Table of contents

12 Foundation, World and Transformer Models (Advanced Topic)

Model-Based Reinforcement Learning

Last Module: Actor critic (policy gradient)

Previous Modules: Value function approximation using deep learning

This Module: Combining deep learning, tree search and transformer models

AlphaGo, AlphaZero & MuZero

AlphaGo - Core Idea

First system to *combine deep learning and tree search* for superhuman play.

Domain: *Go* only.

Pipeline integrates:

1. *Supervised learning* from expert games of human players
2. *Reinforcement learning* through self-play
3. *Monte Carlo Tree Search (MCTS)* guided by neural networks

AlphaGo - Neural networks & losses

Neural Network	Description
Policy Network (π_1)	Trained *supervisedly* on human moves; 13-layer CNN (Go board 19×19 × 48 planes)
Policy Network (π_2)	Refined by self-play RL (same architecture)
Value Network (v)	13-layer CNN + 2 fully connected layers; outputs scalar win probability $v(s)$

Training objectives:

$$\mathcal{L}_\pi = -\log \pi_\theta(a^*|s), \qquad \mathcal{L}_v = (v_\phi(s) - z)^2$$

where $z \in \{-1, +1\}$ is the game outcome.

AlphaGo Planning integration

MCTS uses:

- *Policy prior* $\pi_\theta(a|s)$ from policy network $\pi 1$ (parameters θ) $\rightarrow$ biases search toward likely moves
- *Value estimate* $v_\phi(s)$ from value network v (parameters ϕ) $\rightarrow$ evaluate leaves

Move selection at root:

$$\pi_{\mathrm{MCTS}}(a|s_0) \propto N(s_0, a)^{1/\tau}$$

Once π_2 is trained through reinforcement learning, AlphaGo uses it to play millions of games against itself.

- Each game produces pairs (s, z): (s_t, z_t) where $z \in \{-1, +1\}$ is the game outcome.

These are used to train the value network $v_\phi(s_t)$ via regression:

$$\min_\phi \left(v_\phi(s_t) - z_t\right)^2$$

- So the value network learns to predict *who* will win from any board position that strong play (i.e., π_2) would reach.

Achieved 4–1 win vs Lee Sedol (2016)

AlphaZero - Unified Self-Play RL

Extends AlphaGo $\longrightarrow$ *Go*, *Chess*, *Shogi*

- Removes human data and hand-crafted rollout policy
- Fully self-play training loop

AlphaZero - Neural network and objective

Single residual CNN shared by policy + value

- 20 or 40 ResNet blocks, 256 filters, BatchNorm + Rectified Linear Circuit (ReLU)
- Input: stack of board planes (19×19×N)
- Heads:
  - *Policy head:* 1 conv + 1 FC $\rightarrow$ softmax over legal moves
  - *Value head:* 1 conv + 2 FC $\rightarrow$ scalar $v_\theta(s)$

Loss:

$$\mathcal{L}(\theta) = (z - v_\theta(s))^2 - \pi_{\mathrm{MCTS}}^\top \log \pi_\theta + c\|\theta\|^2$$

AlphaZero - Learning & Planning Loop

$$\text{Network} \Rightarrow \text{MCTS} \Rightarrow \text{Self-play games} \Rightarrow \text{Network update}$$

- *MCTS*: ~800 simulations per move
- *Network*: ResNet trained via SGD on MCTS targets
- Unified architecture simplified training → superhuman performance across games

MuZero - Learning to Plan Without Rules

- AlphaZero still needed explicit game rules
- *MuZero* learns a *latent model* of dynamics for planning without knowing rules
- Same MCTS framework, but search happens in latent space

MuZero - Neural networks (3)

Neural Network	Function	Notes
Representation h_θ	Observation $\rightarrow$ latent state s_0 (learns (latent) state representation)	6 ResNet blocks for Atari (pixels $\rightarrow$ latent)
Dynamics g_θ	Predicts s_{t+1}, r_{t+1} from (s_t, a_t) (learns model)	Small conv stack + reward head
Prediction f_θ	Outputs policy p_t and value v_t from s_t	Two heads (softmax policy, scalar value)

MuZero – TD-Style Learning (Bootstrapped Returns)

Unlike AlphaZero (which uses full-episode Monte Carlo targets), *MuZero* trains its value network using n-step bootstrapped (TD) returns

