# Applications of Model-Based Deep Reinforcement Learning

Johanni Brea

18 April 2023

Artificial Neural Networks CS-456

EPFL

## Table of Contents

# Model-Based Deep Reinforcement Learning

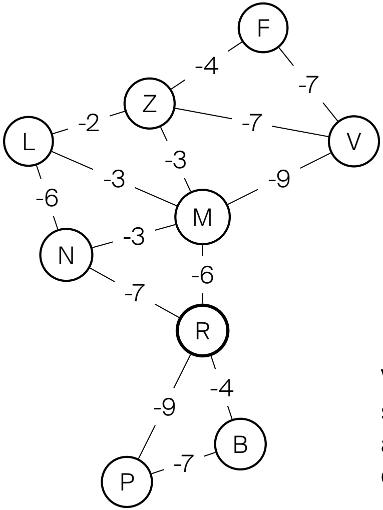
A model-based reinforcement learning method estimates the transition dynamics (e.g. $P^a_{s\to s'}$) and reward structure (e.g. $R^a_{s\to s'}$).

**Today you will learn**

1. how the model can be used for **planning**,
2. how we can use standard planning methods in deep RL with variational state tabulation (**VaST**).
3. why we may prefer **decision time planning** over **background planning**,
4. how **AlphaZero** learns to play games with **Monte Carlo Tree Search**,
5. how **MuZero** reaches the same level as AlphaZero without explicitly knowing the rules of the game.

Suggested: Sutton & Barto Ch. 8,
[Corneil et al., 2018, Silver et al., 2016, Silver et al., 2018, Schrittwieser et al., 2019]

# Background Planning

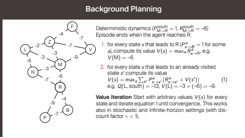


Deterministic dynamics ($P^{\text{south}}_{\text{M}\to\text{R}} = 1$, $R^{\text{south}}_{\text{M}\to\text{R}} = -6$)
Episode ends when the agent reaches R.

1. for every state $s$ that leads to R ($P^a_{s\to R} = 1$ for some $a$), compute its value $V(s) = \max_a R^a_{s\to \text{R}}$, e.g. $V(\text{M}) = -6$

2. for every state $s$ that leads to an already visited state $s'$ compute its value
$$V(s) = \max_a \sum_{s'} P^a_{s\to s'}\left(R^a_{s\to s'} + V(s')\right) \tag{1}$$
e.g. $Q(\text{L}, \text{south}) = -13$, $V(\text{L}) = -3 + (-6) = -9$.

**Value Iteration** Start with arbitrary values $V(s)$ for every state and iterate equation 1 until convergence. This works also in stochastic and infinite-horizon settings (with discount factor $\gamma < 1$).



Background Planning

Deterministic dynamics ($P^{\text{south}}_{\text{M}\to\text{R}} = 1$, $R^{\text{south}}_{\text{M}\to\text{R}} = -6$)
Episode ends when the agent reaches R.

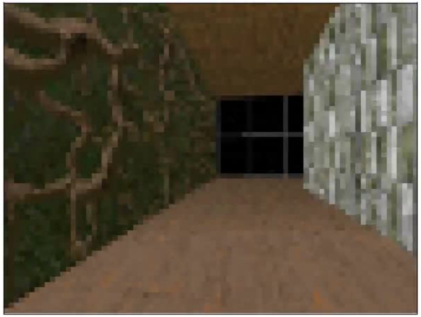
1. for every state $s$ that leads to R ($P^a_{s\to R} = 1$ for some $a$), compute its value $V(s) = \max_a R^a_{s\to\text{R}}$, e.g. $V(\text{M}) = -6$
2. for every state $s$ that leads to an already visited state $s'$ compute its value
$$V(s) = \max_a \sum_{s'} P^a_{s\to s'}(R^a_{s\to s'} + V(s')) \quad (1)$$
e.g. $Q(\text{L}, \text{south}) = -13$, $V(\text{L}) = -3 + (-6) = -9$.

**Value Iteration** Start with arbitrary values $V(s)$ for every state and iterate equation 1 until convergence. This works also in stochastic and infinite-horizon settings (with discount factor $\gamma < 1$).

## Notes

- Planning can be done when the model is known. We find a typical example in car navigation systems.
- On the left you see a schematic map of Europe. The nodes represent cities (Frankfurt, Zürich, Vienna, Lausanne, Milano, Nice, Rome, Bari, Palermo) and the numbers on the edges indicate the cost (e.g. some function of fuel and time costs) of traveling from one city to another one.
- The transition dynamic is deterministic: if one chooses to go south in Milano one arrives deterministically in Rome $P^{\text{south}}_{\text{M}\to\text{R}} = 1$ with cost $R^{\text{south}}_{\text{M}\to\text{R}} = -6$.
- Our goal is to find the shortest path to Rome, starting from any city. To do so, we compute the value $V$ of every city, where the value indicates the costs of the optimal path that starts in that city and ends in Rome.
- The value of neighbouring cities of Rome is the cost of taking the action that leads directly to Rome.
- For all other cities we can compute a preliminary value by considering cities for which we already know the value, e.g. the Q-value of going to Nice in Lausanne is -13, while the Q-value of going to Milano is -9, and the Q-value of going to Zurich first is not yet known. After having the preliminary values for Lausanne, Zurich and Vienna, we can recompute the values for all neighbours and find eventually that the preliminary values are identical to the final values. This example should have given you the intuition of value iteration.
- In value iteration one starts with arbitrary values for all states and iterates Eq. 1 for all states until convergence. There are some refinement of this basic planning strategy, like hierarchical planning (first city to city, then road to road), neglecting irrelevant parts of the environment (the value of Frankfurt is irrelevant, if you start in Nice) or prioritizing updates (if the preliminary value of Zurich changed by a large amount, the preliminary values of the neighbouring cities are also likely to change).

## Model-Based Deep RL with Variational State Tabulation (VaST)

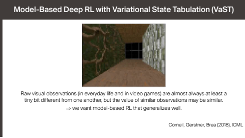


Raw visual observations (in everyday life and in video games) are almost always at least a tiny bit different from one another, but the value of similar observations may be similar.

$\Rightarrow$ we want model-based RL that generalizes well.

Corneil, Gerstner, Brea (2018), ICML

## Notes



Model-Based Deep RL with Variational State Tabulation (VaST)

Raw visual observations (in everyday life and in video games) are almost always at least a tiny bit different from one another, but the value of similar observations may be similar.

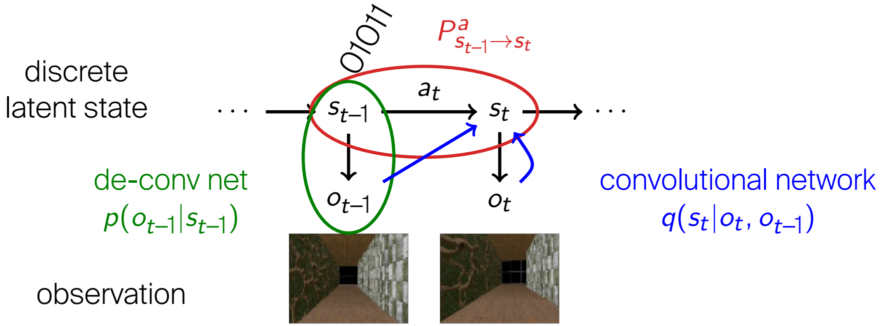
$\Rightarrow$ we want model-based RL that generalizes well.

