In the case of a full information game, however, we actually have access to the whole state graph. We already know all the rules of the environment. We can use this to our advantage.

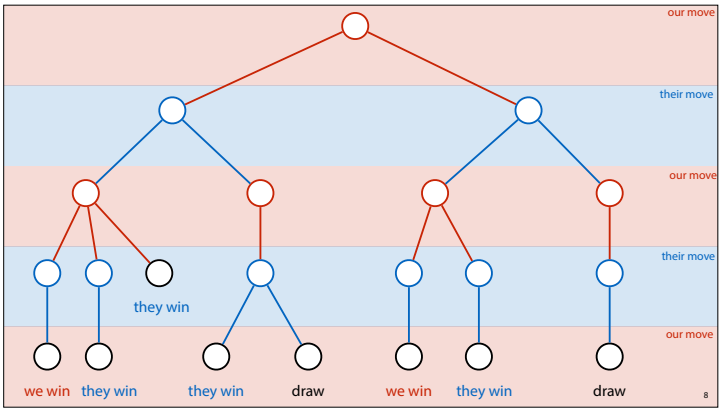


The main concept we will be building on in this section is the **game tree**. This is a tree with the start state at the root. Its child nodes are the states that can be reached in one move by the player who moves first. For each of these children all their children are the states that can be reached by the player who moves second, and so on until we get to the leaf nodes: those states where the game has ended.

Even for a game as simple as tic tac toe, the full game tree is too big to show in a slide like this. What we've shown here is just a small part of the full tree.

The key idea to tree search methods is that by exploring this tree, from the node representing the current state of the game, we can reason about which moves are likely to lead to better outcomes.

This is similar to the state space we get when we cast this as a reinforcement learning problem but not quite the same. In that case we only see states where the opponent has moved, so only half of these nodes are states the the RL agent would observe.



Since even the tic tac toe game tree is too complex to fully plot, we will use this game tree as a simple example. It doesn't correspond to any particular realistic game, but you can hopefully map the idea presented here to the move realistic game trees of tic tac toe, chess and go.

Tree search

- Rollouts
- Minimax
- Monte Carlo Tree Search

How to:

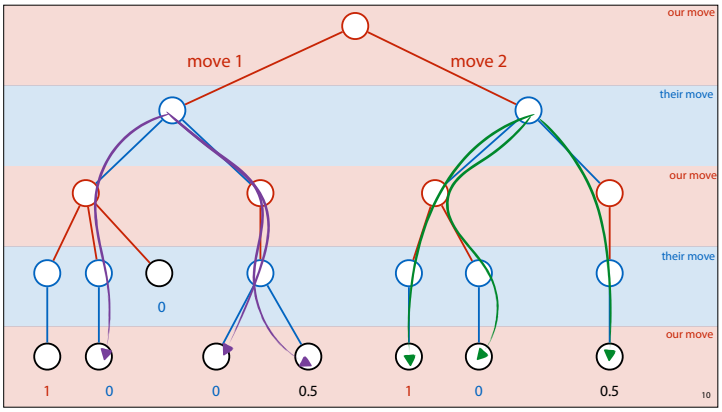
- use them without any learning,
- use them **after** you've learned a policy and value net,
- use **to** learn apolicy and value net.

9

We'll look at a few simple methods of tree search. These by themselves are not reinforcement learning methods. They aren't even learning methods in any meaningful way. All of them just explore the game tree as much as possible, and try to come up with a good move.

This is how many of the earliest game playing engines worked: they just search the game tree from the current state, return a good move and play it. The opponent plays their move, and they start the whole process again.

For each of them we will first see how they work by themselves, and then we will see how we can use them to improve an existing policy during play, and how to use it during training to improve a policy network.



We start with a simple, but powerful idea: **random rollouts**.

First, we label the leaf nodes with their value. This is 1 if we win in that node, 0.5 if there is a draw, and 0 if the opponent wins.

In the previous sections we used -1, 0 and 1 as rewards, but the difference is arbitrary for almost all algorithms. The current values serve to make the exposition clearer when we get to the MCTS algorithm.

In this picture, we have the next move, so we need to decide between move 1 and move 2.

The way random rollouts work is that for every node we reach by making one of the moves we're considering, we simply simulate a series of random games starting at that node. This means that we just play random moves for both players until we reach a leaf node. This is called a **random rollout**.

