Day 7. Model-based RL

NPEX Reinforcement Learning

2020.09.04

Jaeuk Shin

CORE
Control + Optimization Research Lab

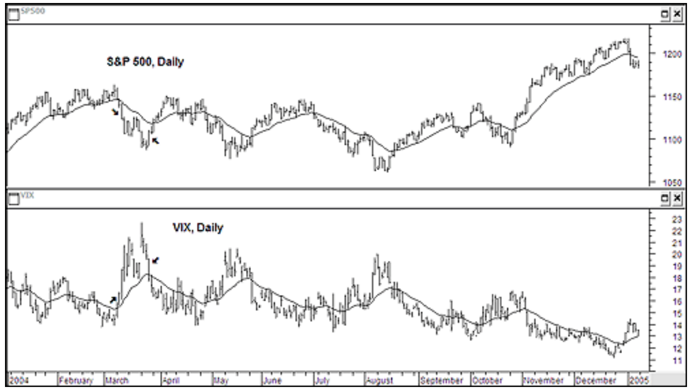
Model-based RL - Review

So far, we have learned **model-free** RL algorithms,
i.e., learning policy/value function were done without model info:

$$s_{t+1} \sim p(\cdot|s_t, a_t), \quad r_t = r(s_t, a_t).$$

What is p?

$\longrightarrow$ law of physics, artificial rules, etc.



Model-Based RL - Review

Model-based RL - Review

Can we **learn** p?

simplest case : $s_{t+1} = s_t + f(s_t, a_t)$

Assume we have a large number of transition samples (s_j, a_j, s'_j) from the **true** transition dynamics f.

Then, we may learn a parametrized model f_θ which is **close** to f.

How?

Model-based RL - Review

Given a batch $B = \{(s_j, a_j, s'_j)\}_{j=1}^{N}$, we construct a loss as follows:

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{j=1}^{N} \| s_{t+1} - s_t - f_\theta(s_t, a_t) \|^2$$

and update θ by gradient-based algorithms.

What can we do if we have a good model?

Model-based RL - Review

One can apply some well-known methods in control theory, such as **model predictive control(MPC)**:

$$\max_{A_t^{(H)}=(a_t,\cdots a_{t+H-1})} \sum_{t'=t}^{t+H-1} r(\hat{s}_{t'}, a_{t'})$$

$$where$$

$$\hat{s}_t = s_t, \quad \hat{s}_{t'+1} = \hat{s}_{t'} + f_\theta(\hat{s}_{t'}, a_{t'}), \quad t' = t, \cdots t+H-1.$$

In this case, we will use a simple algorithm so called **random sampling shooting method**.

CORE
Control + Optimization Research Lab

Model-based RL - Review

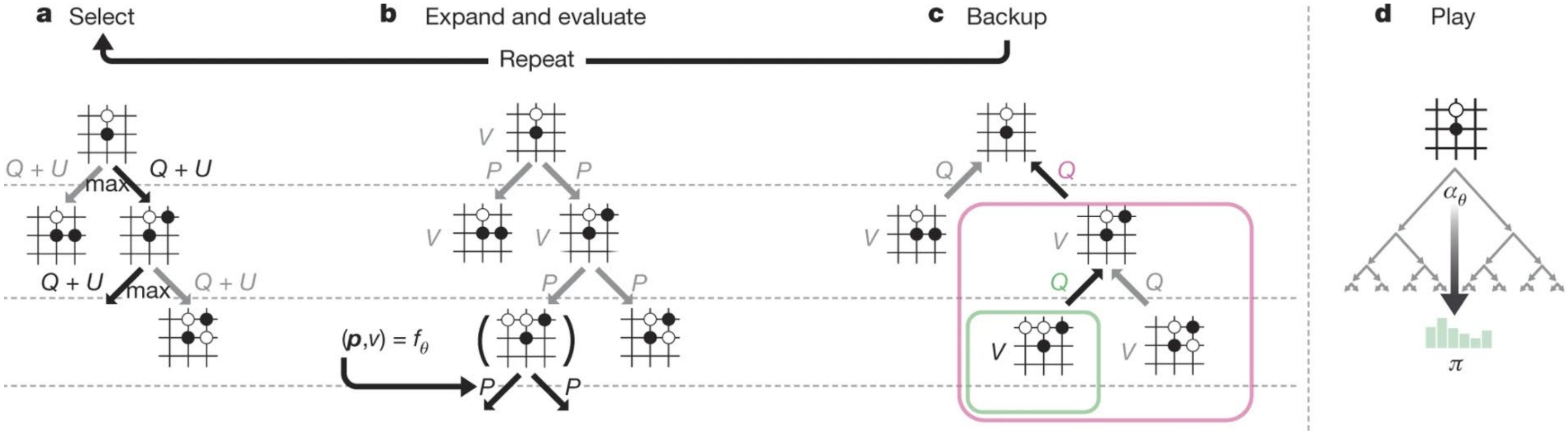
random sampling shooting is...

computational efficiency?