Corneil, Gerstner, Brea (2018), ICML

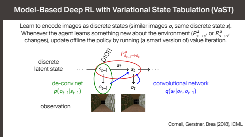
- What is a discrete state in the environment of animals, humans or in video games? Every raw observation looks at least a tiny bit different from all other observations, but the value of states that look similar (e.g. observations that are only one footstep separated) are likely to be similar.
- Naively assigning each raw observation to a new, discrete state and learning the value for each state is unreasonable: there are many states, most states are visited only once or a few times and in standard tabular reinforcement learning there is no generalization across states.
- Idea: learn a mapping to abstract discrete states such that neighbouring observations are mapped to the same discrete state and define a similarity metric in the space of abstract discrete states such that similar observations are mapped to similar discrete states.
- With discrete abstract states we can use again standard planning methods.

# Model-Based Deep RL with Variational State Tabulation (VaST)

Learn to encode images as discrete states (similar images $o$, same discrete state $s$). Whenever the agent learns something new about the environment ($P^a_{s\to s'}$ or $R^a_{s\to s'}$ changes), update offline the policy by running (a smart version of) value iteration.



Corneil, Gerstner, Brea (2018), ICML


Model-Based Deep RL with Variational State Tabulation (VaST)

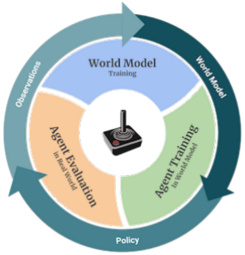
Learn to encode images as discrete states (similar images $o$, same discrete state $s$). Whenever the agent learns something new about the environment ($P^a_{s\to s'}$ or $R^a_{s\to s'}$ changes), update offline the policy by running (a smart version of) value iteration.

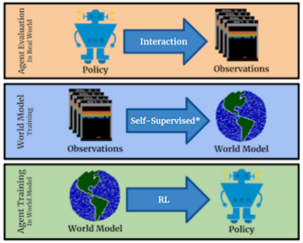
Corneil, Gerstner, Brea (2018), ICML

## Notes

- The abstract states (discrete latent states) are binary words of length $K$. The similarity metric is the Hamming distance, i.e. the number of bits that are different between any pair of binary words.
- The binary words can be represented as integers. These integers are the states that can be used to estimate the model, i.e. the transition probabilities and the reward table. The model is used to find a good policy by running a smart version of value iteration that prioritizes updating the values of states, whose next states had recently an update of the value.
- The function that maps observations to binary words (encoder) is a convolutional network with $K$ binary output neurons. To deal with moving objects we use the same trick as in DQN for Atari games and give as input multiple subsequent observations.
- To make sure that the discrete latent state contains relevant information about the state we use a deconvolutional network that takes latent states as input and produces raw observations. The only purpose of this decoder is to enforce the encoder to learn a latent representation that discriminates different states; otherwise it may happen that the encoder maps every observation to a single latent state $s_0$ and $P^a_{s_0\to s_0} = 1$, $R^a_{s_0\to s_0}$ =average reward, but no useful policy can be found.

# WorldModels, SimPLe, Dreamer





Other approaches to model-based deep RL exist.

Most of them

interact with the environment to collect data,

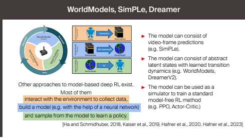
build a model (e.g. with the help of a neural network)

and sample from the model to learn a policy.

- The model can consist of video-frame predictions (e.g. SimPLe).
- The model can consist of abstract latent states with learned transition dynamics (e.g. WorldModels, DreamerV2).
- The model can be used as a simulator to train a standard model-free RL method (e.g. PPO, Actor-Critic.)

[Ha and Schmidhuber, 2018, Kaiser et al., 2019, Hafner et al., 2020, Hafner et al., 2023]

# Notes



WorldModels, SimPLe, Dreamer

Other approaches to model-based deep RL exist. Most of them interact with the environment to collect data, build a model (e.g. with the help of a neural network) and sample from the model to learn a policy.

- The model can consist of video-frame predictions (e.g. SimPLe).
- The model can consist of abstract latent states with learned transition dynamics (e.g. WorldModels, DreamerV2).
- The model can be used as a simulator to train a standard model-free RL method (e.g. PPO, Actor-Critic.)

[Ha and Schmidhuber, 2018, Kaiser et al., 2019, Hafner et al., 2020, Hafner et al., 2023]

- Alternatively to classic planning methods, the world model can be used to simulate experiences and train a model-free reinforcement learning policy on the samples from the world model.
- In video-frame prediction a neural network learns to approximate directly $p(o_{t+1}|a_t, o_t, o_{t-1}, ...)$ (SimPLe = simulated policy learning).
- WorldModels and DreamerV2 learn an abstract representation with an autoencoder approach similar to VaST, but instead of using planning methods in the latent state, they sample (dreamed) trajectories to train a policy.
- It may sound silly to replace a simulator (like the Atari video game environment) by an approximate simulator that is learned with a neural network and then use this approximate simulator to train a policy instead of training a policy directly on the original simulator. The goal is, however, to show that training with the learned simulator allows to reduce the sample complexity, i.e. the number of interactions with the actual environment. This can be useful for environments that are costly to interact. For example, imagine a robot that can cook. If it has a somewhat accurate model of the things that can happen in a kitchen, this robot can simulate many cooking episodes until it has found a policy that leads to a great meal, without having to waste any resources during the episodes that are used to train the policy.
- Currently, model-based deep reinforcement learning is still a very active field of research, where the goal is to show that using a model leads to policies that are as good as those obtained with model-free reinforcement learning while requiring far less interactions with the environment than the model-free methods.

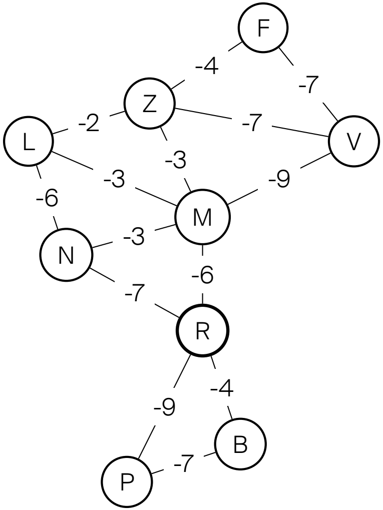
## Summary: Background Planning and Variational State Tabulation

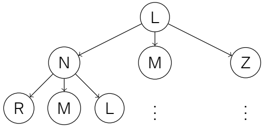
- In model-based reinforcement learning we can use background planning to update the Q- and V-values efficiently (e.g. by value iteration), if the number of states and actions is not too large.
- Alternatively, the model can be used to sample experiences offline and train a model-free agent with these artificial samples.
- Learned abstractions may be useful to map high dimensional states (like images) to discrete states that can be used with standard planning methods.