We then average all the values we get at the end of each rollout per starting node. In this example, we get -2/3 for move 1, and 0 for move 2. We take this as estimates for the values of the two nodes we reach by playing the two moves.

In this case, the node following move 2 gives us the highest estimated value, so we choose to play move 2.

---

random rollouts

```
given start state s
for all possible moves a:
    let s' be the result of playing a
    repeat N times:
        simulate a full random game from s'
        play random moves for both opponents
        observe outcome: 1 for win, 0 for draw, -1 for loss
play the move a that led to the highest average outcome
```

11

Here is the algorithm in pseudocode.

It may seem a little mysterious why random rollouts work at all against a non-random player, since these random games will be so different. One way to think about it is that we're evaluating different *subtrees.* If the subtree below node 1 has many more leaf nodes where we win, than the subtree below node2, then it can't be *too* bad to move to node 1, since at the very least there are many opportunities to win from that node.

Perhaps the opponent is too smart to let us get to any of those opportunities, but for such a simple method it gives us a pretty good opportunity.

12

Here is an illustration for why even such random play can be informative. Chess is a particularly illustrative example, since random play is so far removed from what a good player would do. Therefore, how could a series of random plays tell us anything about what intelligent players would do from a given state?

In this chess position, black has just made a tremendous blunder, by moving its queen in the path of the white knight. White can take the queen with no repercussions. Can random rollouts identify that taking the queen is a good move?

If white doesn't take the queen, and moves, say, one of its pawns instead, all the rollouts from that point are games with equal material, and they will likely all end in a draw. If white takes the queen, all random rollouts are played with a massive material advantage for white, and even though most of them will still end in a draw, the probability that we will see a checkmate increases. With enough random rollouts, we should be able to tell the difference.

Of course, the difference is still minimal, and for such an obviously good move, we'd like to draw our conclusions a little quicker.

---

policies and value functions

policy function: p(a|s)

Used to simulate an agent or assign probabilities of winning to a state.

value function: V(s)

Used as a *heuristic* to value particular states during search or to estimate the expected outcome.

13

To improve rollouts, and tree search methods, we can introduce *policies* and *value functions*.

For now, we won't assume that these are neural nets, so we don't need to worry about training them. You can imagine using a simple handwritten policy and value function that isn't great, but is probably better than picking random moves.

For instance, we could define a policy function that plays entirely randomly, except that it assigns a little more probability to moves that capture a piece. Likewise with the value function: we could write simple value function that assigns -1, 0 and 1 to lost, drawn and won states respectively, but assigns values in [-1, 0] for states where the opponent has a material advantage, and states in [0, 1] for states where we have a material advantage.

Another difference is that policies in reinforcement learning were only defined from the perspective of one player. There are simple fixes for this. If we have a policy for white in chess, we can, for instance simply invert the board (making white pieces black and vice versa) to get a policy for black. The precise details depend on what the implementation looks like, but the zero sum nature of these games means that a good policy for one player is always automatically also a good policy for another player. The move player 1 likes a move, the less player 2 likes it.

random rollouts

given start state s

for all possible moves a:

let s' be the result of playing a

repeat N times:

simulate a full r... from s'
play random moves for both opponents

observe outcome: 1 f... loss

play the move a that led to the highest average outcome

If we now look at the random rollout algorithm again, we see that we are implicitly already using a very simple policy and a very simple value function. The policy function we are using is simply a fully random one. And the value function is one that only assigns non-zero values to leaf nodes.

We can now improve the algorithm by replacing these with a better policy function and a better value function to improve the rollouts.

---

rollouts with a policy and a value function

with a policy function:

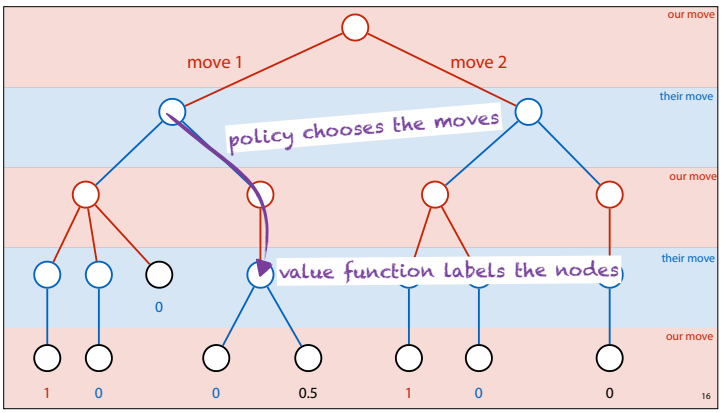
Instead of playing random moves, sample moves from the policy.

with a value function:

Limit the rollout depth, and label nodes with the value from the value function.

