MBRL(Discrete actions)

Model-based RL offers several advantages over model-free methods:

- sample efficiency
- adaptation to changing rewards and dynamics
- exploration
  Disadvantages would include deployment (planning) costs.
  There are many types of models one can define and learn:
- transition model : $s_{t+1} = f(s_t, a_t)$
- reward model: $r_{t+1} = f(r_t, a_t)$
- inverse dynamics model: $a_t = f^{-1}(s_t, s_{t+1})$
- distance model: $d_{if} = f_d(s_i, s_j)$
  In MBRL literature, mostly transition and/or reward models are learned

Parametric vs non-parametric

Parametric models include neural networks and physics-based methods.
Non-parametric models include GMMs, GPs, Decision Trees.
Parametric models are highly expressive but can be hard to train. GPs are highly sample efficient, but does not generalize well to higher dimensional state inputs

Inputs
The current practice is to learn a useful compressed latent representations from raw data(via VAEs) and use these features as inputs to the model.
Example 1: (MBRL for Atari)
This is for example done in paper where they concatenate 4 frames to predict the next frame and reward associated with it using UNet-like architecture.
Example 2: (World Models) by David Ha
Learn a latent space via Variational Autoencoder
This is the comparison of different dynamics models in terms of learning speed, inference

speed, require domain knowledge, and long-term accuracy.

Model desiderata

Name	Features	Speed of learning	Speed of predictions	Domain knowledge	Long-term accuracy
Dynamical system	States	Fast	Fast	High	High
MLP	States	Med	Fast	Low	Med
Observation	Observations	Slow	Slow	Low	Low
State-space models	Latent States	Slow	Fast	Low	Med

Model usage

How do we use a model?
There are two main approaches: background planning and online planning

Background planning

This also is divided into 2 categories: environment data augmentation/planning and sample-efficient policy learning

Environment data augmentation/planing:

Mix real and model-generated experience & apply traditional model-free methods(Q-learning, PGs). Models offer larger, augmented training datasets.
MBPO does policy gradient only on **simulated data**. Dyna-Q does Q-learning on **both** simulated and real data.

Dyna-Q:
Use collected data to learn a transition and reward model. Train a traditional RL algorithm (Q-learning) using **both** simulated data and real data that is collected.
Model-based Policy Optimization (MBPO)
Use collected data to learn transition-reward joint predictive model
Apply policy gradient methods **only** on **simulated model rollouts.** k rollouts
Take action in real environment.
Domain Randomization & Sim2Real
Learn a good policy on a "distribution of similar environments" by changing the dynamics of the simulation engine while training. The most remarkable example is from OpenAI's Rubik's cube challenge

Sample-efficient policy learning (assist learning algorithm)

The main idea here it train model and policy jointly end-to-end via backprop because aside from **imaginary experience**, models offer **derivatives**
Model-based backprop can be done either along **real** or **model-generated** trajectories
(Dream to Control by Hafner et al)
Goal is the same as before:

$$J(\theta) = \sum_{t=0}^{\inf} \gamma_t R(s_t, a_t)$$

with $a_t = \pi_\theta(s_t)$ and $s_{t+1} = T(s_t, a_t)$. And these policy and model are learned jointly by gradient ascent (like in policy gradients)
Remember we had our baseline REINFORCE that has high-variance and required stochastic policy
In this case, we can do more with **differentiable models**. It is possible to optimize the objective via end-to-end differentiation via backpropagation

Simple Algorithm:

1. Run a base policy(random policy) $\pi_0(a_t|s_t)$ to collect data samples $D(s, a, s')$
2. Learn a dynamics model $f_\theta(s, a)$ by minimizing MSE model prediction error
3. With a learned model, backpropagate through into policy to optimize a better policy $\pi_\theta(a_t|s_t)$
   Only **problem** here is that we have a **distribution mismatch**. The data we used to train the model and the new policy that we obtain have different distributions.
   Better algorithm:
4. Run a base policy(random policy) $\pi_0(a_t|s_t)$ to collect data samples $D(s, a, s')$
5. Learn a dynamics model $f_\theta(s, a)$
6. With a learned model, backpropagate through into policy to optimize a better policy $\pi_\theta(a_t|s_t)$
7. Run $\pi_\theta(a_t|s_t)$
8. Append visited tuples (s, a, s') to D.

In short, policy backprop allows to do :

1. Approximate transition and reward model with differentiable models
2. Calculate policy gradient via BPTT
   Advantages:

- Long-term credit assignment
- Sample efficiency with differentiable models
- Deterministic & no variance allowed
  Disadvantages:
- Vanishing and exploding gradient problem like in RNNs
- Prone to local minima

Background vs Decision-Time Planning

What's the difference?
Background planning: learn how to act for **any** situation. Optimization variables are parameters of policy, value)

$$J(\theta) = E_{s_0}[\sum_{t=0}^{H} \gamma^t r_t]$$

Online planning: find best sequences of actions for my **current** situation. Optimization variables are sequence of actions and/or states:

$$J(a_0, \ldots, a_H) = \sum_{t=0}^{H} \gamma^t r_t$$

This is an analogy to Daniel Kahneman's Thinking Fast and Slow paradigm (background=fast, decision-time planning slow)
Background planning allows these:

- act on partial observability
- fast computation at deployment
- predictability and coherence
- **same for discrete and continuous actions** (decision-time planning is different)

Decision-Time planning allows these:

- act on most recent world state
- act without learning
- competent in unfamiliar situations
- independent of observation space

So, the important thing for background planing is that there is not much distinction between discrete and continuous action spaces. For example, backprop is still possible with discrete actions via reparameterization.