## Value Iteration Does Not Scale!

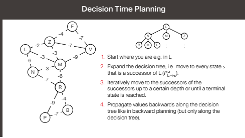
- There are presumably less than $10^8$ intersections of roads in Europe. Value iteration on current compute hardware is feasible for this problem.
- But there are more than $10^{170}$ possible positions for the game Go. (The number of atoms on earth is roughly $10^{50}$).

# Decision Time Planning





1. Start where you are e.g. in L
2. Expand the decision tree, i.e. move to every state $s$ that is a successor of L ($P^a_{\mathrm{L}\to s}$).
3. Iteratively move to the successors of the successors up to a certain depth or until a terminal state is reached.
4. Propagate values backwards along the decision tree like in backward planning (but only along the decision tree).


Decision Time Planning

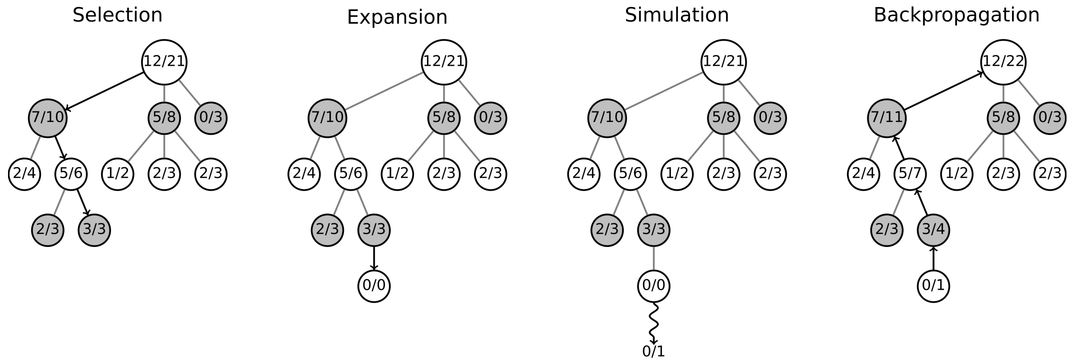
1. Start where you are e.g. in L
2. Expand the decision tree, i.e. move to every state $s$ that is a successor of L ($P^a_{L\to s}$).
3. Iteratively move to the successors of the successors up to a certain depth or until a terminal state is reached.
4. Propagate values backwards along the decision tree like in backward planning (but only along the decision tree).

## Notes

- When full value iteration is not feasible, one can start from the current state and expand the decision tree.
- For example, if we want to find the shortest path from Lausanne to Rome, we can start in Lausanne, expand the decision tree by moving to every successor state (neighbouring city) and from each successor state iteratively continue expanding the decision tree up to a certain depth or until a terminal state is reached. Then we propagate values backwards along the decision tree. In this example we see that the optimal solution can be found at depth 2 of the decision tree. Thus, the update equation of value iteration (Eq. 1 on slide 3) needs to be applied only three times (for Nice, Milano and Lausanne; the branch following Zurich has not yet reached any terminal node).
- In general, it is not a good idea to ignore branches that have not yet reached a terminal node (if there were a teleporter in Frankfurt that takes zero cost to travel to Rome, the optimal path would only be discoverable at depth 3 of the decision tree and thus ignoring the node Zurich at depth 2 would not be a good idea), but it may be too costly to expand the decision tree to large depth. One strategy to deal with this is Monte Carlo Tree Search.

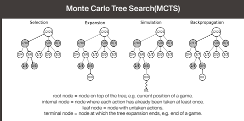
## Table of Contents

# Monte Carlo Tree Search(MCTS)



root node = node on top of the tree, e.g. current position of a game.
internal node = node where each action has already been taken at least once.
leaf node = node with untaken actions.
terminal node = node at which the tree expansion ends, e.g. end of a game.

## Notes


Monte Carlo Tree Search(MCTS)

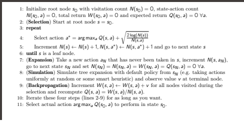
root node = node on top of the tree, e.g. current position of a game.
internal node = node where each action has already been taken at least once.
leaf node = node with untaken actions.
terminal node = node at which the tree expansion ends, e.g. end of a game.

- In MCTS the tree gets expanded in an iterative fashion.
- In the figure you see a tree starting at a certain position in a two player game. It is known that white won 12 out of 21 simulated games from the current position. If white chooses the first action the game reaches a position (where it is the black player's turn), where white won 7 out of 10 simulated games, etc.
- In MCTS, one of the leaf nodes is selected for expansion (in this case it is a position where it is black's turn and white won 3 out of 3 simulated games). One of black's allowed actions is taken and a full game is simulated with a random policy (wiggly line). The random game from the expanded position led to a loss (0/1). This value is back-propagated. At the end of this iteration white one 12 out of 22 simulated games from the root position.
- Thanks to the simulation step that always ends at a terminal node, every leaf node has an estimate of the actual value. However, this estimate can be terrible, because the simulation is run with a random policy, which may fail to reflect that some positions lead to a win with certainty, if the actions are selected carefully.

# Monte Carlo Tree Search

```
1:  Initialize root node $s_0$ with visitation count $N(s_0) = 0$, state-action count
    $N(s_0, a) = 0$, total return $W(s_0, a) = 0$ and expected return $Q(s_0, a) = 0 \ \forall a$.
2:  (**Selection**) Start at root node $s = s_0$.
3:  **repeat**
4:      Select action $a^* = \arg\max_a Q(s, a) + \sqrt{\frac{2\log(N(s))}{N(s,a)}}$
5:      Increment $N(s) \leftarrow N(s) + 1$, $N(s, a^*) \leftarrow N(s, a^*) + 1$ and go to next state $s$
6:  **until** $s$ is a leaf node.
7:  (**Expansion**) Take a new action $a_N$ that has never been taken in $s$, increment $N(s, a_N)$,
    go to next state $s_N$ and set $N(s_N) = N(s_N, a) = W(s_N, a) = Q(s_N, a) = 0 \ \forall a$.
8:  (**Simulation**) Simulate tree expansion with default policy from $s_N$ (e.g. taking actions
    uniformly at random or some smart heuristic) and observe value $v$ at terminal node.
9:  (**Backpropagation**) Increment $W(s, a) \leftarrow W(s, a) + v$ for all nodes visited during the
    selection and recompute $Q(s, a) = W(s, a)/N(s, a)$.
10: Iterate these four steps (lines 2-9) for as long as you want.
11: Select actual action $\arg\max_a Q(s_0, a)$ to perform in state $s_0$.
```

## Notes



```
1: Initialize root node $s_0$ with visitation count $N(s_0) = 0$, state-action count
   $N(s_0, a) = 0$, total return $W(s_0, a) = 0$ and expected return $Q(s_0, a) = 0 \ \forall a$.
2: (**Selection**) Start at root node $s = s_0$.
3: **repeat**
4:     Select action $a^* = \arg\max_a Q(s, a) + \sqrt{\frac{2\log(N(s))}{N(s,a)}}$
5:     Increment $N(s) \leftarrow N(s) + 1, N(s, a^*) \leftarrow N(s, a^*) + 1$ and go to next state $s$
6: **until** $s$ is a leaf node.
7: (**Expansion**) Take a new action $a_N$ that has never been taken in $s$, increment $N(s, a_N)$,
   go to next state $s_N$ and set $N(s_N) = N(s_N, a) = W(s_N, a) = Q(s_N, a) = 0 \ \forall a$.
8: (**Simulation**) Simulate tree expansion with default policy from $s_N$ (e.g. taking actions
   uniformly at random or some smart heuristic) and observe value $v$ at terminal node.
9: (**Backpropagation**) Increment $W(s, a) \leftarrow W(s, a) + v$ for all nodes visited during the
   selection and recompute $Q(s, a) = W(s, a)/N(s, a)$.
10: Iterate these four steps (lines 2-9) for as long as you want.
11: Select actual action $\arg\max_a Q(s_0, a)$ to perform in state $s_0$.
```

1: If one has already visitation counts etc. from previous runs of MCTS, one can use them to initialize the root node with other values than 0.