We replace the random moves with moves sampled from the policy. As noted before, it's usually a simple matter to turn a policy for player 1 into a similar policy for player 2.

If we have a value function, what we can do is limit the depth of the rollout (either to a fixed value or a random one). This allows for faster rollouts, meaning that we can do more rollouts in the same time, but it also allows us to recognize that we have an advantage without going through all the highly particular steps of the endgame. In particular in a game like chess, it's unlikely that a mostly random policy will find a checkmate, but if we have a strong material advantage, the value function can let us know that much earlier.



---

rollouts during play

Train the networks normally
Using policy gradients, Q-learning, random search, etc.

Use them in the rollout algorithm to *improve* the policy
The rollout algorithm should pick a *better* move than the policy network by itself

Ok, so let's imagine we've managed to train up a policy network and a value network somehow. How do we use the idea of rollouts?

The first idea is to use it **during play**. That is, when we're training, we don't use tree search at all, but when the time comes to face off against a human player, we take our policy network and we take our value network and we put them into the rollout algorithm. Then **we play the moves that the rollout algorithm returns**.

The idea here is that we could simply play whatever the policy network suggests directly, but with the right hyperparameters, the rollout algorithm should usually do better than the plain policy algorithm it uses internally. **We use the rollout algorithm to *improve* the policy.**

This is more or less how the first AlphaGo worked. It used a different tree search algorithm (which we'll discuss later), but the basic idea was the same: during training use policy gradients and simple reinforcement learning to train a policy network and a value network, and then during play, use those inside a tree-search network.

---

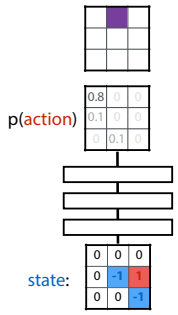
rollouts during training

Given a policy p.

Generate a realistic game state **s**.
For instance by letting p play a game against some opponent (maybe itself).

Do rollouts from **s** to estimate values for all actions.

- Train the value network to mimic these values
  Instead of the ones it now predicts (L2 loss).
- Train the policy to predict what rollout will chose to do
  Log loss: minimize $-\log p(a)$ for the chosen action a.



18

---

rollouts during training

Tree search functions as a **policy improvement operator**.

init policy p_0 randomly

loop t = 0, 1, 2, ...:

train p_{t+1} to mimic rollout(p_t)

We're looking for **the fixed** point of the improvement operation.

19

In contrast to this approach we can also use the tree search **during training**. The key insight here was already stated in the last slide rollouts are a way to improve your policy. In fancy terms: they are a policy improvement operator.

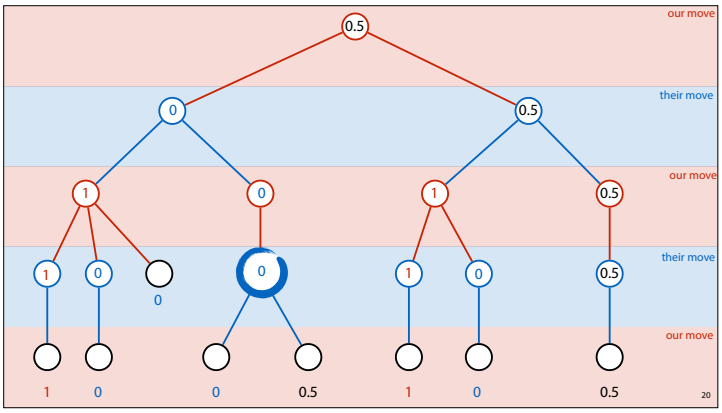
That is, if we trust that the move chosen by rollouts with our policy is always better than the move chosen by our policy alone, then we can use the move chosen by the rollout algorithm **as a training target**, for the next iteration of our algorithm.