For each step (t), the target value is:

$$\hat{v}_t = \sum_{i=0}^{n-1} \gamma^i r_{t+i} + \gamma^n v_\theta(s_{t+n})$$

- Combines *observed rewards* and *bootstrapped value* from the predicted future state
- Allows *credit assignment* across long horizons without waiting for episode termination

MuZero minimises a combined loss:

$$\mathcal{L} = \sum_k \left[(v_k - \hat{v}_k)^2 + (r_k - \hat{r}_k)^2 - \pi_k^\top \log p_k \right]$$

- *Value loss:* TD-style bootstrapped error
- *Reward loss:* immediate reward prediction
- *Policy loss:* cross-entropy with MCTS visit-count distribution

TD bootstrapping makes MuZero more sample efficient than AlphaZero

Planning (MCTS) provides strong policy/value targets; TD updates keep learning continuous

MuZero - Training and Integration

MCTS operates within the learned model:

$$s_{t+1}, r_t = g_\theta(s_t, a_t)$$

Targets from MCTS train all three nets end-to-end

Loss:

$$\mathcal{L} = \sum_k \left[(v_k - \hat{v}_k)^2 + (r_k - \hat{r}_k)^2 - \pi_k^\top \log p_k \right]$$

- Achieves AlphaZero-level play in Go/Chess/Shogi and strong Atari results from pixels

MuZero - Results and Significance

Domain	*Training time to superhuman level*	*Benchmark / Opponent*	*Notes*
Chess	≈ 4 hours (on 8 TPUv3 pods)	*Stockfish*	Surpassed world-champion chess engine performance
Shogi	≈ 2 hours	*Elmo*	Surpassed leading professional Shogi engine
Go	≈ 9 hours	*AlphaZero / KataGo*	Matched AlphaZero's superhuman play using only learned dynamics
Atari (57 games)	~200M frames	*Rainbow / IMPALA*	Exceeded or matched best model-free RL baselines across games

Key insights

- *No rules given:* MuZero learned *dynamics, value, and policy* purely from experience
- *Unified algorithm:* Same architecture and hyperparameters across all domains
- *Planning efficiency:* Performed *Monte Carlo Tree Search (MCTS)* entirely in *latent space*
- *Sample efficiency:* Achieved AlphaZero-level play within *hours of self-play training*

AlphaGo, AlphaZero & MuZero Comparison

System	# of NNs	Architecture	Uses known rules?	Learns model?	Planning
AlphaGo	2 (policy + value)	13-layer CNNs	✓	✗	MCTS with rules
AlphaZero	1 (shared policy-value ResNet)	20–40 ResNet blocks	✓	✗	MCTS with rules
MuZero	3 (h, g, f modules)	ResNet latent model	✗	✓	MCTS in latent space

Foundation, World and Transformer Models

Three Layer Agent Stack

Layer	Role (e.g. of an Agent or Robot)	Typical tools
Cognition / Reasoning	Query answering, Programming, Multi-step thinking: goal parsing, task decomposition, tool selection, hypothesis & plan revision, safety checks	LLM w/ *CoT / ToT/GoT*, value-guided decoding, process rewards
Semantic Policy (Vision–Language–Action)	Ground instructions & scene into actionable subgoals / waypoints	*RT-X / RT-2-X* (VLA Transformer), affordance & object-centric models
Control / Dynamics	Execute precise motions, stabilize, react to feedback	*Dreamer V3 / TD-MPC2 / Diffusion Policy* (model-based or policy learning)

CoT: Chain of Thought

ToT: Tree of Thought

GoT: Graph of Thought

VLA: Visual, language, Action Transformer

MPC: Model Predictive Control

Modern Trends

Sequential decision-making is on the ascendant: RL, model-based control, and planning-like reasoning are central to agents, robotics, and tools using tensor flow (transformer) architectures—so "planning" and "RL" must live inside differentiable, scalable systems.

Limiting assumptions: Classical planning & Reinforcement Learning's typical teaching setup (fully observable, deterministic, stationary, discrete) mismatches many modern AI settings.

Compute & tooling: Tree search doesn't map cleanly onto GPU/TPU throughput the way dense tensor ops do; differentiability matters for end-to-end training, credit assignment, and integration with deep stacks.