2: Every iteration starts at the root node, e.g. the current game position.

4: Here we show a variant of the UCB1 exploration-exploitation criterion. $Q(s, a)$ drives exploitation, the second term drives exploration. The exploration term should be such that it decreases with $N(s, a)$ (actions with low $N(s, a)$ should be explored) and increases slowly with $N(s)$ (if $s$ is visited often, we want to be really sure that none of the less taken actions would in fact be optimal; an increase in $N(s)$ drives occasional re-exploration). Other explorations policies are possible, e.g. other functions of $N(s)$ and $N(s, a)$, $\epsilon$-greedy exploration, Thompson sampling.

7: If a node has actions that have never been taken before, one of this actions is taken and the landing state $s_N$ is initialized.

8: The random game is played to have a very crude approximation of the value of state $s_N$.

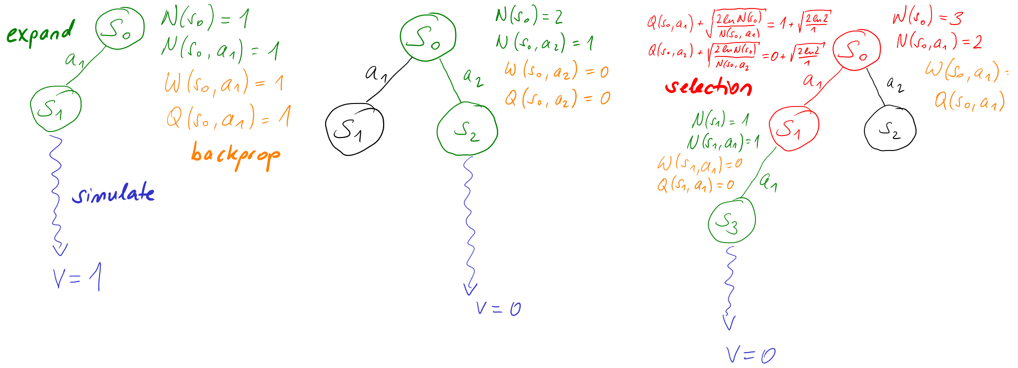
9: Despite its name, backpropagation in MCTS is completely different from the error backpropagation algorithm for deep neural networks! In MCTS, backpropagation means that the Q-values along the path that was taken during selection and expansion should be updated with the value that was obtained at the terminal node.

11: After many iterations the tree has grown quite a bit and we can select at the root node the action that has the highest Q-value. We do not need to perform exploration in the actual environment, because the model is perfectly known and MCTS would converge to the correct $Q$-value in the limit of $N(s_0) \to \infty$.

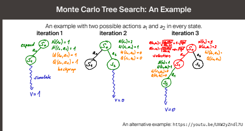
# Monte Carlo Tree Search: An Example

An example with two possible actions $a_1$ and $a_2$ in every state.

**iteration 1** **iteration 2** **iteration 3**



An alternative example: `https://youtu.be/UXW2yZndl7U`


Monte Carlo Tree Search: An Example

An example with two possible actions $a_1$ and $a_2$ in every state.

An alternative example: `https://youtu.be/UXW2yZndl7U`

## Notes

**iteration 1** The root node is a leaf node, because it has actions that have never been explored. Therefore we **expand** with one of the untaken actions ($a_1$) and increment the counts. Next we **simulate** a random game and observe a win. This is **backpropagated** to keep track of the fact that taking action $a_1$ in $s_0$ led to a win.

**iteration 2** The root node is still a leaf node, because action $a_2$ has not yet been explored. We **expand** the tree with action $a_2$, **simulate** a random game and observe a loss. This is **backprpagated** to keep track of the fact that taking action $a_2$ in $s_0$ led to a loss.

**iteration 3** Now the root node is not a leaf node anymore and we need to select. Because both actions have been taken once and therefore have the same exploration bonus, the Q-value determines the action and $a_1$ is **selected**. State $s_1$ is a leaf node, therefore it gets **expanded** etc.

**iteration 4** (not shown on the slide). At the root node we need now to compare $\frac{1}{2} + \sqrt{\frac{2\log(3)}{2}}$ (for action $a_1$) to $0 + \sqrt{\frac{2\log(3)}{1}}$ (for action $a_2$) to decide which action to take. Despite the larger exploration bonus for action $a_2$, the total value for action $a_1$ is larger and therefore $a_1$ would be selected. Next, $s_1$ would be expanded with action $a_2$ and thus become an internal node.

## Quiz

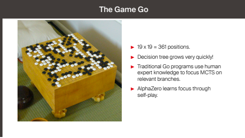
Which statement is correct?

- ☐ With background planning, the computation at decision time is very cheap, since one can use the stored $Q$-values to find the optimal action.
- ☐ In Monte Carlo Tree Search, the computation at decision time is very cheap, since one can use the stored $Q$-values to find the optimal action.
- ☐ Monte Carlo Tree Search plays games with a random policy to estimate the value of a position.
- ☐ The model in model-based deep RL can be used to “dream up” sequences that were never actually experienced.
- ☐ Typically, model-based deep RL methods are computationally heavier but more sample efficient than model-free deep RL methods.

## Table of Contents

# The Game Go

- 19 x 19 = 361 positions.
- Decision tree grows very quickly!
- Traditional Go programs use human expert knowledge to focus MCTS on relevant branches.
- AlphaZero learns focus through self-play.



The Game Go

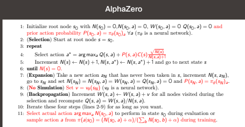
- 19 x 19 = 361 positions.
- Decision tree grows very quickly!
- Traditional Go programs use human expert knowledge to focus MCTS on relevant branches.
- AlphaZero learns focus through self-play.

## Notes

- Go is usually played on a board with 19 x 19 = 361 positions.
- For the first few actions there are are thus more than 300 legal positions to place the stone; even in later stages of the game, there are many legal actions. Expanding the decision tree in MCTS is therefore very costly ($300^3 = 27$ million).
- Traditional Go computer programs (like CrazyStone) use MCTS with heuristics to exclude moves that human experts would never play.
- AlphaZero does not use any expert knowledge, but learns to focus on relevant parts of the decision tree thanks to some neural networks that are trained with self-play.