That is, starting with policy p_0, we train the next policy p_1 to mimic what the rollout algorithm does when augmented with p_1. When this learning has converged (or simply afetr a few steps), we discard p_0, and train a new policy p_2 to mimic what the rollout algorithm does when augmented with p_1, and so on.

For the value network we can do the same thing. The average value over all the rollouts should be a better value function than the value function we start with, so we can train the next value network to mimic the average values returned by the rollouts using the old value network.

One benefit of this approach is that it stops working when we have found a **fixed point** of the policy improvement operator. If the rollout algorithm returns the same move probabilities and values as the policy and value networks we started out with, the policy improvement operator has become useless, and the policy by itself contains everything we need. This means that (if we can be sure we've reached a fixed point), we can actually *discard* the tree search during play, and play only with the policy network, which is a lot faster.

The idea of using tree search as a policy improvement operator during training was introduced in AlphaZero.



The next tree search algorithm we'll look at is called minimax. The basic idea here is that if we had sufficient compute to search the whole tree, we should be able to play perfectly: if it's possible to guarantee a win, we should win.

Assuming that we can search the whole tree, how should we choose our move? The idea of minimax is that the player whose turn it is labels each node **with the best score they can guarantee from that node**. For us, this is the maximum score, and for the opponent, this is the minimum score.

This is why the algorithm is called minimax: we are maximizing the score, and the opponent is minimizing the score.

For the nodes at the top we have no idea what we can guarantee, but for the nodes one step away from the leaves, it's easy to see. In most of these nodes, it's the opponent's turn, so we know that if we hit these nodes, there is only one move left, and the opponent chooses that. In short, whatever the lowest outcome is among the children, we know that the opponent can guarantee that.

For instance, in the highlighted node, there are two children, with outcomes 0 and 0.5. The opponent prefers the minimum, so we know that from this node, the opponent can guarantee an outcome of 0, and there's nothing we can do about it. Unless the opponent plays sub-optimally, we know the value of this node is 0.

With this logic, we can label all nodes that are one opponent move away from the leaf node. No matter what we do, if the opponent plays their best, this is the outcome. Note that some of these nodes still have a value of 1. If we maneuver the opponent into this state, we've already won. Even though they still have a move left, there's nothing they can do to avoid us winning.

Now that we know the value of these nodes for a fact, we can move up the tree. This time it's our turn. For every parent of a set of nodes whose value we know, we simply label it with the maximum of all the values of the children. Again, in some cases, we cannot avoid a loss, despite the fact that we're in control.

Note that we're always taking the maximum or the minimum. Unlike in rollouts, where we were averaging over all nodes visited, here only one leaf node ultimately decided the value of the internal nodes.

Moving up the tree like this, we see that despite the fact that there are many branches where we can force a win, if the opponent plays optimally, the can guarantee that we never visit those branches. Unless we get lucky and they make a mistake, the best we can do is to force a draw.

---

minimax (full search)

```
function minimax(s):
    if s is a leaf node:
        return value(s)
    values = []
    for a in moves(s):
        let s' be the result of playing a in s
        add minimax(s') to values
    return max(values) if my_turn(s) else min(values)
```

21

Here is a recursive, depth-first implementation of minimax.

Usually breadth-first is a more flexible way to implement minimax, but this leads to the simplest pseudocode. In practice you can also work out that certain parts of the tree don't need exploring (because some player can already guarantee a better score somewhere else that that part of the tree can offer). This is called alpha-beta pruning.

using a policy and value function

with a value function:

Set maximum depth. Minimax the values from the value function.

with a policy function:

Ignore low-probability nodes (beam search), sample next node to expand based on policy.

22

In practice, tic-tac-toe is about the only game for which you can realistically search the whole game tree. In practice, we limit our search to a subtree.

The traditional way to do this is with a value function. We search the whole tree but only up to a maximum depth. At this depth, we use the value function to label the nodes, and then work these back up the tree.

This simple algorithm formed the basis for Deep Blue, the first chess computer to beat a grandmaster. IBM simply spent a lot of time developing a very strong, hand tuned value function, and then built custom hardware to implement the minimax algorithm very efficiently.

It's less popular in combination with minimax, but we could also include a policy function here. This could, for instance allow us to prioritize certain nodes over others, searching them first. In real chess matches, time is a factor, so chess computers need to search as much of the tree as they can, within a particular time limit. A policy function can help us determine which moves are more likely to yield good results, so we can search different parts of the tree to different depths.