Model-based RL - Review

other options?

$\longrightarrow$ cross entropy method, tree search, etc.



Model-based RL - Review

Summary:

Algorithm 1 Model-based Reinforcement Learning

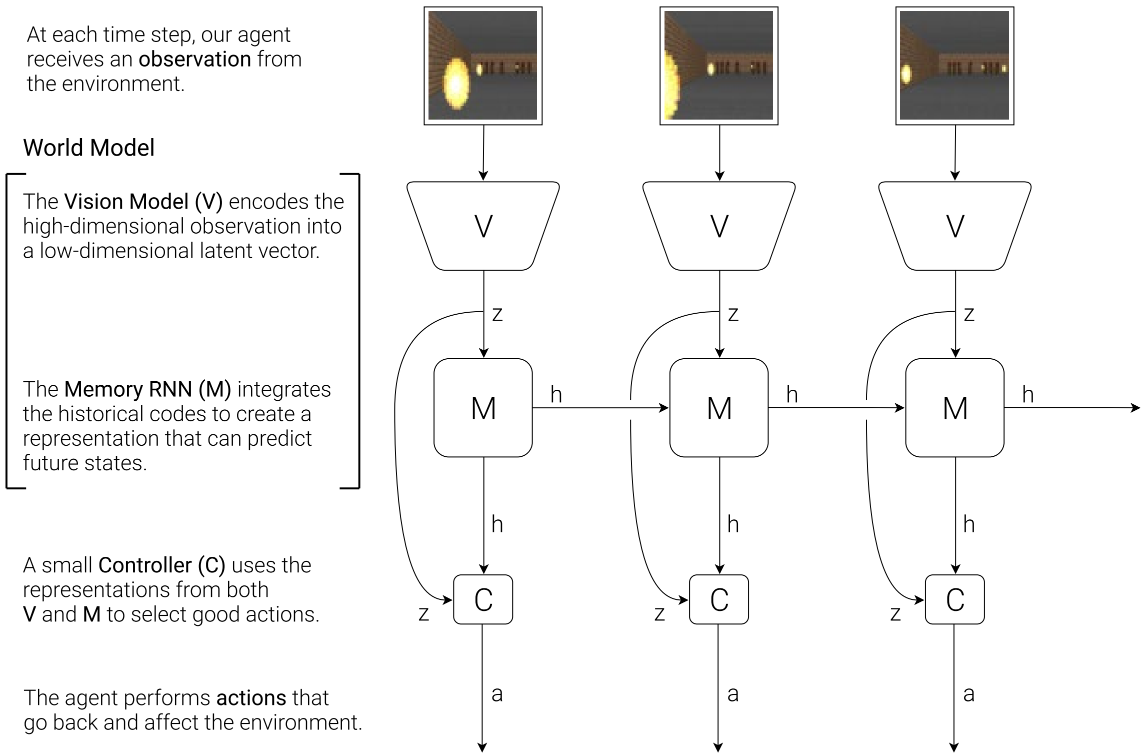
gather dataset $\mathcal{D}_{\text{RAND}}$ of random trajectories
initialize empty dataset $\mathcal{D}_{\text{RL}}$, and randomly initialize $\hat{f}_\theta$
for iter=1 **to** max_iter **do**
    train $\hat{f}_\theta(\mathbf{s}, \mathbf{a})$ by performing gradient descent on Eqn. 2, using $\mathcal{D}_{\text{RAND}}$ and $\mathcal{D}_{\text{RL}}$
    for $t = 1$ **to** T **do**
        get agent's current state $\mathbf{s}_t$
        use $\hat{f}_\theta$ to estimate optimal action sequence $\mathbf{A}_t^{(H)}$ (Eqn. 4)
        execute first action $\mathbf{a}_t$ from selected action sequence $\mathbf{A}_t^{(H)}$
        add $(\mathbf{s}_t, \mathbf{a}_t)$ to $\mathcal{D}_{\text{RL}}$
    end for
end for

so simple!

Model-based RL - Review

more advanced Model-based RL algorithms?

Ex) World Models (Ha, Schmidhuber, 2018)



https://worldmodels.github.io/

Model-based RL - Implementation

Model-based RL - Implementation

```python
class TransitionMemory:
    def __init__(self, state_dim, act_dim):
        # shape arguments must be tuple!
        self.data = []
        self.state_dim = state_dim
        self.act_dim = act_dim

    def append(self, state, act, next_state):
        self.data.append((state, act, next_state))

    def sample_batch(self, size):
        # uniform sampling
        # prepare batch containers
        state_batch = np.zeros((size, self.state_dim))
        act_batch = np.zeros((size, self.act_dim))
        next_state_batch = np.zeros((size, self.state_dim))

        num_data = len(self.data)
        rng = np.random.default_rng()
        idxs = rng.choice(num_data, size)
```

transition only

Model-based RL - Implementation

```python
class TransitionModel(nn.Module):
    def __init__(self, state_dim, act_dim, hidden1, hidden2):
        super(TransitionModel, self).__init__()
        self.state_dim = state_dim
        self.act_dim = act_dim
        self.fc1 = nn.Linear(state_dim + act_dim, hidden1)
        self.fc2 = nn.Linear(hidden1, hidden2)
        self.fc3 = nn.Linear(hidden2, state_dim)

    def forward(self, state, act):
        x = torch.cat([state, act], dim=1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        delta = self.fc3(x)

        next_state = state + delta  # \hat{s}_{t+1} = s_t + f(s_t, a_t ; \theta)

        return next_state
```