Requirements of RL for Foundation and World Models

Requirements of RL (additive)	Approach / System
Non-deterministic (Stochastic)	Policy Gradient
Non-stationary (Generalisation)	Value Approximation
Partially observable (Epistemic)	Actor-Critic
Differentiable (Nnet/GPU integration)	SAC, Dreamer V3
Distributed (Industrial Scaling)	IMPALA, V-trace
Agentic (Layers)	RT-X, LLM, World Models

We explore requirements of differentiable, distributed and agentic RL needed for foundation and world models

Differentiable Planning: Soft Actor Critic (SAC)

Greedy next-step choice using max; defines the Bellman optimality operator used in Q-learning/DQN.

$$Q^*(s,a) = r(s,a) + \gamma\, \mathbb{E}_{s' \sim P}\left[\max_{a'} Q^*(s',a')\right]$$

Soft (Entropy-Regularized) Bellman Backup - “softmax”

$$Q_{\text{soft}}(s,a) = r(s,a) + \gamma\, \mathbb{E}_{s' \sim P}\left[V_{\text{soft}}(s')\right]$$
$$V_{\text{soft}}(s) = \mathbb{E}_{a' \sim \pi(\cdot|s)}\left[Q_{\text{soft}}(s,a') - \alpha \log \pi(a'|s)\right]$$

- Adds entropy bonus (temperature α) ⇒ softens the hard max.
- As $\alpha \to 0$: $V_{\text{soft}}(s) \to \max_{a'} Q(s,a')$ (recovers hard backup).

Implementation note (SAC): a *min* over two critics, $\min\{Q_{\theta_1}, Q_{\theta_2}\}$, is often used to reduce overestimation bias (Double-Q trick), not as the backup operator.

This relaxation allows *gradients to flow* through the planning step.

Differentiable Planning: Dreamer V3

Era	Key system	What it added	Influence on Dreamer V3
2019	*PlaNet* (Hafner et al.)	Latent dynamics model (RSSM) + Cross-Entropy Method (CEM) planning in latent space	Showed model-based imagination from pixels works
2020-22	*Dreamer V1–V2*	Replaced CEM with *actor-critic training in imagination* (no explicit planning), making everything differentiable	More efficient, easier to train on GPU
2022-24	*TD-MPC / TD-MPC 2* (Hansen et al.)	Combined short-horizon latent Model Predictive Control (MPC) with *TD learning*, strong continuous-control results	Reinforced ideas of gradient-based MPC and temporal-difference consistency
2023-24	*Dreamer V3*	Unified architecture: one RSSM world model + imagination-based actor-critic + robust scaling across domains	Synthesizes both model-based planning and policy-gradient RL advantages

Dreamer V3 is developed by DeepMind.

Recurrent State Space Model (RSSM): Dreamer V3

Dreamer V3 uses a learned latent world model and *imagination* (planning during training) and is fully differentiable.

- It uses a recurrent state space (RSSM) world model (both stochastic and recurrent) to imagine trajectories for policy/value learning.
- Think of the RSSM as a latent recurrent simulator

An RSSM is implemented as a recurrent neural network (RNN) and unrolled through time

- Once the RSSM is trained, Dreamer can roll out future trajectories purely in its imagination
- RSSM is *unrolled through time* during training and “imagination” to simulate trajectories

Comparison of MuZero with Dreamer V3

Aspect	*MuZero*	*Dreamer V3*
Core idea	Combines learning + *Monte-Carlo Tree Search (MCTS)* in latent space	Learns a *Recurrent State-Space Model (RSSM)* and performs *differentiable imagination rollouts*
Planning form	Expands a *search tree*: $s0 \rightarrow s1, s2, \ldots$	Rolls out a *latent sequence*: $(h, z)t \rightarrow (h, z)t{+}1 \rightarrow (h, z)t{+}2 \ldots$
Model components	Representation $h(o_t)$, Dynamics (g(s,a)), Prediction $f(s)$	RSSM with deterministic h_t and stochastic z_t states
Computation	Search-based, *CPU-heavy*, not fully differentiable	*GPU-friendly*, fully differentiable RNN unrolled through time
Learning loop	Tree search generates improved policies; network distils them via supervised losses	Actor–critic trained entirely on *imagined trajectories* from the RSSM
Search structure	Discrete branching, value backups	Sequential imagination, no branching
Output policy	Derived from *visit counts* in the search tree	Learned directly through gradient updates in imagination
Analogy	*"Plan by explicit search"*	*"Plan by differentiable imagination"*

MuZero: explicit look-ahead search $\Longleftrightarrow$ **Dreamer: continuous latent imagination**

Take-away summary

- For most continuous-control, robotics, or fine-tuning tasks: actor-critic / policy-optimisation (PPO, SAC) are easier, faster, and competitive.
- For structured combinatorial or look-ahead-heavy tasks: planning-based hybrids (MuZero, Sampled MuZero, EfficientZero) can still outperform, but with higher engineering and compute cost.