---

minimax with learned policies and values

minimax during play:

Use the policy and value networks to search a subtree of the gametree.

minimax during training:

Use minimax as a policy/value improvement operator.

23

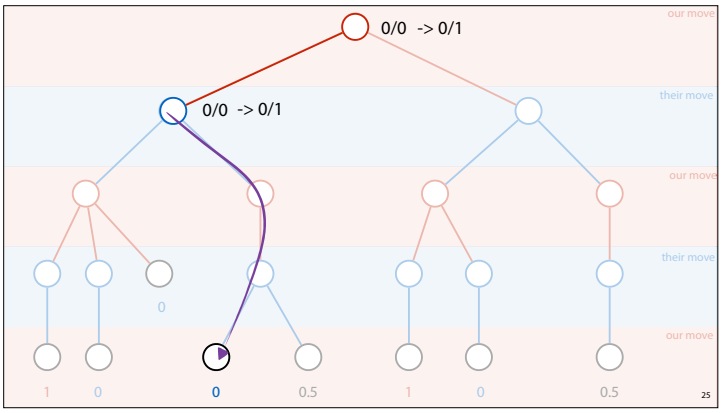
If we have learned policies and value functions, we can use the same approaches we used before. We can train the policy and value network using basic reinforcement learning, and then during play, given them a little boost by using them to search the game tree with minimax.

But, we can also use minimax as a policy improvement operator. If we can trust that the moves chosen, and the values assigned by minimax are really better than those of the plain networks by themselves, we can simply set them as targets for a new iteration of the policy and value network.

MCTS

As you may have concluded yourself already, minimax and rollouts are at something of a spectrum. Minimax searches the whole tree. Using a value function and a policy, we can limit this search to a subtree. Rollouts is the most extreme case of searching just a subtree: we search only a single path, but, we do it multiple times and average the results. We can, of course, come up with a variety of algorithms that are somewhere in between: always searching subtrees probabilistically, and repeating the search to reduce variance.

One of the more elegant algorithms to combine the best of both worlds is **Monte Carlo Tree Search** (MCTS). This is the basic algorithm that was used to beat Lee Sedol at Go.



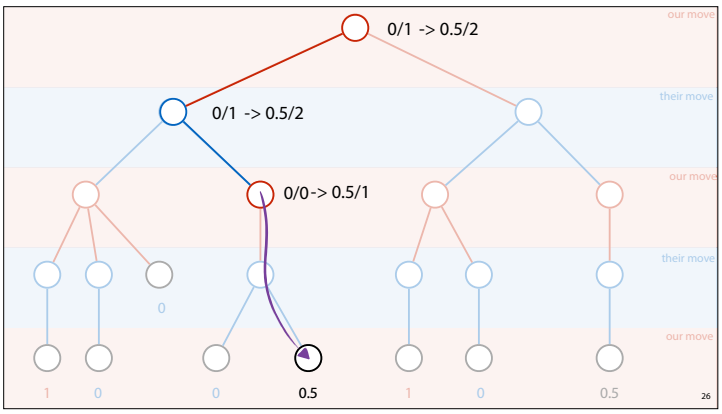
Here's how MCTS works.

We will build a subtree of the game tree in memory step by step. At first this will be just the root node, which we will extend with one child at a time. Each node, we will label with a probability: the times we've won from that node, over the total times we've visited that node. At the start this value is 0/0 for the root node, and there are no other nodes in the tree.

We then iterate the following four steps

- **Selection:** select an unexpanded node. At first, this will be the root node. But once the tree is further expanded we perform a random walk from the root down to one of the leaves.
- **Expansion:** Once we hit a leaf, we add one of its children to the tree, and label it with the value 0/0
- **Simulation:** From the expanded child we do a rollout.
- **Backup**: If we win the rollout let v = 1 otherwise v = 0. For the new child and every one of its parents update the value. If the old value was a/b, the new value is a+v / b+1. The value is the proportion of simulated games crossing that state that we've won.

