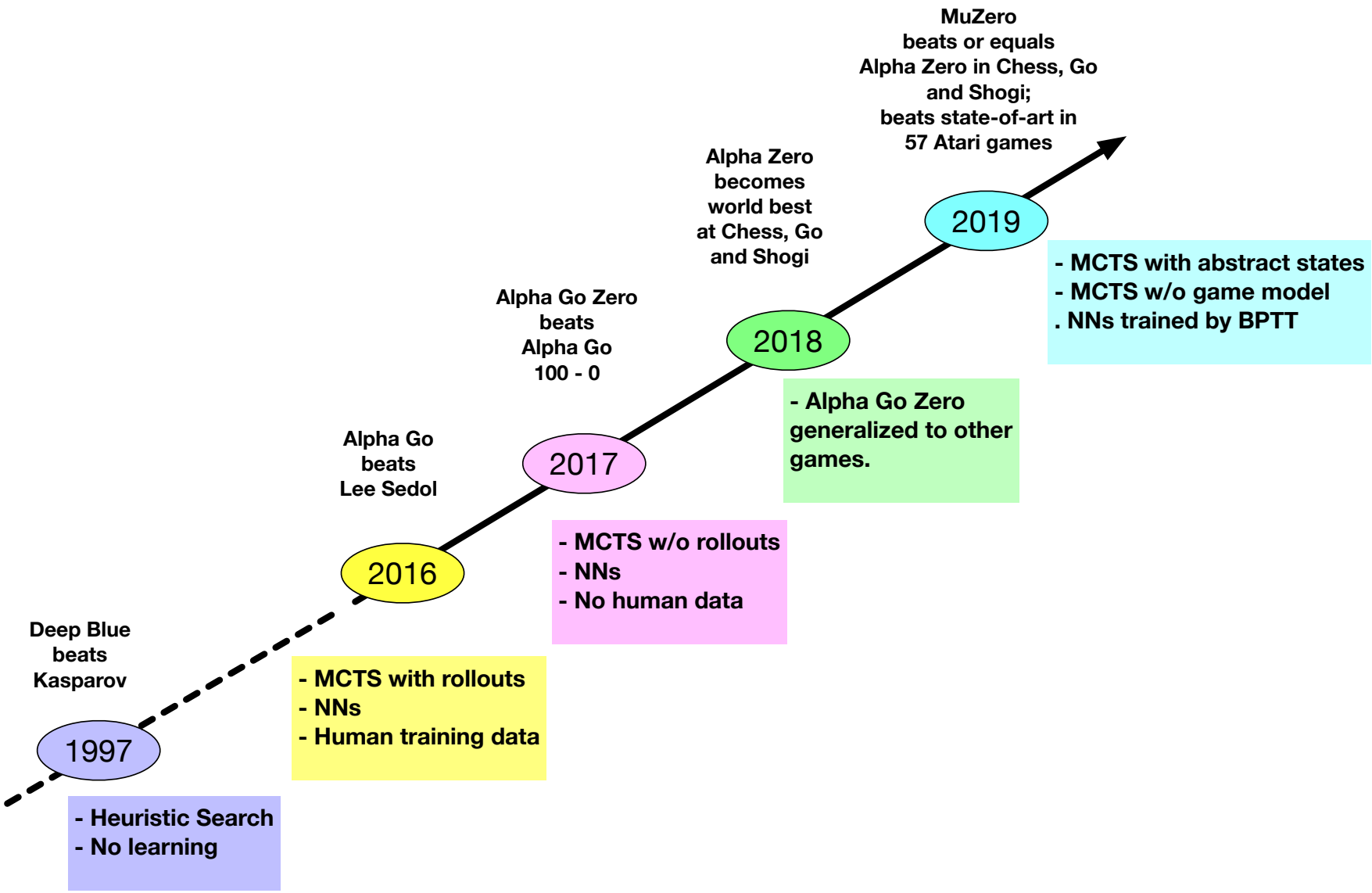
MuZero

Keith L. Downing

December 6, 2024

* This presentation assumes familiarity with Reinforcement Learning (RL), Monte Carlo Tree Search (MCTS) and JAX, as presented in other lectures for this course.

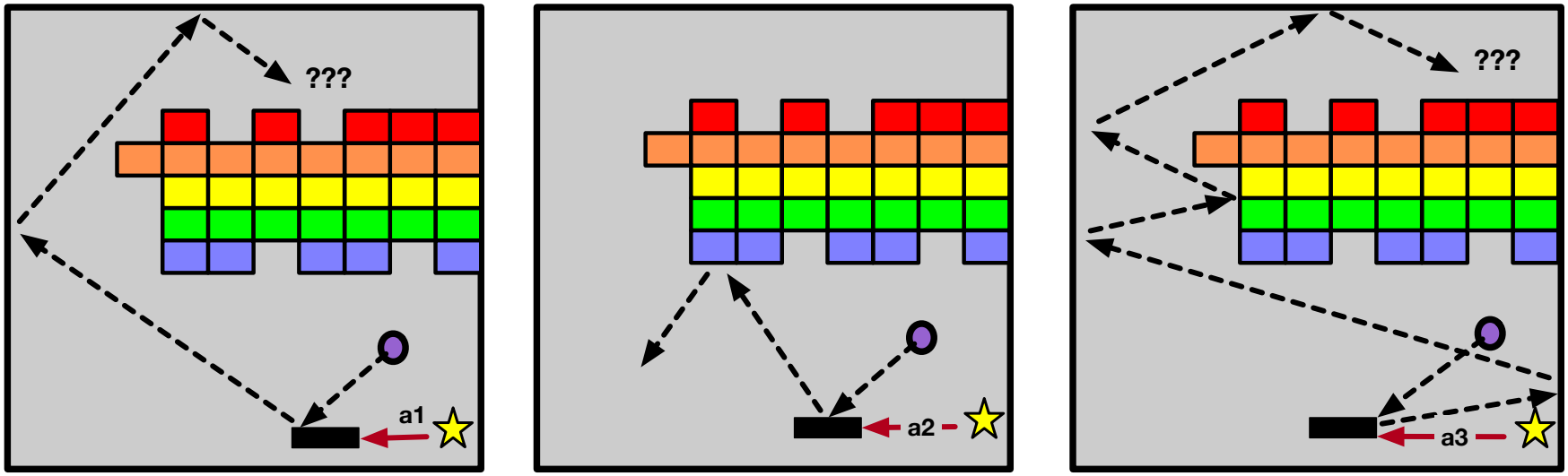
DeepMind Dominance



Model-Based -vs- Model-Free RL

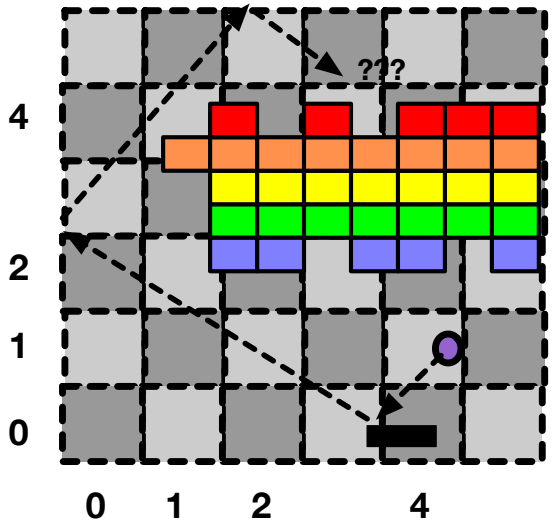
- In a Markov Decision Process (MDP), **model**(M) := mapping: (state, action) $\rightarrow$ (state, reward).
- In AI, a **model** may also include details such as all legal moves, terminal-state criteria, and any other rules of the game or constraints of the domain.
- **Policy** := mapping: state $\rightarrow$ action.
- **Model-based RL** systems either have M at the outset or try to learn M via trial-and-error domain experience (i.e. *episodes* of problem-solving / game-play).
- **Model-free RL** uses domain experience to adjust its policy but does not rely on a model, and does not try to learn one. It just uses trial-and-error to learn what actions do / don't achieve good results in different situations / states: it learns a useful policy.
- M is well-known for board games such as Chess, Checkers, Go, etc., since the rules of the game are non-ambiguous, and for any action (a) taken in game state (s) (1): the legality of the action is easily determined, and (2) the new state (s') and reward (r) that result from applying a to s are easily determined.
- Models for arcade games (e.g. Atari) are **much** more elusive: although the legality of actions is not so difficult to determine, their consequences can be very complex, often requiring a good simulator, and occasionally involving stochasticity. E.g. results of one *paddle hit* in Breakout.