Trend: research is moving toward differentiable world models (DreamerV3) that keep MuZero's model-based benefits while retaining the simplicity and efficiency of actor–critic learning—essentially bridging the two families.

Distributed RL: IMPALA & V-trace

Importance Weighted Actor-Learner Architecture (IMPALA)

- In production in DeepMind, OpenAI & Google DeepRL
- *Decoupled actor-learner*: many CPU actors generate trajectories under behaviour policy μ; a central GPU learner updates π.
- *High throughput* via batched unrolls (e.g., length (n)); supports RNNs (LSTM) and multi-task.
- *Challenge*: policy lag $\rightarrow$ off-policy data.
- *Solution*: *V-trace* targets for stable off-policy learning.

Off-policy with *correction* that handles *policy lag* without sacrificing throughput

IMPALA was developed at DeepMind

Distributed RL: V-trace essentials

Let importance ratios $\rho_t = \min\left(\bar{\rho}, \frac{\pi(a_t|x_t)}{\mu(a_t|x_t)}\right)$

$c_t = \min\left(\bar{c}, \frac{\pi(a_t|x_t)}{\mu(a_t|x_t)}\right)$ with $\bar{\rho} \geq \bar{c}$

Value target (per time s)

$$\delta_t^V = \rho_t\Big(r_t + \gamma V(x_{t+1}) - V(x_t)\Big), \quad v_s = V(x_s) + \sum_{t=s}^{s+n-1} \gamma^{t-s}\left(\prod_{i=s}^{t-1} c_i\right)\delta_t^V$$

Policy gradient with V-trace advantage

$$A_t^{\text{V-trace}} = r_t + \gamma v_{t+1} - V(x_t), \qquad \nabla\theta J \propto \rho_t \nabla\theta \log \pi\theta(a_t|x_t)\, A_t^{\text{V-trace}}$$

Loss (typical)

$$\mathcal{L} = \underbrace{\mathbb{E}\big[(v_s - V(x_s))^2\big]}_{\text{value}} - \beta\underbrace{\mathbb{E}\big[\rho_t \log \pi(a_t|x_t)\, A_t^{\text{V-trace}}\big]}_{\text{policy}} - \eta\underbrace{\mathbb{E}\big[\mathcal{H}(\pi(\cdot|x_t))\big]}_{\text{entropy}}$$

Why it works

- *Clipped IS ratios* (ρ_t, c_t) tame variance/bias;
- *Multi-step correction* handles *policy lag* without sacrificing *throughput*.

Representative efficient actor-critic methods

Category	Example algorithms	Key strengths
On-policy	PPO	Stable, parallelizable, easy; standard in LLM fine-tuning (RLHF)
Off-policy (stochastic)	SAC	Maximum-entropy objective → robust exploration; excellent data efficiency
Distributed	IMPALA, V-trace	Massive scalability; production in DeepMind, OpenAI, Google DeepRL

Efficiency and Performance Comparison

Dimension	MuZero / Sampled MuZero / EfficientZero	PPO / SAC / IMPALA
Sample efficiency	Excellent when planning can reuse a model (Atari, board games)	High for off-policy (SAC); moderate for PPO
Wall-clock / GPU efficiency	Poor (search is serial & CPU-bound)	Very good (fully parallel on GPU)
Robustness & stability	Sensitive to model errors / rollout length	Stable with tuned hyper-parameters
Scalability to real-time tasks	Hard (search latency)	Good; used in robotics, continuous control, large-scale RL (IMPALA, V-trace)
Best-case performance	Outstanding in structured domains (Go, Atari)	State-of-the-art in most continuous-control and real environments

RL for Frontier & World Models

For training:

- PPO, DPO & GRPO

For Query

- Self-consistency
- Tree of Thought (beam-style)
  - Maintain and progress frontier nodes in parallel
  - Value-Guided Decoding

Self-Consistency for LLM Reasoning (on Chains of Thought)

Idea (Wang et al., 2023):
Instead of trusting a single Chain-of-Thought (CoT), *sample many diverse CoTs* and *aggregate their final answers*.

- Majority (or verifier-weighted) agreement $\approx$ more reliable reasoning.