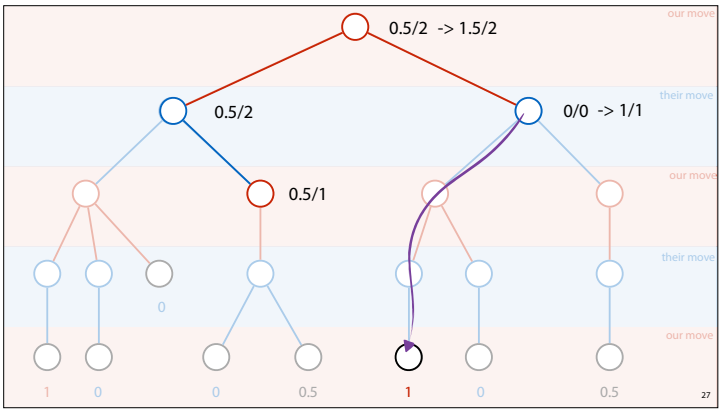
Backup is sometimes. called backpropagation, which is not to be confused with the backpropagation used in neural networks.



In the next step we expand another node. This could be any child of a node already expanded. Currently, we have three options, the two children of the node we just added, of the second child of the root node.

We could choose a node uniformly at random, or according to the values of the nodes we have so far established. This is an exploration/exploitation tradeoff, and a large part of using MCTS effectively boils down to making this tradeoff carefully. The most common techniques are too technical for this course, but ***the wikipedia article on MCTS*** *provides some pointers.*

We proceed as before: we do a random rollout from the new node, observe whether we've won the rollout and update the values of all ancestors of the current node: we always increment the number of times visited by one, and the number of wins only if we won the rollout. If we drew (as in this case), we increment by 0.5.



In the next iteration, we again add a node. Note that we again have three options for nodes to add. We pick the second child of the root node.

We do another rollout and this time we win.

Note how the values are backed up: only the newly expanded node and the root note change their values, but **not the other two nodes in the tree**.

You can think of the values on each node as an estimate of the probability of winning when starting at that node. The other two nodes were not part of this path, so their estimates aren't affected by the trial.

---

monte carlo tree search

starting with a single node with value 0/0

loop:

select a node n to add to the graph

rollout from n

update n and all ancestors
Increment the denominator by 1 and the numerator with the result of the rollout.

28

After iterating for a while (usually determined by the game clock), we have both a value for the root node, and an idea of what the best move is (the one that leads to the child node with the highest value).

MCTS with policies and values

with a policy function:

Instead of playing random moves, sample moves from the policy.

with a value function:

Limit the rollout depth, and label nodes with the value from the value function (these need to be win probabilities)

same as the rollout algorithm

29

The way we insert a policy function and a value function into MCTS is the same as it was for the plain rollout algorithm. We replace the random rollout with a policy rollout and we use the value function by limiting the rollout depth and backing up the values provided by the value function. This works best if the values can be interpreted as win probabilities (i.e. they're between 0 and 1).

---

minimax with learned policies and values

MCTS during play:

Use the policy and value networks to search a subtree of the gametree.

MCTS during training:

Use MCTS as a policy/value improvement operator.

30

Again, we can use MCTS during play to boost the strength of the learned policy and value networks. This is what the original AlphaGo did when it beat Lee Sedol in 2016.

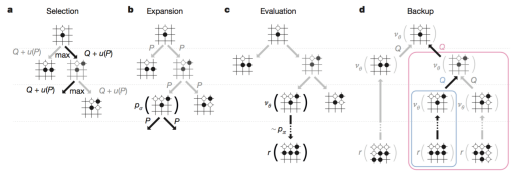
But, as before, we can also view MCTS as a way of improving a given policy and value function. The strategy is exactly the same as before, each new iteration is trained to mimic the behavior of the MCTS used with the previous iteration.

AlphaGo (2016)

Start with imitation learning
Learn to copy human players

Train by playing against previous iterations and self
update weights by policy gradients

Boost network performance by MCTS



31

These were the basic ingredients of the first AlphaGo. It used two policy networks, a fast one and a slow one, and a value network. These were first trained by imitation learning from a large database of Go games, and then by self play, using reinforcement learning.

After training, during gameplay, the performance was boosted by using the policy and value networks in a complicated MCTS algorithm.

AlphaGo Zero (2017)

Learns from scratch, no imitation learning, reward shaping etc.
Also applicable to Chess, Shogi…

Uses three tricks to simplify/improve AlphaGo

1. Combine policy and value nets
2. View MCTS as a *policy improvement operator*
3. Add residual connections, batch normalization

32

AlphaGo Zero (2017)

trick 1: Combine policy and value nets.

trick 2: Use MCTS during training as a policy improvement operator.

trick 3: Use residual connections and batch normalization.