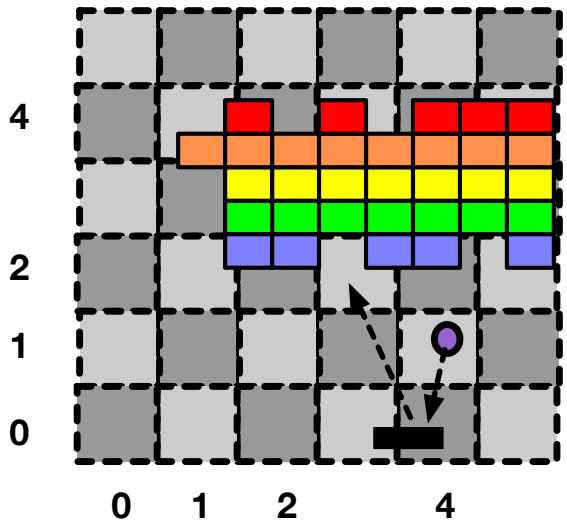
Complexity of Arcade-Game Pixel Images

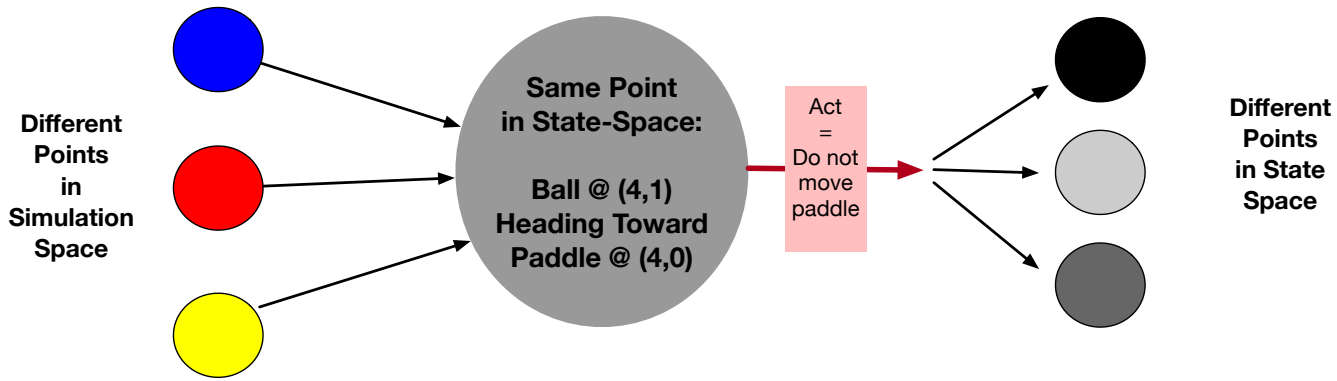


- Resolution of a Go Board: 19 x 19.
- Pixel Resolution of Atari 2600 games: 192 x 160.

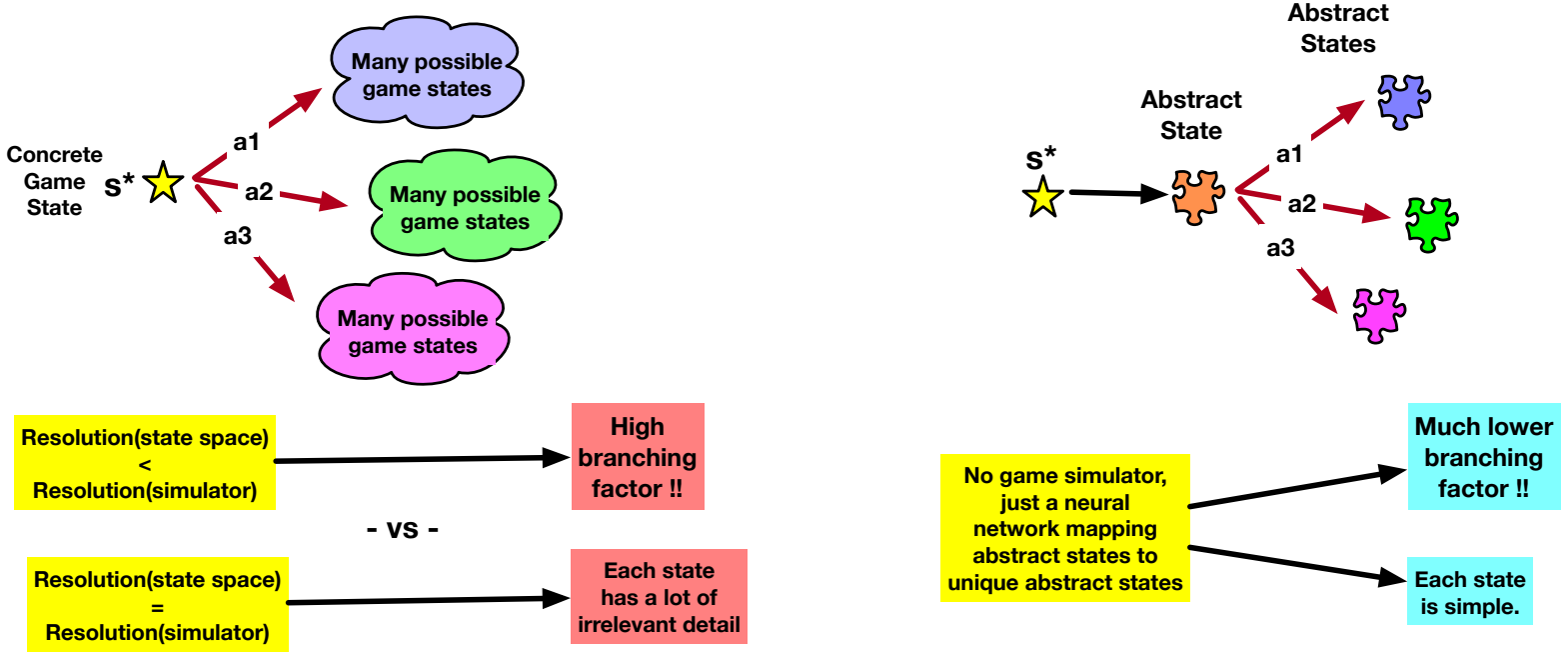
Resolution of Simulations -vs- RL Game States







Abstract States



- Abstract states are **not** alternate representations of real-game states.
- They are functional patterns (identical to the activation patterns in NN hidden layers) that enable predictions of policies, values and rewards in the real-game situation.
- But they do **not** support the prediction of future real-game states.

Action Time -vs- Reaction Time

- Human Reaction Time $= 200 - 300$ msec := Time to process sensory information and then perform an action based on it.
- Human Action Time $= 60 - 80$ msec := Time to perform a simple action, such as a finger tap.