Self Consistency

1. *Prompt* the same question with CoT enabled ("think step-by-step")
2. *Sample* K reasoning paths with stochastic decoding (e.g., temperature $0.7-1.0$, nucleus $p \approx 0.9$).
3. *Extract final answers* from each path.
4. *Aggregate*
    - *Majority vote* over final answers (self-consistency).
    - or *Score with a verifier/rubric* and pick highest-scoring.
    - Optional: *consensus check* (e.g., numeric tolerance).

Why it helps

- Diversity $\rightarrow$ reduces single-path errors/hallucinations.
- Voting/verification $\rightarrow$ filters spurious but fluent chains.

How paths are progressed

- *CoT:* each path is a linear sampled chain.
- *ToT (Tree-of-Thought):* expand multiple partial chains (branching), keep top beams.
- *GoT (Graph-of-Thought):* allow branches to *merge/reuse* subresults; select best subgraph.
- In all cases, *progress = decode next step* (sample or beam), *prune* with a heuristic/verifier, *repeat* until a stopping rule.

Practical settings

- K: $10-40$ (cost vs. accuracy).
- *Extractor:* robust regex/templates for the final answer.
- *Verifier:* separate model or rules (units, constraints, tests).
- *Failure mode:* consistent but wrong consensus $\rightarrow$ add tools/checks (calculator, code, retrieval).

Self-consistency = ensemble of CoTs + aggregation; ToT/GoT generalize progression by branching/merging before voting or verification.

Vision Transformers (ViTs)

Since ~2020, attention-based Transformers have started competing and often surpassing CNNs in large-scale vision benchmarks.

Image $\rightarrow$ patches $\rightarrow$ tokens $\rightarrow$ transformer

- *Patchify the image:* split an image of size (HW C) into non-overlapping patches (PP).
  Number of tokens $N = \frac{HW}{P^2}$.

- *Linear patch embedding:* flatten each patch $x_i \in \mathbb{R}^{(P^2C)}$ and project
  $z_i^0 = W_E x_i + b_E \in \mathbb{R}^D$.
  (Often implemented as a conv with kernel/stride P.)

Vision Transformers - Tokens and Transformer Encoding

- *Add a class token and positions:* prepend $[\mathrm{CLS}]$ and add learnable positions $\tilde{z}_i^0 = z_i^0 + p_i$, with sequence $[\tilde{z}_{\mathrm{CLS}}^0, \tilde{z}_1^0, \ldots, \tilde{z}_N^0]$.
- *Transformer encoder stack (repeated (L) times):*

  $\mathrm{SA}(X) = \mathrm{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$ with multi-head self-attention,

  then MLP; both with residuals + layer norm.
- *Prediction head:* take the final $[\mathrm{CLS}]$ (or pool all tokens) $\rightarrow$ linear head () class probs.
- *Note:* smaller $P \Rightarrow$ more tokens (detail ↑, cost ↑); larger $P \Rightarrow$ fewer tokens (detail ↓, cost ↓).
  Variants like *Swin* use local windows with shifts for scalability; ViT uses global attention.

RT-X Transformers in Robotics

RT-X:

Increasingly transformers are also being used for robotics (e.g. RT-1, RT-2, RT-X Google DeepMind)

- large-scale imitation across many robots.

“RT-family” includes hybrid attention across vision, language, and control.

- They utilises Visual, Language, Action (VLA) transformers

RL Fine Turning & Query optimisation

Foundation Model	RL/Query Optimiser	Example
Attention-based transformer / *LLM*	PPO, DPO, GRPO / CoT / ToT/GoT	Gemini 2.5, ChatGPT 3.5-5.0, ChatGTP Operator, Claude Computer Use, DeepSeek R1
Attention-based transformer / Vision + Language + Action (VLA)	PPO	RT-X
RSSN / Control	Diffusion Policy	Dreamer V3

Chain of thought (CoT) / Tree of thought (ToT) & Graph of Thought (GoT)

Convergence of RL, Autoregression & Transformers

Implicit planning through attention inside world models

Transformers can simulate “multiple futures” inside the hidden state

1. Self-attention makes “lookahead” possible
2. Attention learns which futures matter
3. Transformers can simulate “multiple futures” inside the hidden state
4. Implicit planning is amortised planning
5. The policy uses the world model’s implicit planning

Autoregression & joint distribution factorisation

Autoregression models a sequence by predicting each element from all previous elements:

$$x_t = f(x_{1:t-1}) + \epsilon$$

Used in time series, sequence modelling, and autoregressive transformers.