# AlphaZero

```
1: Initialize root node $s_0$ with $N(s_0) = 0, N(s_0, a) = 0,\ W(s_0, a) = 0\ Q(s_0, a) = 0$ and
   prior action probability $P(s_0, a) = \pi_\theta(s_0)_a\ \forall a$ ($\pi_\theta$ is a neural network).
2: (**Selection**) Start at root node $s = s_0$.
3: **repeat**
4:     Select action $a^* = \arg\max_a Q(s, a) + P(s, a)C(s)\frac{\sqrt{N(s)}}{N(s,a)+1}$
5:     Increment $N(s) \leftarrow N(s) + 1, N(s, a^*) \leftarrow N(s, a^*) + 1$ and go to next state $s$
6: **until** $N(s) = 0$.
7: (**Expansion**) Take a new action $a_N$ that has never been taken in $s$, increment $N(s, a_N)$,
   go to $s_N$ and set $N(s_N) = N(s_N, a) = W(s_N, a) = Q(s_N, a) = 0$ and $P(s_N, a) = \pi_\theta(s_N)_a$.
8: (**No Simulation**) Set $v = v_\theta(s_N)$ ($v_\theta$ is a neural network).
9: (**Backpropagation**) Increment $W(s, a) \leftarrow W(s, a) + v$ for all nodes visited during the
   selection and recompute $Q(s, a) = W(s, a)/N(s, a)$.
10: Iterate these four steps (lines 2-9) for as long as you want.
11: Select actual action $\arg\max_a N(s_0, a)$ to perform in state $s_0$ during evaluation or
    sample action $a$ from $\pi(a|s_0) = (N(s_0, a) + \alpha)/(\sum_b N(s_0, b) + \alpha)$ during training.
```



AlphaZero

```
1: Initialize root node $s_0$ with $N(s_0) = 0, N(s_0, a) = 0$, $W(s_0, a) = 0$ $Q(s_0, a) = 0$ and
   prior action probability $P(s_0, a) = \pi_\theta(s_0)_a$ $\forall a$ ($\pi_\theta$ is a neural network).
2: (**Selection**) Start at root node $s = s_0$.
3: **repeat**
4:     Select action $a^* = \arg\max_a Q(s, a) + P(s, a) C(s) \frac{\sqrt{N(s)}}{N(s,a)+1}$
5:     Increment $N(s) \leftarrow N(s) + 1, N(s, a^*) \leftarrow N(s, a^*) + 1$ and go to next state $s$
6: **until** $N(s) = 0$.
7: (**Expansion**) Take a new action $a_N$ that has never been taken in $s$, increment $N(s, a_N)$,
   go to $s_N$ and set $N(s_N) = N(s_N, a) = W(s_N, a) = Q(s_N, a) = 0$ and $P(s_N, a) = \pi_\theta(s_N)_a$.
8: (**No Simulation**) Set $v = v_\theta(s_N)$ ($v_\theta$ is a neural network).
9: (**Backpropagation**) Increment $W(s, a) \leftarrow W(s, a) + v$ for all nodes visited during the
   selection and recompute $Q(s, a) = W(s, a)/N(s, a)$.
10: Iterate these four steps (lines 2-9) for as long as you want.
11: Select actual action $\arg\max_a N(s_0, a)$ to perform in state $s_0$ during evaluation or
    sample action $a$ from $\pi(a|s_0) = (N(s_0, a) + \alpha)/(\sum_b N(s_0, b) + \alpha)$ during training.
```

## Notes

- AlphaZero modifies MCTS in two important ways: a neural network to compute prior action probabilities helps in the selection process to focus on relevant parts of the decision tree and another neural network helps to estimate the value at leaf nodes, making simulations of random games unnecessary.

1: The prior action probabilities for root node $s_0$ are computed with a neural network.

4: In the selection process, these prior action probabilities influence exploration. For example, "silly actions" that lead almost certainly to a loss will not be explored with $P(s, a) \approx 0$ (usually one does not need to consider how a chess game continues after sacrificing the queen for no reason), whereas promising actions with large $P(s, a)$ are heavily explored. Side note: the exploration rate $C(s) = c_1 + \log((1 + N(s) + c_2)/c_2)$ grows slowly with search time, but is essentially constant if $N(s)$ is small (in MuZero $c_1 = 1.25$, $c_2 = 19652$; I didn't find the parameter values for AlphaZero). I do not know why the factor $C(s)\sqrt{N(s)}/(N(s, a) + 1)$ is chosen instead of the UCB1 exploration term (it may have better convergence bounds). But this term satisfies the criteria we discussed on line 4 in slide 12.

7: In the expansion step the prior action probabilities are computed using the neural network $\pi_\theta$.

8: Instead of simulating a random game, a value network is used to estimate the value (win probability) of a given position.

11: Greedy actions are selected during evaluation. The argmax of the counts is apparently less sensitive to outliers than the argmax of the Q-values. During training the actions are sampled from $\pi(a|s_0) = (N(s_0, a) + \alpha)/(\sum_b N(s_0, b) + \alpha)$ where $\alpha = \{0.3, 0.15, 0.03\}$ for chess, shogi and go, respectively. Thanks to non-zero $\alpha$ random actions are occasionally taken.

## AlphaZero: How to train the networks

Loss function

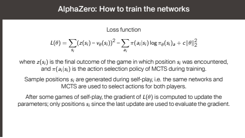
$$L(\theta) = \sum_{s_i} (z(s_i) - v_\theta(s_i))^2 - \sum_{a_i} \pi(a_i|s_i) \log \pi_\theta(s_i)_a + c\|\theta\|_2^2$$

where $z(s_i)$ is the final outcome of the game in which position $s_i$ was encountered, and $\pi(a_i|s_i)$ is the action selection policy of MCTS during training.

Sample positions $s_i$ are generated during self-play, i.e. the same networks and MCTS are used to select actions for both players.

After some games of self-play, the gradient of $L(\theta)$ is computed to update the parameters; only positions $s_i$ since the last update are used to evaluate the gradient.

# Notes


AlphaZero: How to train the networks

Loss function

$$L(\theta) = \sum_{s_i} (z(s_i) - v_\theta(s_i))^2 - \sum_{a_i} \pi(a_i|s_i) \log \pi_\theta(s_i)_a + c\|\theta\|_2^2$$

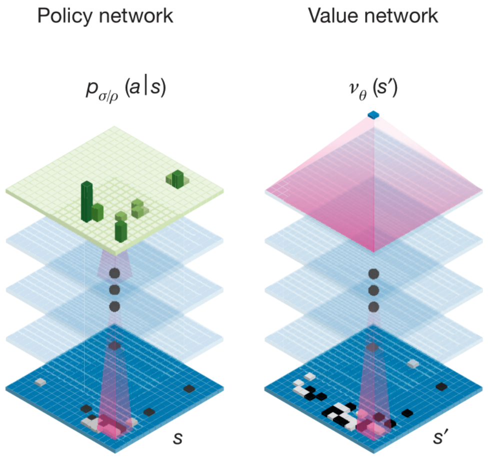
where $z(s_i)$ is the final outcome of the game in which position $s_i$ was encountered, and $\pi(a_i|s_i)$ is the action selection policy of MCTS during training.