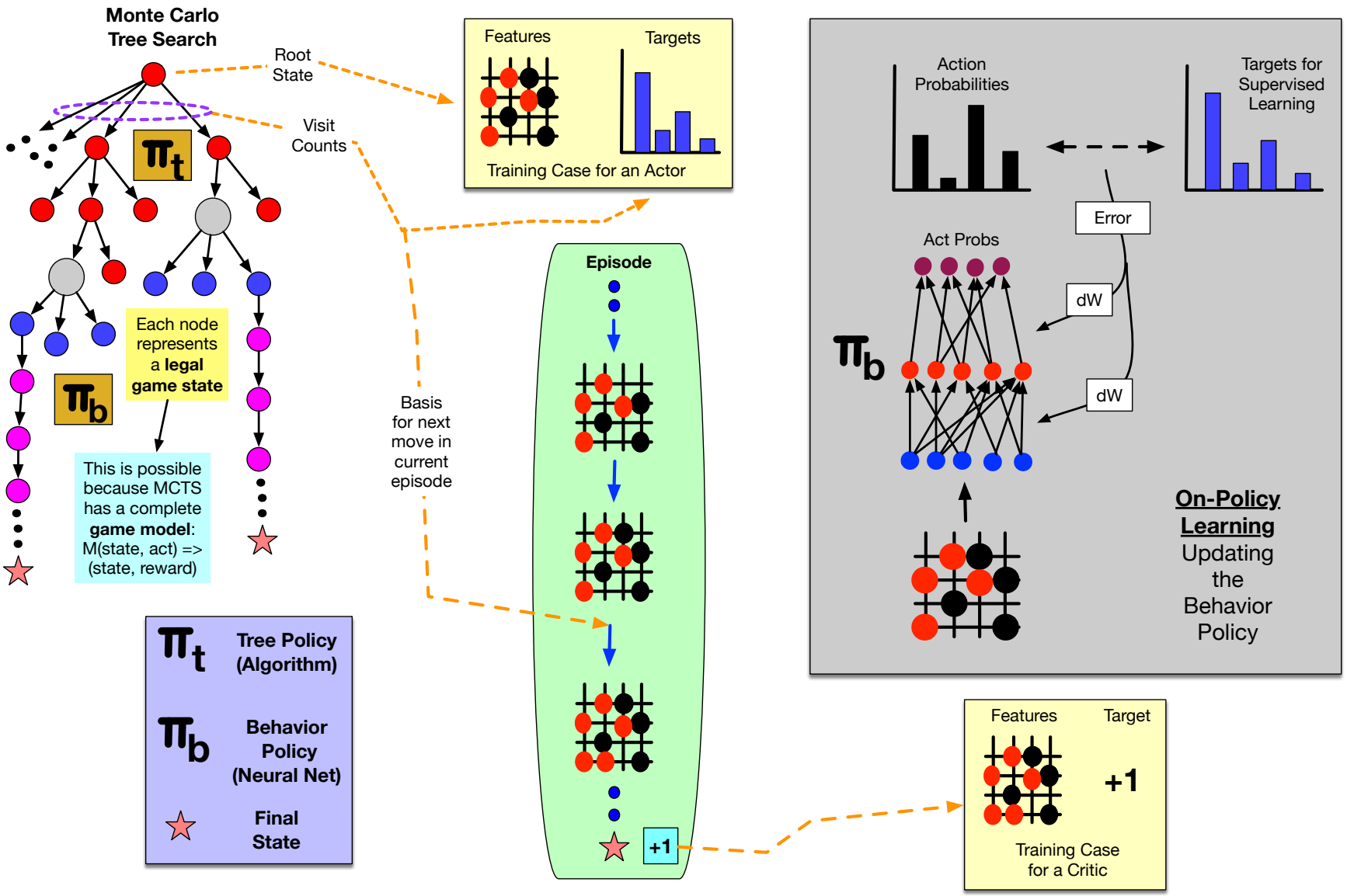
$\rightarrow$ Humans can perform 3-5 actions per round of sensory processing.

$\rightarrow$ In a video game, they can do 3-5 actions per viewed image.

$\rightarrow$ An AI system to play video games might take several actions per input image.

- This is obviously not the case for turn-taking board games (e.g. chess, checkers, poker, go, etc.)
- MuZero allows each action to transform an **abstract** state to a new one, but for each **real** game state it permits a short sequence of actions without requiring explicit transitions to other real game states.

Basic Monte Carlo Tree Search (MCTS)



Reinforcement Learning with MCTS

Play many episodes of the *game*. $\forall$ action decisions of each episode, do:

MCTS Algorithm

- Use episode's current game state (s_k) as basis for tree root (R).
- ** Use Tree Policy from root to a leaf node
- Expand the leaf node: generate all children.
- Randomly pick one child.
- Do a rollout from that child to a final state (s*) using the Behavior Policy.
- Backpropagate value(s*) along the vine from s* $\rightsquigarrow$ R, updating Q values and visit counts on all edges.
- Go to **

After building the tree via many simulations (i.e. loops in above algorithm):

- Use visit counts as basis for a probability distribution (D_k).
- Sample from D_k to get next move in the current episode.
- Save pair (s_k,D_k) as training case for the **actor** neural network.
- At episode end (state s*), use value (v*) of the final state to create training cases for **critic** neural network: $(s, v*)$ $\forall s \in s_0 \rightsquigarrow s*$ - discounts to v* may apply.

Model-Based MCTS

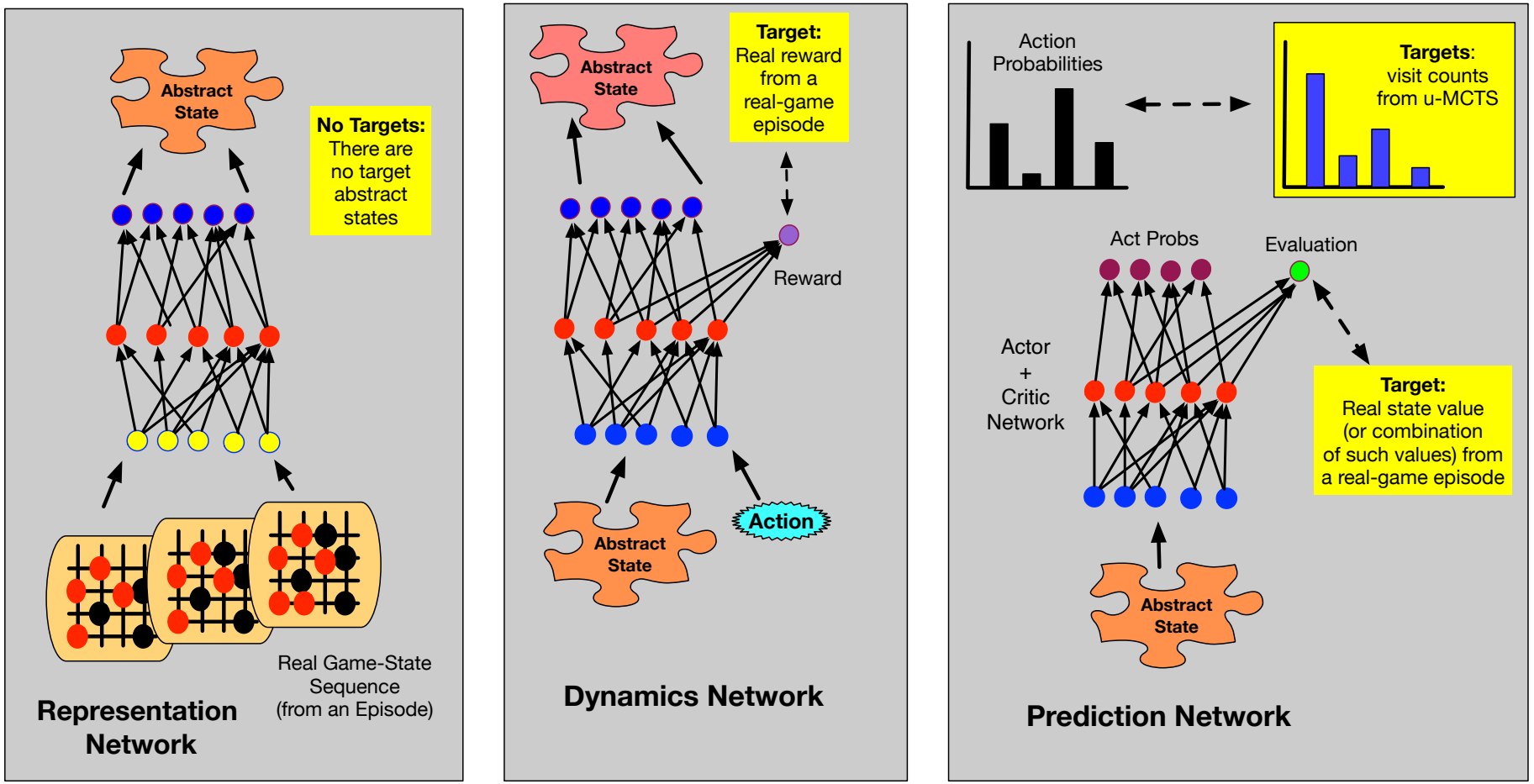
Classic MCTS

- Classic MCTS is *model-based* and uses a complete model (M) known from the start.
- M is used to compute all legal actions for a state, and all consequent states and rewards once the action is applied.
- Hence, the MCTS tree is built from real game states and legal successor (child) states based on legal actions.
- This is impossible without the model.