(Online) Decision-Time Planning

However, we need specialized methods for decision-time planning when it comes to discrete vs continuous actions

Discrete actions

Monte Carlo Tree Search (MCTS):

1. Select: in nodes we have seen before, select action via Upper Confidence Bounds until we reach a leaf node where we don't know how to act next

2. Expand: we have access to transition model so we know where actions lets us end up in or we can do 1-step simulation to know successor states.
3. Simulate: this is where Monte Carlo kicks in. We randomly choose a new child node and play randomly until game finishes These Monte Carlo rollouts (it requires episodes!) gives use $\tilde{Q}(s,a)$, approximation of Q-values (lower bound).
4. Backup: backup Q-values and number of visits all the way up to the root of the tree.
   So far, we had two separate policies: selection policy π_s and random simulation policy π_r (for leaves onward where we don't know anything)
   Now, we can improve our selection policy π_s

How to **stop** MCTS rollouts? Until convergence starting from current real-world state or specify maximal computation time per MCTS run
We underestimate Q because of the random rollout policy π_r.

AlphaGo family

What's better in AlphaGo compared to usual MCTS?
Value Network: it helps to evaluate board positions to help **prune** the tree *depth.*
Policy Network: it helps to **select moves** to help prune the tree *breadth.*

AlphaGo:

Enhance MCTS with a learning component by having two networks: policy network that selects actions and value network that predicts win/lose for each position. The algorithm has large breadth and depth($b = 250$ and $d = 150$)

SL Training

Supervised policy network trained on expert moves that predicts **good actions** in states.

$$\Delta\sigma \sim \frac{\delta \log p_\sigma(a|s)}{\delta\sigma}$$

Input: randomly sampled state-action pairs from expert games. Output: a probability distribution over all legal moves a. **57%**

Rollout policy network has the same initialization and training but smaller and faster(less accurate) for fast inference. Accuracy **24%**

RL Training

RL Policy Network

All games are played against a **previous** random RL policy as opponent(self-play).
Goal is to maximize wins z_t by policy gradient RL:

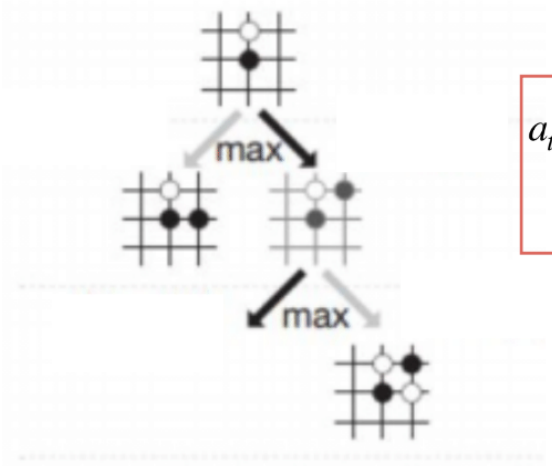
$$\Delta\rho \sim \frac{\delta \log p_\rho(a_t|s_t)}{\delta\rho} z_t$$

Value network is trained to approximate the value function under p_θ. Accuracy of the position evaluation of value function decreases as MC rollouts increase.
Goal is to minimize MSE on state-outcome pairs (s, z) using SGD:

$$\Delta\theta \sim \frac{\delta v_\theta(s)}{\delta\theta}(z - v_\theta(s))$$

RL for policy networks: initialize weights from SL network, but improve over SL policy by maximizing actions that lead to a win(policy gradients)
Finally, at **deployment**, combine the policy and value networks with MCTS at test time.

Selection: selecting actions within the expanded tree



$$a_t = \underset{a}{\text{argmax}} \underbrace{Q(s_t, a)}_{\text{exploitation}} + \underbrace{c\sqrt{N(s_t)}\frac{\pi_\theta(a_t \mid s_t)}{1 + N(s_t, a_t)}}_{\text{exploration}}$$

- a_t - action selected at time step t from state s_t
- $Q(s_r, a)$ - average reward collected so far from MC simulations
- $\pi_\theta(a \mid s)$ - prior expert probability provided by the SL policy p_σ
- $N(s, a)$ - number of times we have taken action a from state s during MC simulations

MCTS in AlphaGo

Selection

Selection

- In nodes we have visited before, we select which action to take based on the RL policy network action probabilities and the Q-approximation in the tree nodes, in an UCB-like process[1]
- Predictor + UCB applied to trees (PUCT) with policy network as predictor/prior:

$$\mathrm{a_t} = \mathrm{argmax_a} Q(s_t, a) + c_{puct} \cdot p_\rho(s)|_a \cdot \frac{\sqrt{\sum_b N(s,b)}}{1+N(s,\mathrm{a})} \qquad c_{puct} = 5$$

Expansion

Use SL Policy network to assign a prior action probability. The tree is expanded according to the most probable action of the RL Policy network.

Simulation

Multiple simulations are performed in parallel via the **the fast rollout** policy until the game finishes

Backup

Combine RL value network and final average reward from the rollouts. Q-values and visitation counts are updated in a similar fashion to classical MCTS

AlphaGo Zero:

It learns to play completely on its own, without human knowledge.
Main differences:

- Lookahead search during training!
- No supervised initialization from human games
- Joint RL policy and Value Network
- Self-play against **strongest** version of policy/value network

AlphaZero:

It masters three perfect information games using a single algorithm for all games
Augmented data and self-play is against current version not the previous strongest.

MuZero

It learns the rules of the game, allowing it to also master environments with unknown dynamics.
Learns three components:

1. H: function that transforms observations to latent states
2. G: latent transition model
3. F: function that predicts the current policy(from MCTS), and value of latent space.