Sample positions $s_i$ are generated during self-play, i.e. the same networks and MCTS are used to select actions for both players.

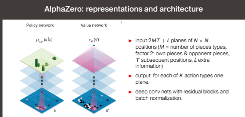
After some games of self-play, the gradient of $L(\theta)$ is computed to update the parameters; only positions $s_i$ since the last update are used to evaluate the gradient.

- With the squared error loss the value $v_\theta(s_i)$ learns to approximate the win probability from a given position.
- The prior action probabilities are trained with the cross-entropy loss to approximate the policy with which MCTS selects actions. Like this, exploration will be biased towards actions that were in the past selected by MCTS. This may look like a vicious loop, where MCTS is biased towards actions deemed useful by $\pi_\theta$ and $\pi_\theta$ learns to imitate the MCTS policy. I guess the sampling step in line 11 of AlphaZero during training, assures that $\pi_\theta$ does not get stuck at a sub-optimal policy where the probability of some actions would be zero and does never explored.
- The prior action probabilities $\pi_\theta$ resemble the policy of a model-free RL agent. However, it is not trained by a standard model-free RL method, but instead learns to imitate the policy of MCTS. We are thus in a funny situation, where we use model-free reinforcement learning to drive exploration in a setting where the full model is known, but the state space is so large that standard model-based approaches, like value iteration, fail to solve the problem.
- Why does MCTS still make sense, once we have $\pi_\theta$? There are so many possible states that $\pi_\theta$ remains a crude approximation of the MCTS policy, even after extensive training. A useful analogy may be that $\pi_\theta$ is the policy that a professional player uses when forced to make a move without time to think (e.g. in blitz-chess). Given time to think, however, the player may use this policy to explore promising actions in the mind, evaluate resulting positions with $v_\theta$ and take a more informed decision than without thinking.

# AlphaZero: representations and architecture



- input $2MT + L$ planes of $N \times N$ positions ($M$ = number of pieces types, factor 2: own pieces & opponent pieces, $T$ subsequent positions, $L$ extra information)
- output: for each of $K$ action types one plane.
- deep conv nets with residual blocks and batch normalization.


AlphaZero: representations and architecture

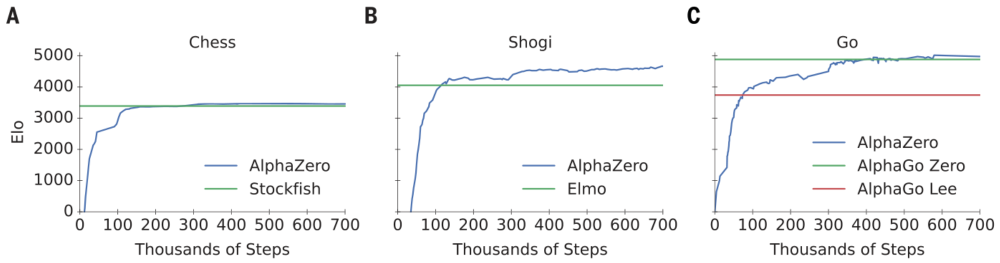
- input $2MT + L$ planes of $N \times N$ positions ($M$ = number of pieces types, factor 2: own pieces & opponent pieces, $T$ subsequent positions, $L$ extra information)
- output: for each of $K$ action types one plane.
- deep conv nets with residual blocks and batch normalization.

## Notes

- For games with $N \times N$ positions and $M$ different types of pieces the input consists of $2MT + L$ planes of size $N \times N$.
- $M$ planes indicate with 0 and 1 the positions of own pieces, $M$ planes the positions of opponent pieces. The previous $T$ positions are stacked to $2MT$ planes.
- $L$ additional planes (constant values across the $N \times N$ positions) are used to represent the color of the player, the move number, the availability of special rules (e.g. castling in chess).
- Outputs of the policy networks are also represented as planes. For go there is only one plane that indicates where to place the piece. For chess there are 73 planes: one plane for each type of move (e.g. move 1 square north, move 3 squares south, move 5 square south-west etc.). A large value at a certain position in one of these planes, indicates that the piece at that position should be taken to perform the action indicated by the plane identity. Illegal moves are masked to have 0 probability.
- $\pi_\theta$ and $v_\theta$ use the same network body: a conv layer with batch-normalization followed by 19 residual blocks. Policy and value heads add some additional layers on top.

# AlphaZero: Success Story

Games: Silver et al. 2018

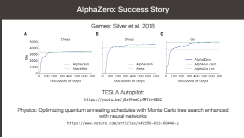


TESLA Autopilot:

`https://youtu.be/j0z4FweCy4M?t=4802`

Physics: Optimizing quantum annealing schedules with Monte Carlo tree search enhanced with neural networks

`https://www.nature.com/articles/s42256-022-00446-y`


AlphaZero: Success Story

Games: Silver et al. 2018

TESLA Autopilot:

https://youtu.be/j0z4FweCy4M?t=4802

Physics: Optimizing quantum annealing schedules with Monte Carlo tree search enhanced with neural networks

https://www.nature.com/articles/s42256-022-00446-y

## Notes

- Except of input and output representations, the same architecture and approach can be applied to reach better better performance than any computer program or human player in chess, shogi and go.
- Similar approaches can be applied to search problems that need to be solved efficiently. In a video from last year, Tesla has an example, where an AlphaZero approach was applied in a car's autopilot, to plan the trajectory from current position to a parking lot while avoiding many obstacles.
- In research we see the same approach applied to search problems in physics.

## AlphaZero: Summary

- In each position of the game (for both players), MCTS runs for some time.
- The sub-tree below the actual next state is retained as the initial MCTS tree.
- In contrast to traditional Go/Chess/Shogi software there is no hand-crafting of promising tree expansions or position evaluations: selection of MCTS is focused on promising actions (according to learned prior action probability $\pi_\theta$) and the value of leaf nodes $v_\theta$ is based on a learned position evaluation function.
- Prior action probabilities $\pi_\theta$ are trained with the cross-entropy loss to match the actual policy $\pi$ of MCTS and the values $v_\theta$ are trained with the squared error loss to the actual outcome $z$ of the game.
- AlphaZero does not need to learn the reward and transition model $R^a_{s\to s'}, P^a_{s\to s'}$; for MCTS it relies on the hard coded rules of the game. What if the model is unknown?

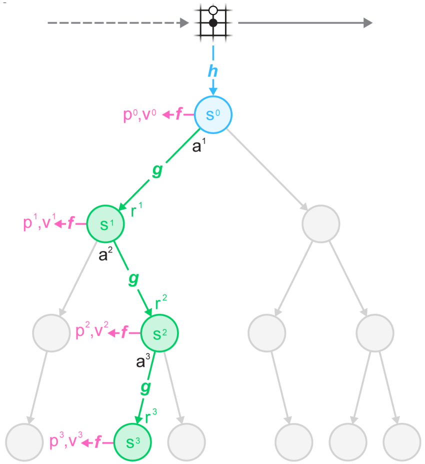
## Table of Contents

## What If the Model Is Unknown?

In high dimensional settings, like in the game Go, it seems hopeless to learn explicitly $R^a_{s \to s'}$ and $P^a_{s \to s'}$.