MuZero MCTS

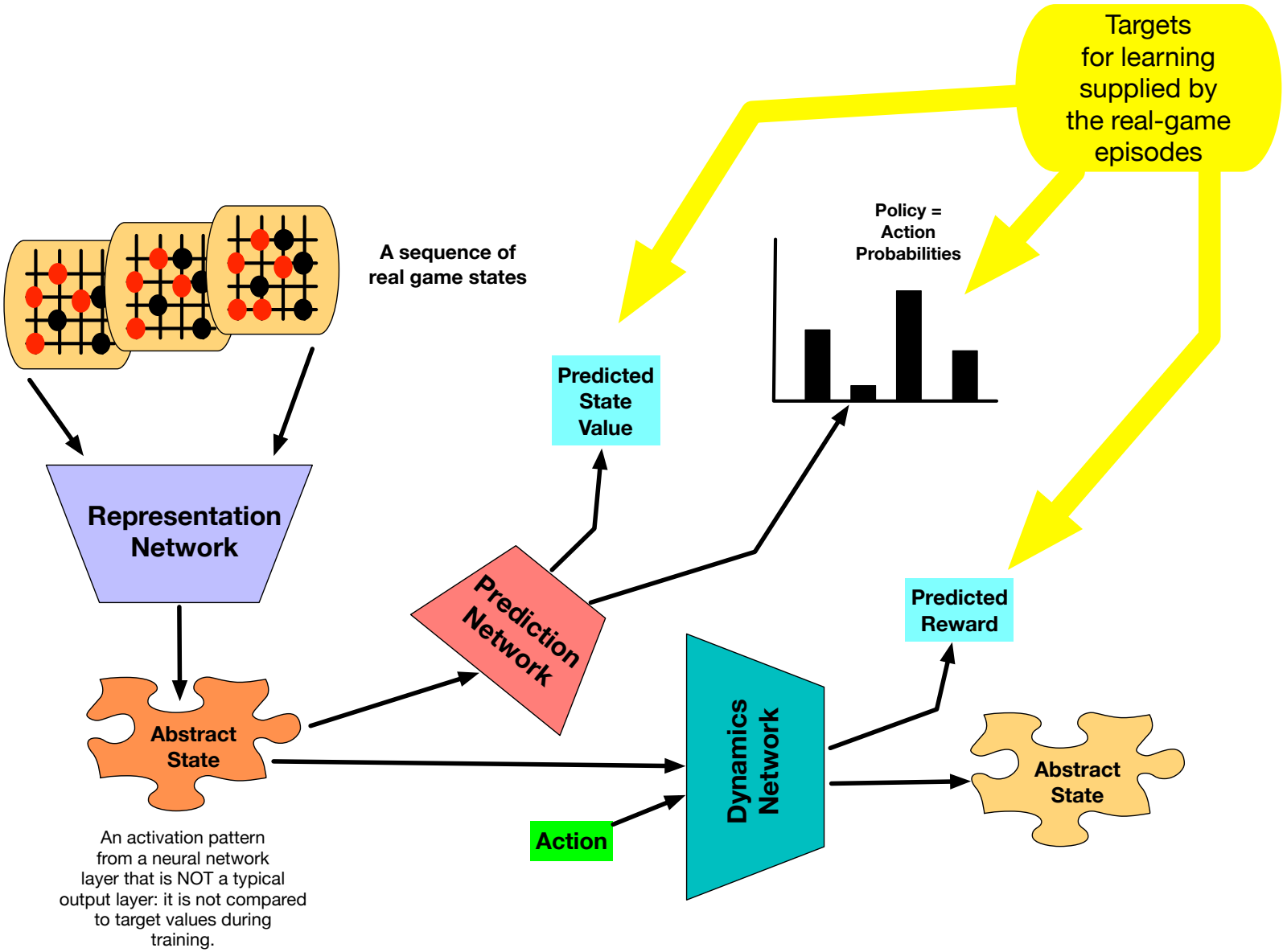
- MuZero may have a model (or a good simulator), but it is **not** available to u-MCTS.
- MuZero uses M to compute next states and rewards in the **real** episodes.
- But u-MCTS has no model for transitions in the **simulated** episodes (from a leaf node on down).
- In fact, u-MCTS cannot even recognize a game-ending state.
- It relies on a dynamics model M_d that tries to learn a mapping between abstract states which have no direct correlation to states in the real-game episodes.
- But by learning M_d (and 2 other mappings) and using training data from the real-game episodes, it indirectly learns a useful policy for real-game states.

Three Core Neural Networks in MuZero



- Representation Network = mapping from real-game to abstract states.
- Dynamics Network = A supporting model, but not an actual game model.
- Prediction Network = A policy (actor) and state evaluator (critic).

Connecting the Networks



Reinforcement Learning with u-MCTS

Play many episodes of the *game*. $\forall$ action decisions of each episode, do:

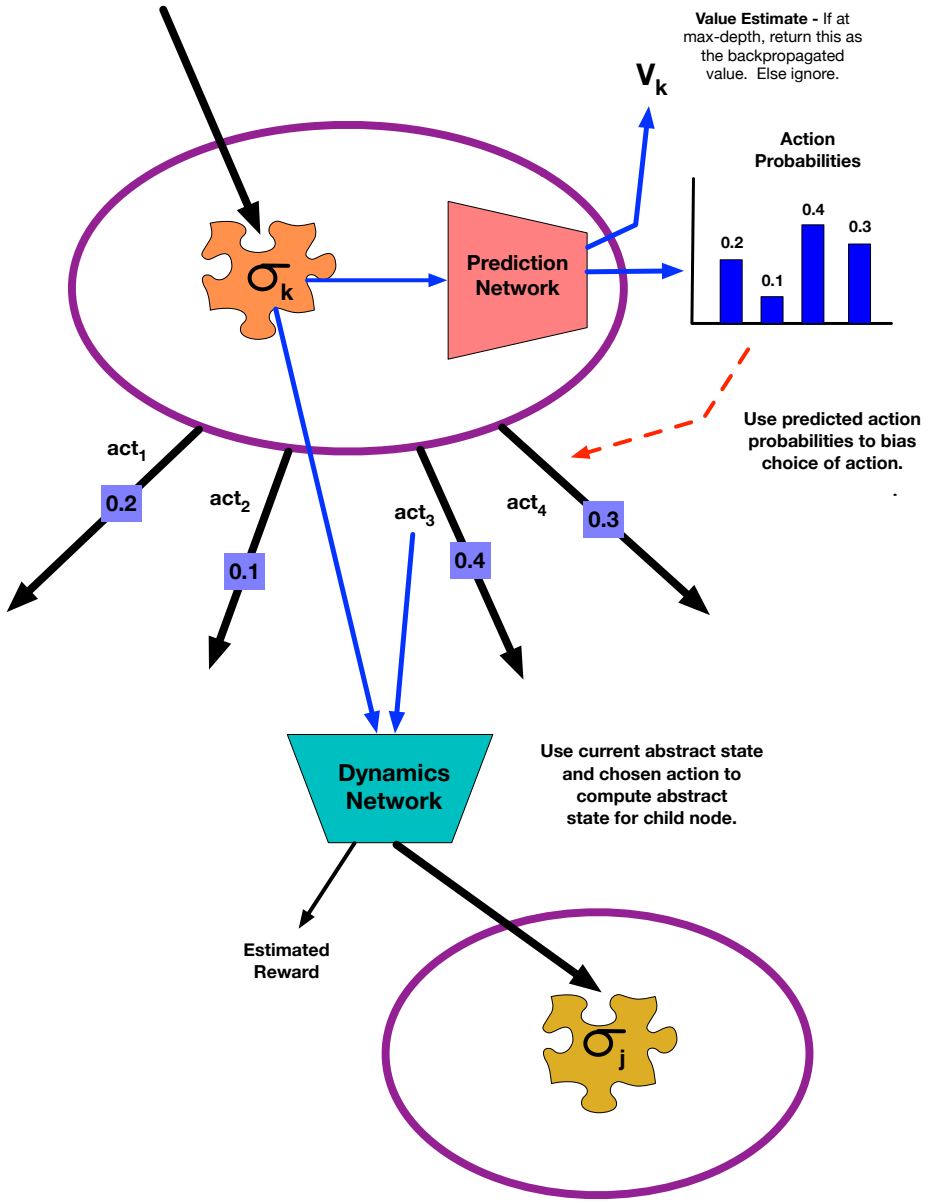
u-MCTS Algorithm

- Use sequence of current episode's game states ($s_{k-q} \ldots s_k$) as input to the Representation Network (NN_r), producing abstract state σ_k.
- Generate a root node (N_k) with state = σ_k.
- ** Use Tree Policy from root to a leaf node
- Expand the leaf node: generate all children.
- Randomly pick one child node (N_c).
- Rollout from N_c to a max-depth state (σ_m) using Dynamics Network (NN_d) and Prediction Network (NN_p). u-MCTS default: rollout just one level below N_c.
- Use NN_p to produce v_m (value of node σ_m). Backpropagate v_m along the vine $\sigma_m \rightsquigarrow N_k$, updating Q values and visit counts on all edges.
- Go to **