Factorises a joint distribution as:

$$p(x_{1:T}) = \prod_{t=1}^{T} p(x_t \mid x_{1:t-1})$$

Masked latent transformers

A masked latent transformer is a transformer that models sequences of latent states, using attention masks to enforce causality or planning structure just like in autoregressive language models.

- It is a transformer that predicts (or refines) latent variables in a sequence, but only using allowed past or partial information, enforced through a mask.

Masked latent transformers appear in world-model RL, video generation, and planning-as-generation frameworks.

- They are increasingly used as a replacement for RNN/RSSM latent dynamics models in Dreamer-like

Attention-baed transformers (recap)

Transformers model sequences using self-attention, where each token computes weighted interactions with other tokens.

Key components:

- Token embeddings and positional encodings
- Self-attention:

$$\text{Attn}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

- Multi-head attention, feedforward layers, residual

Causal masking

Causal masking is used in autoregressive transformers:

- Each token attends only to past tokens
- Enforces the autoregressive condition

$$x_t \sim p(x_t \mid x_{1:t-1})$$

This masked attention mechanism directly carries over to masked latent transformers used in world-model RL.

Masked latent transformers: LLMs versus World Models

In NLP transformers in large language models:

- Tokens = words
- Mask = causal mask (can't see the future)

In masked latent transformers:

- Tokens = latent states z_t (learned hidden representations)
- Mask = ensures correct temporal, causal, or planning structure

Three major paradigms

Three major paradigms reflect the convergence of RL, autoregression and transformer models:

- Decision Transformer (offline, no planning),
- World-Model RL (planning, includes dynamics), and
- Actor-Critic Transformers (online, no dynamics).

Case Study: Actor-Critic Transformers

Actor-critic transformers are RL agents that use transformer architectures to parameterize the actor, critic, or both, while still relying on Bellman equations and policy gradients for learning.

- Actor-Critic Trnsformers essentially outperform LSTMs in long-horizon POMDP

Comparison

Dimension	Decision Transformer	World-Model RL	Actor–Critic Transformers
Core Idea	Offline sequence modelling of trajectories; imitate high-return behaviour	Learn a predictive dynamics model and plan or imagine futures inside it	Classical actor–critic RL but with transformer networks for policy/critic
Learns dynamics model?	✗ No	✔ Yes (explicit or latent dynamics)	✗ No
Does planning?	✗ No explicit planning	✔ Yes (explicit or implicit)	✗ No planning (just TD + policy gradient)
Uses Bellman equations?	✗ Never	✔ Often (Dreamer, MuZero)	✔ Yes (critic learning)
Uses TD learning?	✗ No	✔ Yes (usually; except pure planners like Trajectory Transformer planning mode)	✔ Yes
Training regime	**Offline only**	Typically **online**, but can combine offline + online	**Online RL**
Policy improvement	✗ None (no search, no DP)	✔ Yes (via planning or imagined rollouts)	✔ Yes (policy gradient)
Value function learned?	✗ No	✔ Yes (in most models)	✔ Yes (critic)
Reward used	Only to compute return-to-go (RTG) labels	Used in Bellman updates + imagination	Used in TD error for critic

Dimension	Decision Transformer	World-Model RL	Actor–Critic Transformers
Primary transformer role	Sequence → next action predictor (GPT-like)	Dynamics model + future predictor + latent planner	Memory encoder for policy/value
Handles partial observability?	✔ Through long context window	✔ Through latent state + prediction	✔ Through transformer memory
Long-horizon credit assignment	Weak (depends on data quality)	Strong (via imagined rollouts or implicit planning)	Moderate (TD propagation + attention)
Required data	High-quality offline trajectories	Real interactions + possibly offline data	Real interactions (on-policy or off-policy)
Exploration	✘ None	✔ Yes (through policy learning or planning)	✔ Yes (inherent to actor–critic)
Generalises beyond dataset?	✘ Mostly no	✔ Yes (model-based planning)	✔ Yes (online improvement)
Analogy	"GPT for actions"	"Agent learns a simulator and plans inside it"	"LSTM actor–critic upgraded to a transformer"
Example algorithms	Decision Transformer; Upside-Down RL	DreamerV2/V3, MuZero, Sampled MuZero, Trajectory Transformer (planning), AWM	GTrXL, Transformer-PPO, Transformer-SAC
Main strength	Simple, powerful offline learning	Efficient long-horizon reasoning and planning	Strong online learning with rich temporal representations
Main weakness	Cannot improve beyond dataset; no true RL	Dynamics learning is hard; model bias	No planning; no model; can be sample-inefficient