MuZero = AlphaZero with a learned latent representation of observations and learned dynamics model.

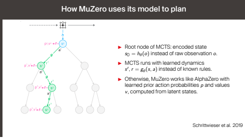
# How MuZero uses its model to plan



- Root node of MCTS: encoded state $s_0 = h_\theta(o)$ instead of raw observation $o$.
- MCTS runs with learned dynamics $s', r = g_\theta(s, a)$ instead of known rules.
- Otherwise, MuZero works like AlphaZero with learned prior action probabilities $p$ and values $v$, computed from latent states.

Schrittwieser et al. 2019

# Notes



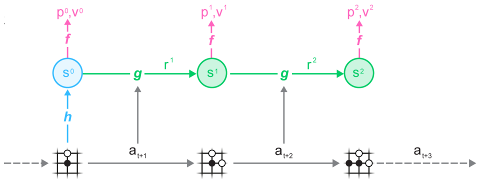
How MuZero uses its model to plan

- Root node of MCTS: encoded state $s_0 = h_\theta(o)$ instead of raw observation $o$.
- MCTS runs with learned dynamics $s', r = g_\theta(s, a)$ instead of known rules.
- Otherwise, MuZero works like AlphaZero with learned prior action probabilities $p$ and values $v$, computed from latent states.

Schrittwieser et al. 2019

- The model consists of three connected components for representation, dynamics and prediction. The initial hidden state $s^{(0)} \in \mathbb{R}^N$ is obtained by passing current (and potentially a stack of past) observations (e.g. the Go board or Atari screen) into a representation function $h_\theta$.
- Given a previous hidden state $s^{(k-1)}$ and a candidate action $a^{(k)}$, the dynamics function $g_\theta$ produces an immediate reward $r^{(k)}$ and a new hidden state $s^{(k)}$. The prediction of the immediate reward $r$ is necessary for environments with intermediate rewards, like the Atari games.
- The policy $p^{(k)}$ and value function $v^{(k)}$ are computed from the hidden state $s^{(k)}$ by a prediction function $f_\theta = (\pi_\theta, v_\theta)$, as in AlphaZero.
- In contrast to VaST, DreamerV2 etc. the latent states $s$ do not need to contain all information to reproduce the raw observations. “There is no direct constraint or requirement for the hidden state to capture all information necessary to reconstruct the original observation, drastically reducing the amount of information the model has to maintain and predict; nor is there any requirement for the hidden state to match the unknown, true state of the environment; nor any other constraints on the semantics of state. Instead, the hidden states are free to represent state in whatever way is relevant to predicting current and future values and policies. Intuitively, the agent can invent, internally, the rules or dynamics that lead to most accurate planning.” This is an advantage when the raw observation contains information that is irrelevant for the task.

# How MuZero trains its model



Model

$$\left.\begin{array}{ll} s^0 & = h_\theta(o_1, ..., o_t) \\ r^k, s^k & = g_\theta(s^{k-1}, a^k) \\ \mathbf{p}^k, v^k & = f_\theta(s^k) \end{array}\right\} \ \mathbf{p}^k, v^k, r^k = \mu_\theta(o_1, ..., o_t, a^1, ..., a^k)$$

Search

$$\nu_t, \pi_t = MCTS(s_t^0, \mu_\theta)$$
$$a_t \sim \pi_t$$

Learning Rule

$$\mathbf{p}_t^k, v_t^k, r_t^k = \mu_\theta(o_1, ..., o_t, a_{t+1}, ..., a_{t+k})$$

$$z_t = \begin{cases} u_T & \text{for games} \\ u_{t+1} + \gamma u_{t+2} + ... + \gamma^{n-1} u_{t+n} + \gamma^n \nu_{t+n} & \text{for general MDPs} \end{cases}$$

$$l_t(\theta) = \sum_{k=0}^{K} l^r(u_{t+k}, r_t^k) + l^v(z_{t+k}, v_t^k) + l^p(\pi_{t+k}, p_t^k) + c||\theta||^2$$

Losses

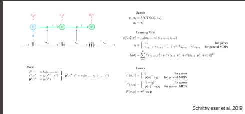
$$l^r(u, r) = \begin{cases} 0 & \text{for games} \\ \phi(u)^T \log \mathbf{r} & \text{for general MDPs} \end{cases}$$

$$l^v(z, q) = \begin{cases} (z - q)^2 & \text{for games} \\ \phi(z)^T \log \mathbf{q} & \text{for general MDPs} \end{cases}$$

$$l^p(\pi, p) = \boldsymbol{\pi}^T \log \mathbf{p}$$

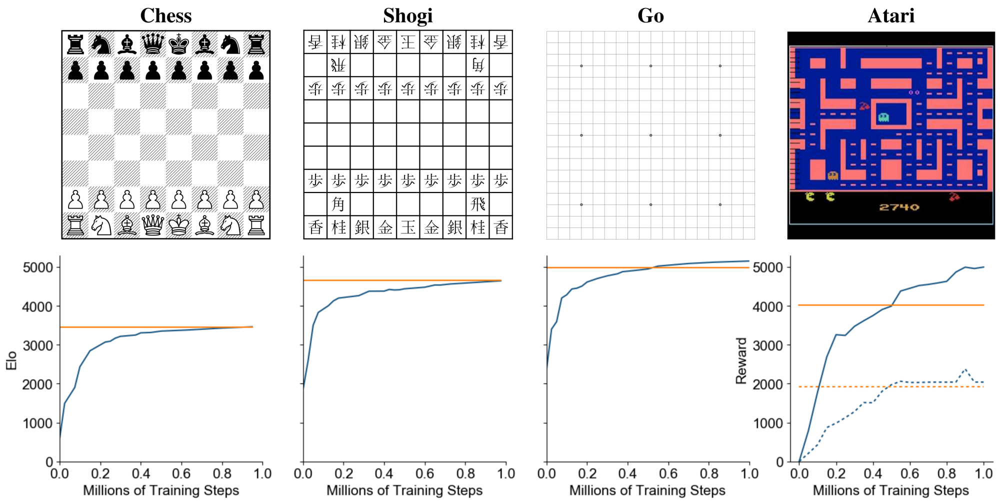
Schrittwieser et al. 2019

## Notes


Schrittwieser et al. 2019

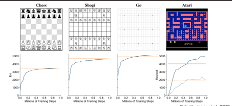
- A trajectory is sampled from the replay buffer. For the initial step, the representation function $h$ receives as input the past observations $o_1, \ldots, o_t$ from the selected trajectory.
- The model is subsequently unrolled recurrently for $K$ steps. At each step $k$, the dynamics function $g$ receives as input the hidden state $s^{(k-1)}$ from the previous step and the real action $a_{t+k}$.
- The parameters of the representation, dynamics and prediction functions are jointly trained, end-to-end by backpropagation-through-time, to predict three quantities: the prior action selection policy $p^{(k)} \approx \pi_{t+k}$ should approximate the MCTS policy $\pi_{t+k}$ in time-step $t+k$, the value $v^{(k)} \approx z_{t+k}$ should approximate the return $z_{t+k}$ ($u_T$ indicates win or loss in a board game and $u_t$ the intermediate rewards in general), and the immediate reward $r^{(t+k)} \approx u_{t+k}$ should approximate the actual immediate reward $u_{t+k}$.