After building the tree via many simulations (i.e. loops in above algorithm):

- Use visit counts as basis for a probability distribution (D_k).
- Sample from D_k to get next move in the current episode.
- Save pair ($s_{k-q} \ldots s_k, D_k$) (along with value(N_k), episode actions and rewards) as training case for the $NN_r + NN_d + NN_p$ aggregate network.

u-MCTS Rollouts



AlphaZero -vs- MuZero

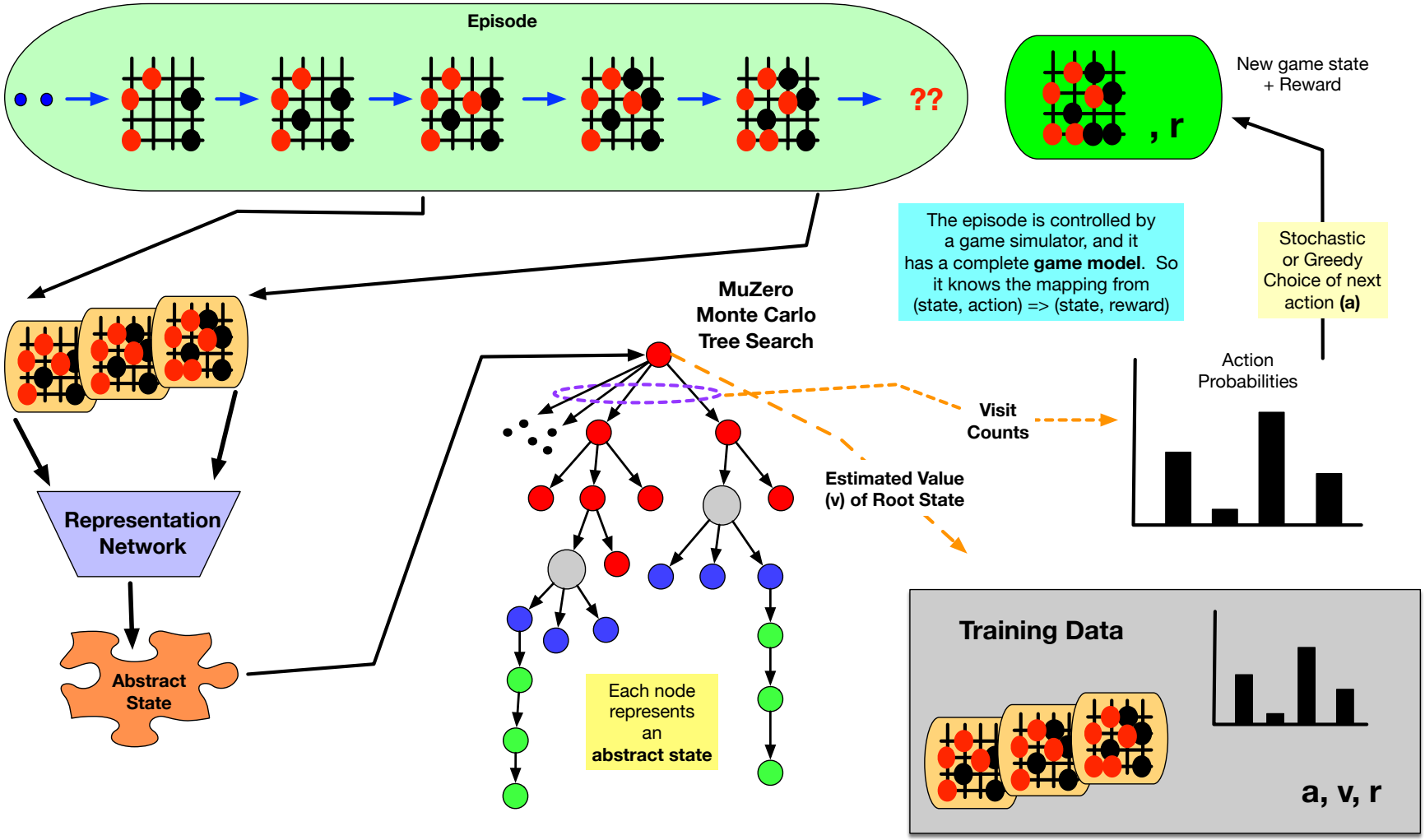
- AlphaZero uses the game model(M) in **both** actual-game episodes and MCTS.
- MuZero uses M in actual-game episodes but not in u-MCTS.
- u-MCTS **only** works with abstract states (σ_k), so u-MCTS a) cannot recognize any abstract states as *final*, and b) cannot classify actions as legal or illegal w.r.t. $\sigma_k \forall k$.

→ All actions (from the complete set of real-game acts) must be considered as possible for $\sigma_k \forall k$ (except for the root state, which does have a direct correspondence to a sequence of real-game states ($s_{k-q} \ldots s_k$) and thus only permits actions that are legal for s_k).

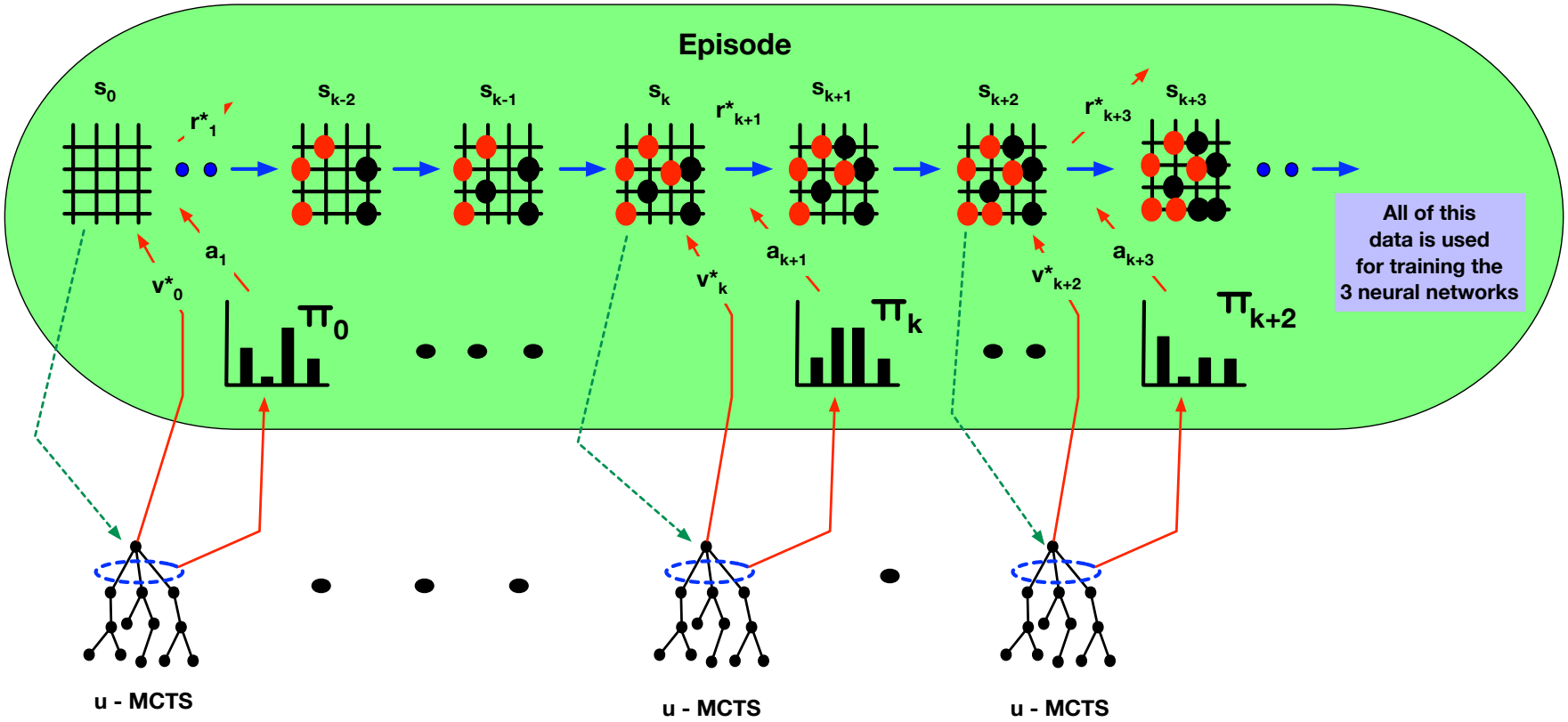
→ All u-MCTS searches must go to a fixed a) max depth or b) number of steps beyond a tree leaf.