33

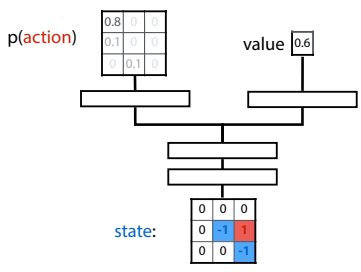
In 2017, DeepMind introduced an updated version: AlphaZero. The key achievement of this system is that it eliminated imitation learning entirely. It could learn to play go entirely from scratch by playing against itself. DeepMind also showed that the same tricks could be used to learn Chess and Shogi (a Japanese game that is similar to chess).

Deepmind indicated in their paper that these were the main improvements they introduced in AlphaGo.

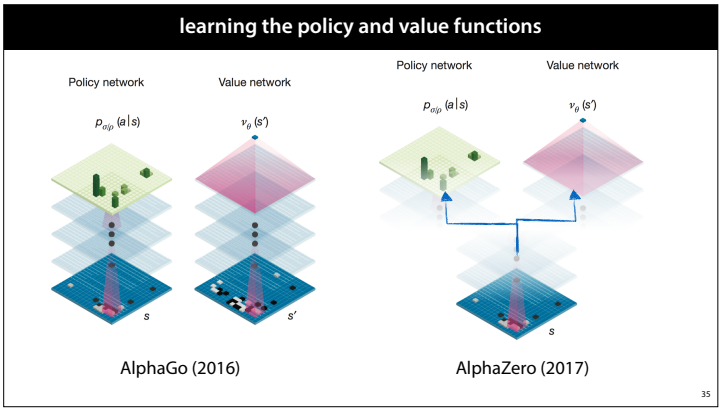
The first two, we have already discussed.

Residual connections and batch normalization are two basic tricks for training deeper neural networks more quickly. It is likely that they simply weren't available or well enough understood at the time of the first Alphago. These are not specific to reinforcement learning, and can be used in any

two-headed beast



34



Of course, these policy and values functions don't need to be hand-written. They can also be **learned**. And this is where we start to connect reinforcement learning to tree search.

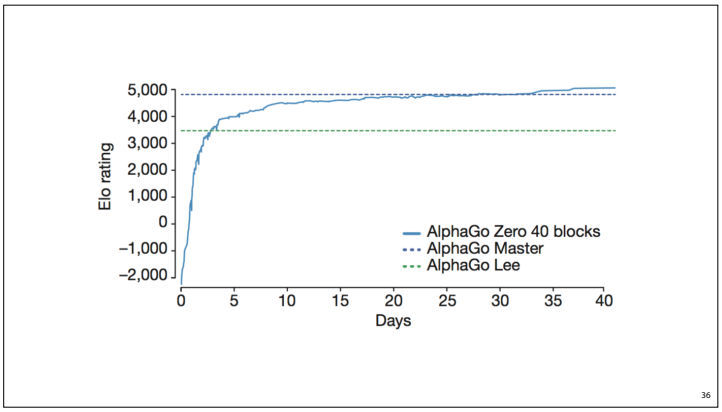
These are rough illustrations of the network structures that DeepMind used for their first AlphaGo instance. Both treated the Go board as a grid, passing it through a series of convolutional layers. The policy network then outputs the same grid, softmaxing it to provide a probability distribution over all the positions where the player can place a stone. The value network uses an aggregation function to reduce the output to a single numerical value.

Using the methods we've already discussed, like policy gradients, q-learning, and imitation learning, these networks can then be trained to provide a decent, fast player (the policy network) and a good indication of the value of a particular state.

In Q learning we trained a policy network and a value network in one. We didn't discuss how to train a value network through policy gradients. The idea used by DeepMind in the first AlphaGo was that the value network V predicts for state s the value of the game played from s by the policy against itself. That is, we simply observe a game of the policy network playing against itself and afterwards we assign the result as the target that V should predict for s.