# MuZero: Success Story



Schrittwieser et al. 2019

## Notes



Schrittwieser et al. 2019

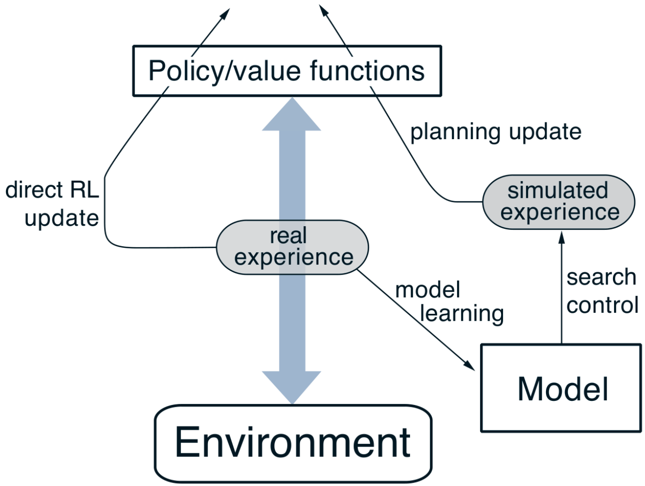
Benchmark lines for Chess, Shogi & Go: AlphaZero.

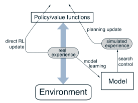
For Atari: R2D2, a model-free agent that was state-of the art in 2019 (solid lines: mean, dashed lines: median)

## MuZero: Summary

- Instead of relying on a given reward and transition model, MuZero learns an encoding of observations to states $h_\theta$ and a transition function from states to next states $g_\theta$.
- Like AlphaZero, it relies on MCTS with a learned function $f_\theta$ that estimates the action probability $\pi_\theta$ and value $v_\theta$ of a state.
- In contrast to VaST, WoldModels and SimPLe, the latent state of MuZero is not trained in an autoencoder approach, i.e. $o_t$ is not necessarily reconstructable from $s_t$.

## Dyna Architecture





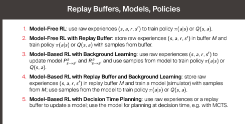
## Notes

Figure 8.1 from Sutton and Barto: The general Dyna Architecture. Real experience, passing back and forth between the environment and the policy, affects policy and value functions in much the same way as does simulated experience generated by the model of the environment.

## Replay Buffers, Models, Policies

1. **Model-Free RL**: use raw experiences $(s, a, r, s')$ to train policy $\pi(a|s)$ or $Q(s, a)$.
2. **Model-Free RL with Replay Buffer**: store raw experiences $(s, a, r, s')$ in buffer $M$ and train policy $\pi(a|s)$ or $Q(s, a)$ with samples from buffer.
3. **Model-Based RL with Background Learning**: use raw experiences $(s, a, r, s')$ to update model $P^a_{s\to s'}$ and $R^a_{s\to s'}$ and use samples from model to train policy $\pi(a|s)$ or $Q(s, a)$.
4. **Model-Based RL with Replay Buffer and Background Learning**: store raw experiences $(s, a, r, s')$ in replay buffer $M$ and train a model (simulator) with samples from $M$; use samples from the model to train policy $\pi(a|s)$ or $Q(s, a)$.
5. **Model-Based RL with Decision Time Planning**: use raw experiences or a replay buffer to update a model; use the model for planning at decision time, e.g. with MCTS.

# Notes



Replay Buffers, Models, Policies

1. **Model-Free RL**: use raw experiences $(s, a, r, s')$ to train policy $\pi(a|s)$ or $Q(s, a)$.
2. **Model-Free RL with Replay Buffer**: store raw experiences $(s, a, r, s')$ in buffer $M$ and train policy $\pi(a|s)$ or $Q(s, a)$ with samples from buffer.
3. **Model-Based RL with Background Learning**: use raw experiences $(s, a, r, s')$ to update model $P^a_{s\to s'}$ and $R^a_{s\to s'}$ and use samples from model to train policy $\pi(a|s)$ or $Q(s, a)$.
4. **Model-Based RL with Replay Buffer and Background Learning**: store raw experiences $(s, a, r, s')$ in replay buffer $M$ and train a model (simulator) with samples from $M$; use samples from the model to train policy $\pi(a|s)$ or $Q(s, a)$.
5. **Model-Based RL with Decision Time Planning**: use raw experiences or a replay buffer to update a model; use the model for planning at decision time, e.g. with MCTS.

1. e.g. standard policy gradient or Q-learning
2. e.g. DQN or ACER (policy gradient with replay buffer).
3. e.g. Dyna-Q (see Sutton and Barto Chapter 8.2)
4. e.g. VaST, Dreamer, etc.
5. e.g. MuZero

# Quiz

Which statement is correct?

- ☐ In AlphaZero's MCTS the actual actions are sampled from the prior action probability $P(s, a)$.
- ☐ In AlphaZero's MCTS the prior action probability is trained to be as close as possible to the actual policy of MCTS.

# References I

Corneil, D., Gerstner, W., and Brea, J. (2018).
Efficient model-based deep reinforcement learning with variational state tabulation.
In Dy, J. and Krause, A., editors, Proceedings of the 35th International Conference on Machine Learning, volume 80 of Proceedings of Machine Learning Research, pages 1057–1066, Stockholmsmässan, Stockholm Sweden. PMLR.

Ha, D. and Schmidhuber, J. (2018).
World Models.
ArXiv e-prints.

Hafner, D., Lillicrap, T., Norouzi, M., and Ba, J. (2020).
Mastering Atari with Discrete World Models.
arXiv e-prints, page arXiv:2010.02193.

Hafner, D., Pasukonis, J., Ba, J., and Lillicrap, T. (2023).
Mastering Diverse Domains through World Models.
arXiv e-prints, page arXiv:2301.04104.

Kaiser, L., Babaeizadeh, M., Milos, P., Osinski, B., Campbell, R. H., Czechowski, K., Erhan, D., Finn, C., Kozakowski, P., Levine, S., Mohiuddin, A., Sepassi, R., Tucker, G., and Michalewski, H. (2019).
Model-Based Reinforcement Learning for Atari.
arXiv e-prints, page arXiv:1903.00374.

# References II

Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., Lockhart, E., Hassabis, D., Graepel, T., Lillicrap, T., and Silver, D. (2019).
Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model.
arXiv e-prints, page arXiv:1911.08265.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., and et al. (2016).
Mastering the game of go with deep neural networks and tree search.
Nature, 529(7587):484–489.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., and et al. (2018).
A general reinforcement learning algorithm that masters chess, shogi, and go through self-play.
Science, 362(6419):1140–1144.