- u-MCTS tries to learn a different model (M_d) via it's dynamics network (NN_d) and training data from the real-game episodes.
- Some of that training data comes from u-MCTS, so this is one more example of bootstrapping in RL.
- AlphaZero has mainly been applied to 2-person, deterministic, complete-information games that typically have few intermediate rewards and a clear final state.
- MuZero is designed for 1-person games / puzzles with complex pixel input that may have many intermediate rewards and may not have a final state (a.k.a. a continuous game; e.g. Atari), although it also works well for 2-person board games such as chess and go.

Generating Training Data

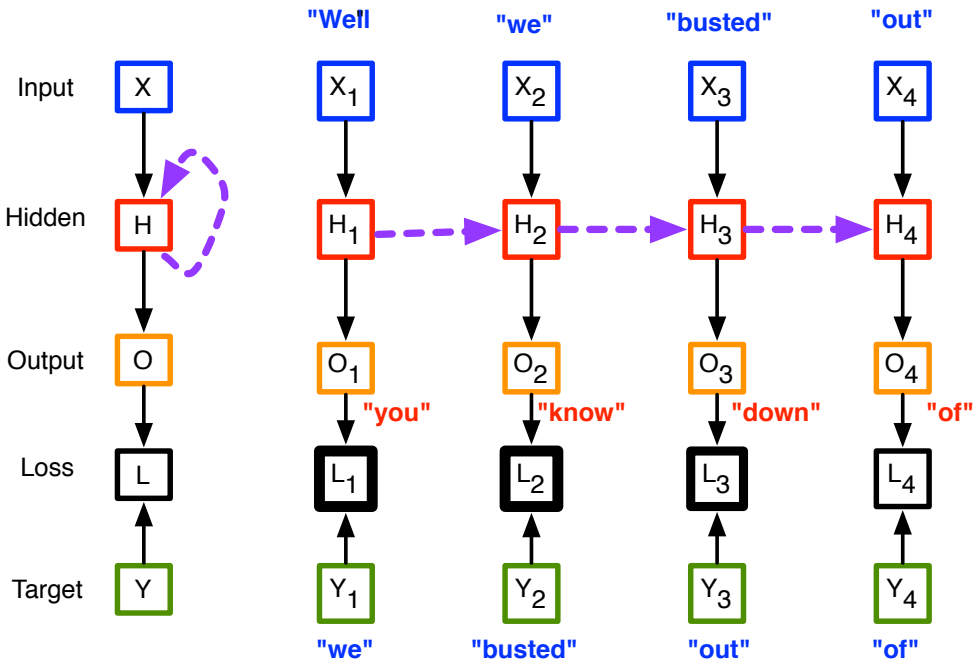


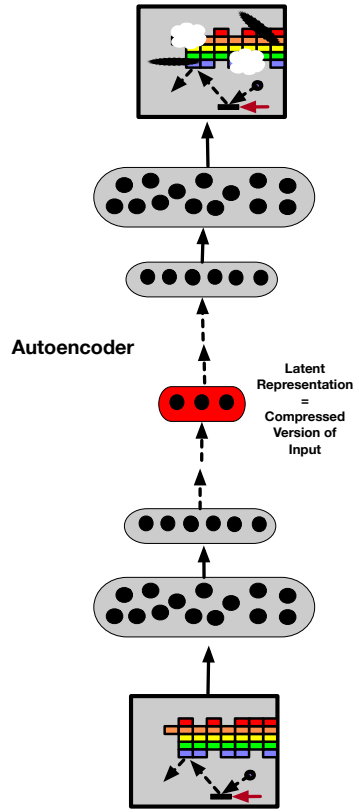
Training: u-MCTS Drives Game Episodes



Input to u-MCTS is actually a sequence of states, e.g. $s_{k-q} \ldots s_k$, which is not shown to avoid visual clutter.