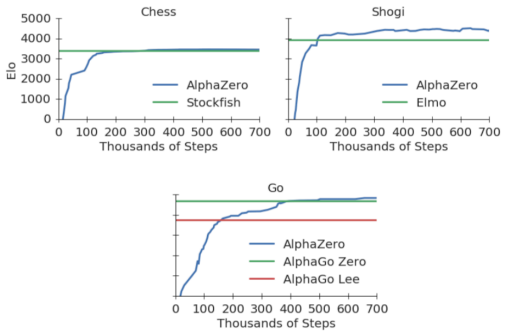
In this sense, the two networks are linked. The value network predicts the value for the current policy. It's unclear from the paper whether in this version of AlphaGo the policy network also uses the value network in training, or only plays full games.

In a later version, called AlphaZero, the researcher hit on the bright idea of making the lower layers of the two networks *shared*. The idea here is that the lower layers to neural networks tend to extract generic features that are largely task-independent. By using the same layers for both tasks, these parts of the network get a stronger training signal.



After 21 days of self-play (on a large computing cluster), AlphaZero surpassed the performance of the version that beat Lee Sedol.

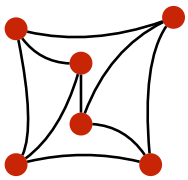
Alpha Zero (2017)

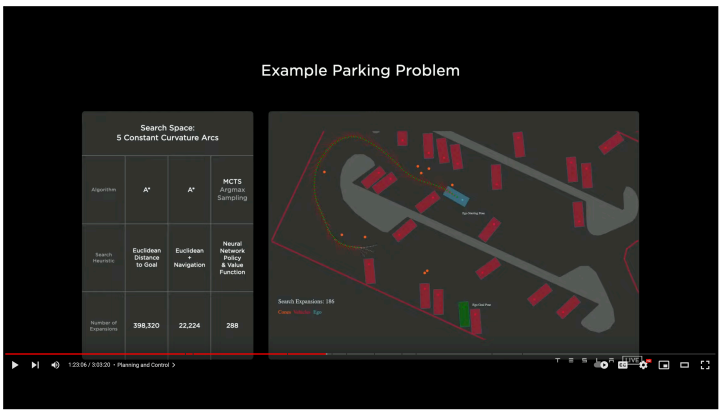


MCTS on non-games

(Path) planning, theorem proving, query answering, etc.

- Game tree search -> graph search
- Rollouts -> random walks
- Minimax -> Breadth/depth first search
- MCTS -> MCTS / Beam search

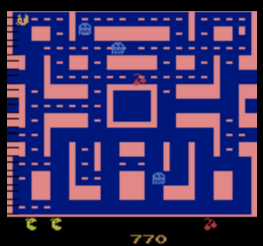




https://www.youtube.com/watch?v=j0z4FweCy4M

---

what if we don't know the rules?



just learn a "dynamics function"

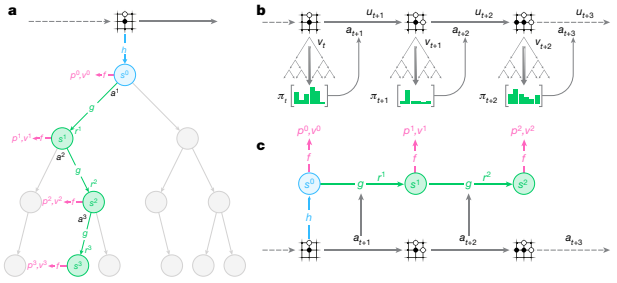
$state_1$, reward <- g($state_0$, action)

41

NB: You need a simple representation for the action space.

---

for example: MuZero

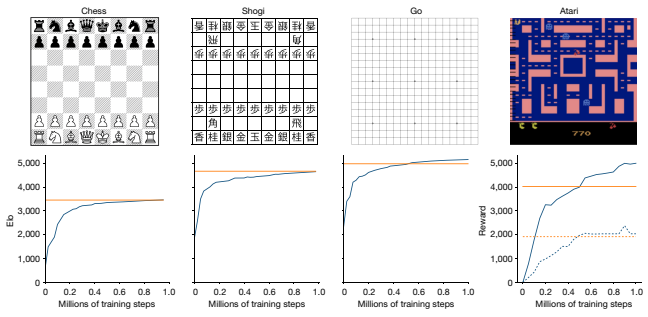


42

a) MCTS with a trained network.

b) MCTS collecting a replay buffer b acting in the environment with MCTS

c)



43

---

any questions?