CORE
Control + Optimization Research Lab

Model-based RL - Implementation

```
def train(self, batch_size):
    self.model.train()
    # note that training of the dynamics does not depend on any reward info
    (state_batch, act_batch, next_state_batch) = self.memory.sample_batch(batch_size)

    state_batch = torch.tensor(state_batch).float()
    act_batch = torch.tensor(act_batch).float()
    next_state_batch = torch.tensor(next_state_batch).float()

    prediction = self.model(state_batch, act_batch)


    loss_ftn = MSELoss()
    loss = loss_ftn(prediction, next_state_batch)

    self.optimizer.zero_grad()
    loss.backward()
    self.optimizer.step()

    loss_val = loss.detach().numpy()

    return loss_val
```

Remark. This is a supervised learning, so we may try to measure validation error!

Model-based RL - Implementation

```python
def execute_action(self, state, rew_ftn, K, H):
    """
    # generate K trajectories using the model of dynamics and random action sampling, and perform MPC
    # Remark! K roll-outs can be done simultaneously!

    given a state, execute an action based on random-sampling shooting method
    :param state: current state(numpy array)
    :param rew_ftn: vectorized reward function
    :param K: number of candidate action sequences to generate
    :param H: length of time horizon
    :return: action to be executed(numpy array)
    """
    assert K > 0 and H > 0

    dimA = self.dimA

    self.model.eval()

    states = np.tile(state, (K, 1)) # shape = (K, dim S)
    scores = np.zeros(K)     # array which contains cumulative rewards of roll-outs

    # generate K random action sequences of length H
    action_sequences = self.ctrl_range * (2. * np.random.rand(H, K, dimA) - 1.)
    first_actions = action_sequences[0]     # shape = (K, dim A)
```

so far so good, but how can we generate a large number of trajectories efficiently?

CORE
Control + Optimization Research Lab

Model-based RL - Implementation

```
for t in range(H):
    actions = action_sequences[t]    # set of K actions, shape = (K, dim A)
    scores += rew_ftn(states, actions)

    s = torch.tensor(states).float()
    a = torch.tensor(actions).float()

    next_s = self.model(s, a)

    # torch tensor to numpy array
    # this cannot be skipped since a reward function takes numpy arrays as its inputs
    states = next_s.detach().numpy()

best_seq = np.argmax(scores)

return action_sequences[0, best_seq]
```

answer : vectorization!

Still, MPC is a bottleneck of the whole process...

Model-based RL - Experiment

TRPO – Complete Implementation

TRPO – Complete Implementation

several components :
conjugate gradient algorithm(to solve $A \cdot s = g$),
and backtracking line search

```
###############
# train actor #
###############

log_probs = self.pi.log_prob(states, actions)

# \pi(a_t | s_t ; \phi) / \pi(a_t | s_t ; \phi_old)
prob_ratio = torch.exp(log_probs - old_log_probs)

actor_loss = torch.mean(prob_ratio * A)
loss_grad = torch.autograd.grad(actor_loss, self.pi.parameters())
# flatten gradients of params
g = torch.cat([grad.view(-1) for grad in loss_grad]).data

s = cg(fisher_vector_product, g, self.pi, states)

sAs = torch.sum(fisher_vector_product(s, self.pi, states) * s, dim=0, keepdim=True)
step_size = torch.sqrt(2 * self.delta / sAs)[0]
step = step_size * s


old_actor = Actor(self.dimS, self.dimA, self.hidden1, self.hidden2)

old_actor.load_state_dict(self.pi.state_dict())

params = flat_params(self.pi)

backtracking_line_search(old_actor,
```

TRPO – Complete Implementation

```python
def fisher_vector_product(v, actor, obs_batch, cg_damping=1e-2):
    # efficient Hessian-vector product
    # in our implementation, Hessian just corresponds to Fisher information matrix I
    v.detach()
    kl = torch.mean(kl_div(actor=actor, old_actor=actor, obs_batch=obs_batch))

    kl_grads = torch.autograd.grad(kl, actor.parameters(), create_graph=True)
    kl_grad = torch.cat([grad.view(-1) for grad in kl_grads])

    kl_grad_p = torch.sum(kl_grad * v)
    Iv = torch.autograd.grad(kl_grad_p, actor.parameters()) # product of Fisher information I and v
    Iv = flatten(Iv)

    return Iv + v * cg_damping
```

How to compute Hessian of f in PyTorch?

Answer : Hessian - vector product

CORE
Control + Optimization Research Lab

Thank you