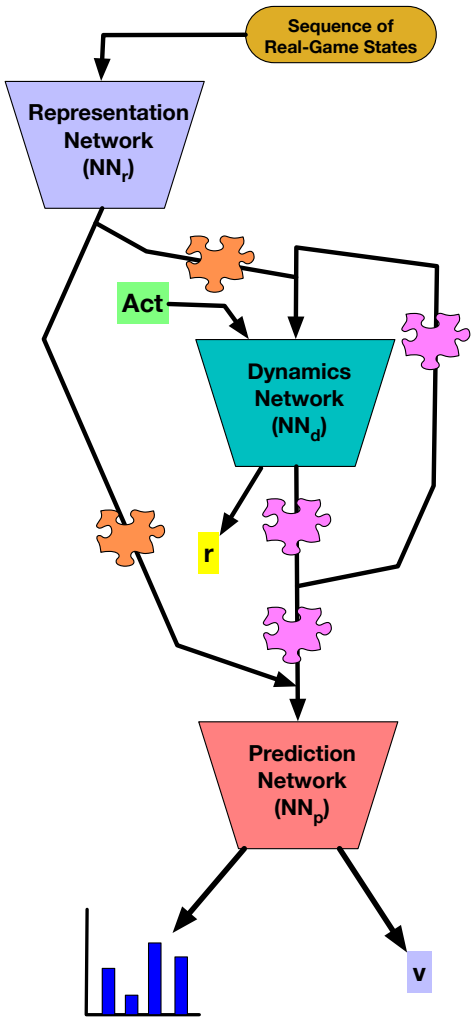
Hidden States in Elman Networks

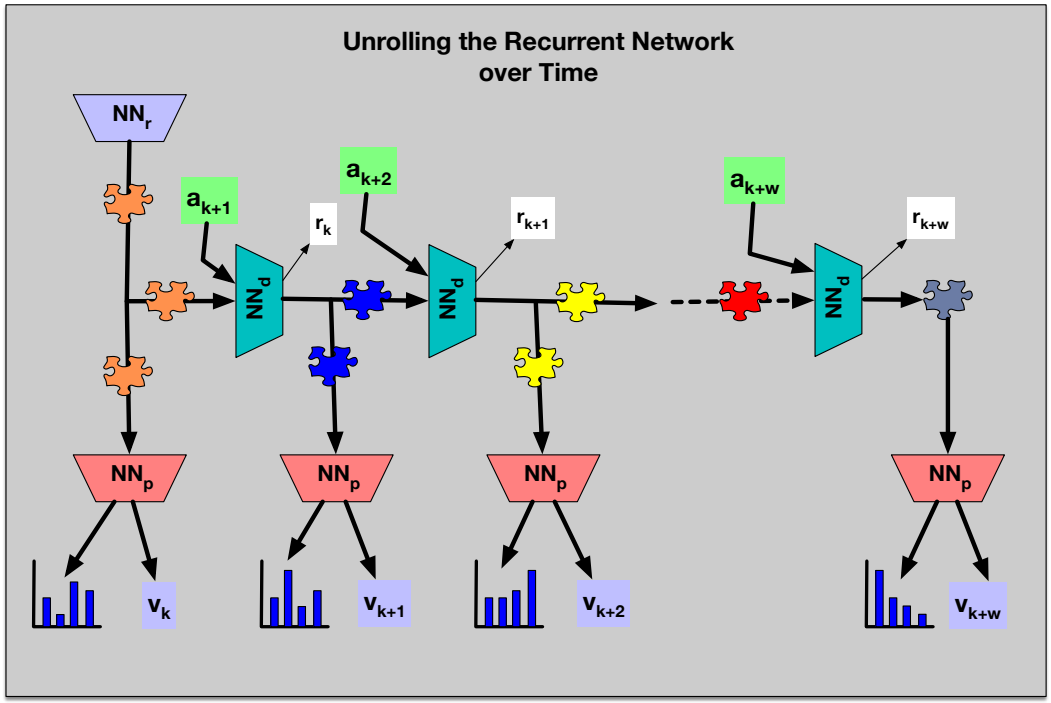




- The hidden states that feed back to H (unrolled as $H_1 \rightarrow H_2 \rightarrow H_3 \rightarrow H_4$) are **not** compact, lossless representations of earlier text.
- They are abstract patterns that help predict the next word based on some useful functional summary of the past.
- This is a lot different from the hidden layers of autoencoders, which can be decoded into something similar to the original input.

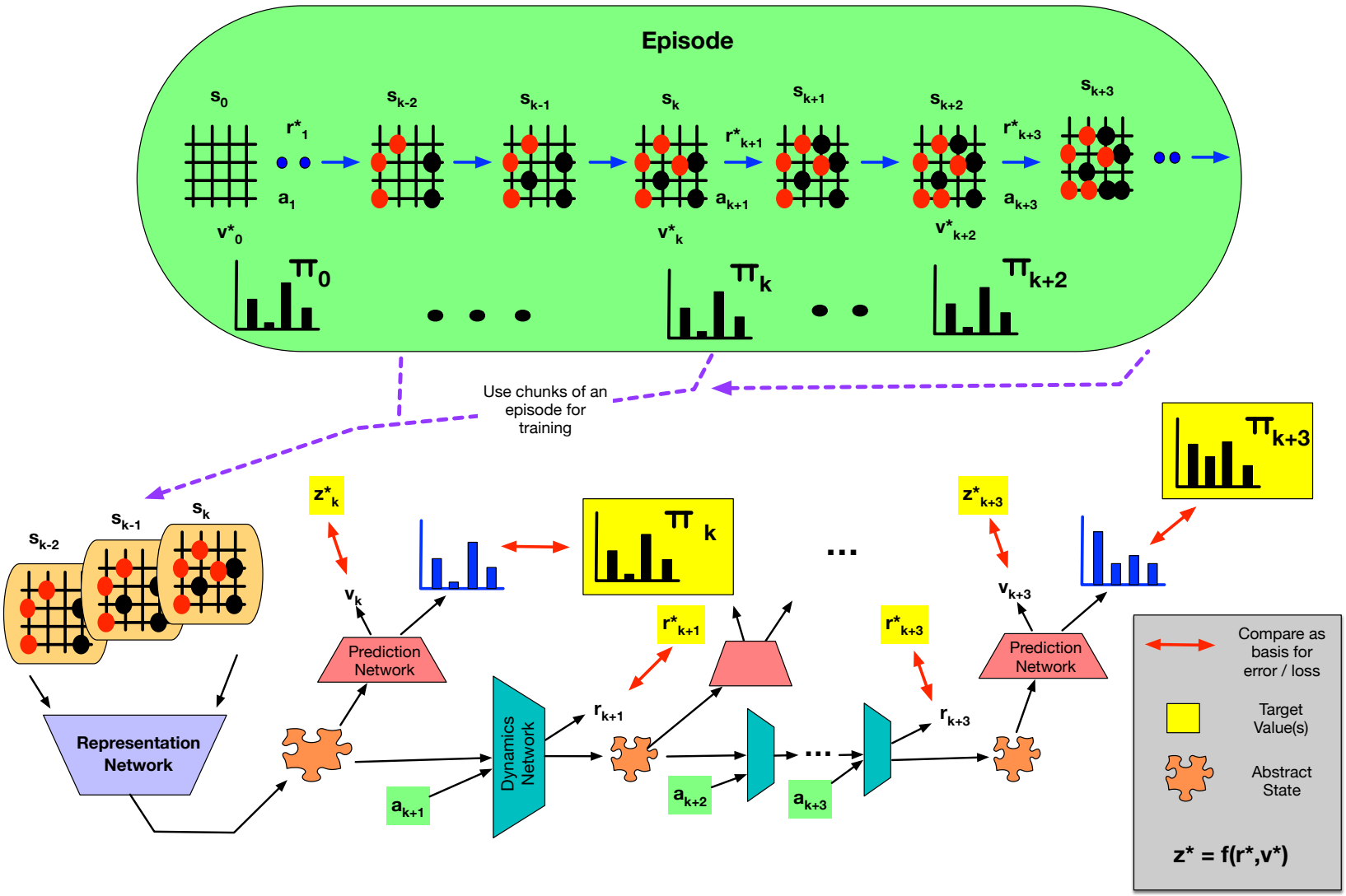
MuZero uses One Big Elman Net





The abstract states are **not** lossless compressions of real states, but patterns that summarize earlier real-game states and actions in a way that provides functional context for predicting proper policies, values and rewards.

Training (BPTT): Episode Provides Features + Targets



Back Propagation Through Time (BPTT)

Computing target state values: z_k^*

$$z_k^* = r_{k+1}^* + \gamma r_{k+2}^* + \gamma^2 r_{k+3}^* + \ldots + \gamma^{w-1} v_{k+w}^*$$

- r_j^* = Reward going from state s_{j-1} to state s_j in the real-game episode.
- γ = Discount
- v_j^* = Value of abstract state σ_j as produced by u-MCTS when σ_j is the root state.
- If the episode ends at a final state, s_f, then use the exact evaluation, $value(s_f)$, instead of v_j^*, which is just an estimate.
- Final states in the episode always have an exact evaluation: it's a win, loss or tie.
- Many video games are continuous and do not have a well-defined final state. So evaluations are based entirely on the collection of intermediate rewards.

The Loss Function for MuZero's BPTT

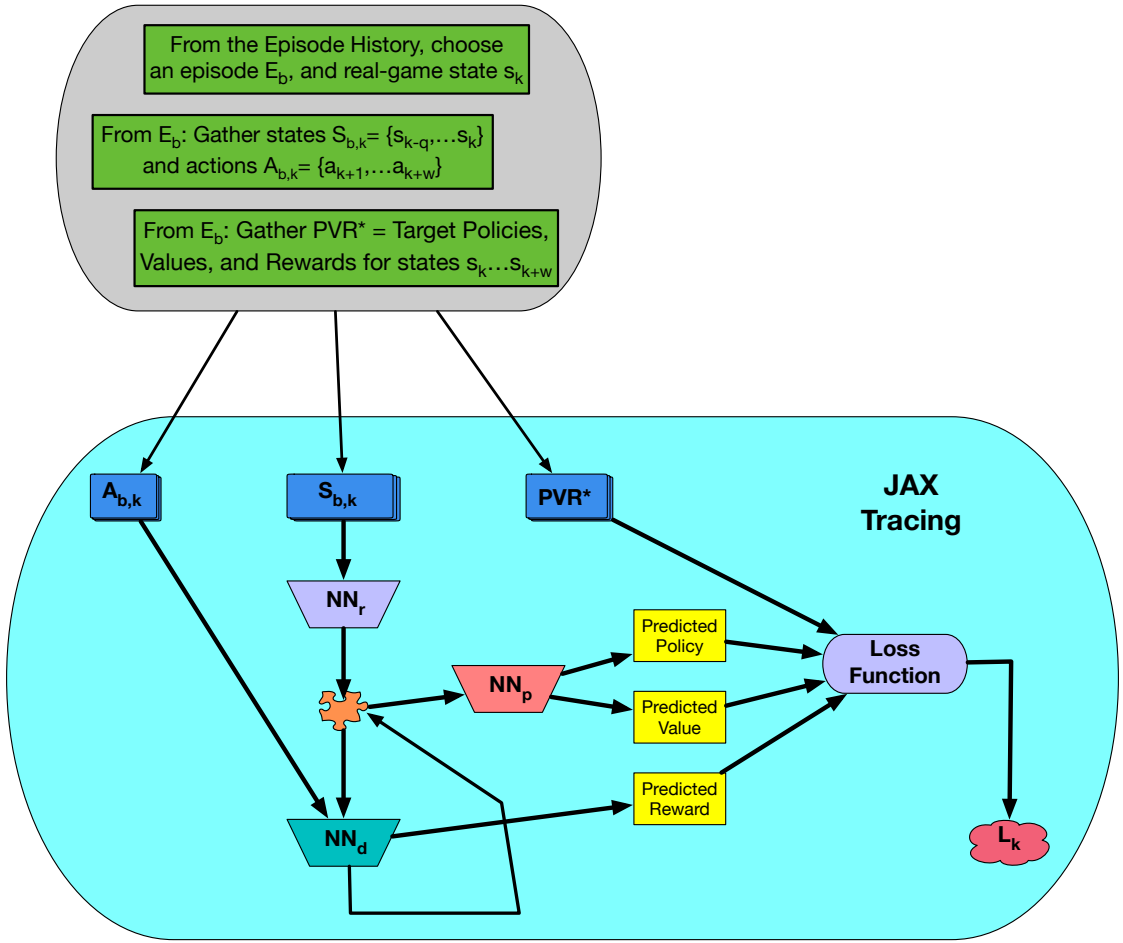
$$L_k = \frac{1}{w+1} \sum_{i=0}^{w} \underbrace{c_r(r^*_{k+i} - r_{k+i})^2}_{reward} + \underbrace{c_v(z^*_{k+i} - v_{k+i})^2}_{value} + \underbrace{c_p E_x(\pi_{k+i} - p_{k+i})}_{policy}$$

where:

- c_r, c_v, c_p = constants for error of reward, value and policy, respectively.
- π_k = policy (probability distribution) produced by u-MCTS when given real-game states $s_{k-q} \ldots s_k$ as input (i.e. the basis for its root abstract state).
- p_k = policy (probability distribution) produced by the prediction network (NN_p) when given abstract state σ_k.
- E_x = an error function for comparing two probability distributions, for example cross-entropy.
- Squared differences imply use of mean-squared error for reward and value, but other error functions (such as absolute values of differences) are also possible.

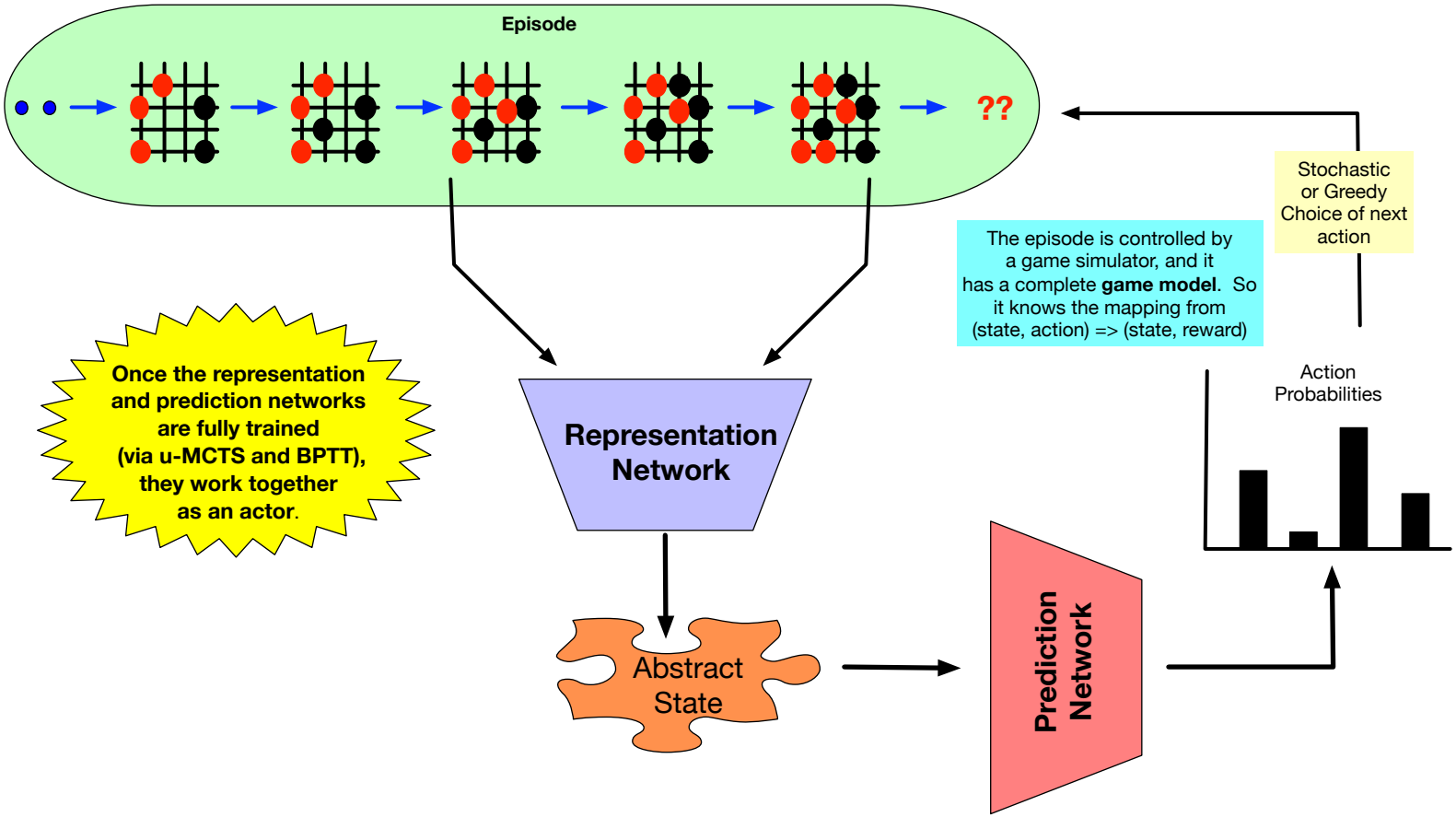
** Deepmind's official MuZero loss function also includes a *regularization term* that sums the absolute values of all weights. This penalizes networks with many large weights, which is often a sign of memorization instead of generalization.

Neural Network Training Algorithm: One Minibatch



- For parameters θ_r, θ_d and θ_p (of NN_r, NN_d and NN_p, respectively),
- JAX computes: $\frac{\partial L_k}{\partial \theta_r}$, $\frac{\partial L_k}{\partial \theta_d}$ and $\frac{\partial L_k}{\partial \theta_p}$.
- Use $\frac{\partial L_k}{\partial \theta_r}$, $\frac{\partial L_k}{\partial \theta_d}$ and $\frac{\partial L_k}{\partial \theta_p}$ and the learning rate(s) to update θ_r, θ_d and θ_p.

Trained Networks ⇒ Actor



* Note: The dynamics network is no longer needed. As it learns the abstract-state transitions and rewards, it serves as scaffolding for training the representation and prediction networks.

Overview: RL with MuZero

- Randomly initialize parameters of the 3 Neural Networks: $(NN_r, NN_d, NN_p) = \Psi$
- Perform many episodes, with each action of each episode determined by probability distributions returned by u-MCTS.
- Use Ψ to govern u-MCTS initialization and search in a space of abstract states.
- Combine (a) actual states, (b) actions and (c) rewards from each episode with (d) probability distributions and (e) state values produced by u-MCTS to yield training cases for Ψ.
- After every mth episode, train Ψ using back propagation through time (BPTT) by drawing random sequences from the episodic data.
- After running $n \ggg m$ episodes, and $\frac{n}{m}$ rounds of BPTT, consider Ψ fully trained.
- Use NN_r and NN_p to provide a real-time policy for a game-playing agent.