Control with Reinforcement Learning

Contents

Comparison of RL and Supervised Learning

Brief introductions of terms and background

Parameter Optimization for Policy (using PI^2)

Skipping Parameter Optimization with MPC

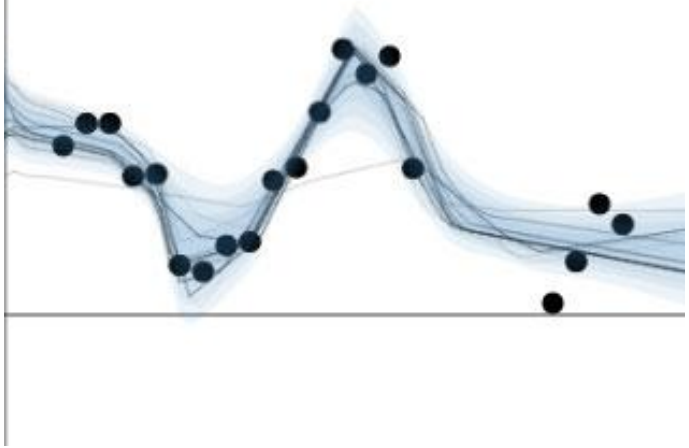
Limits of Supervised Learning (for Robot Control)

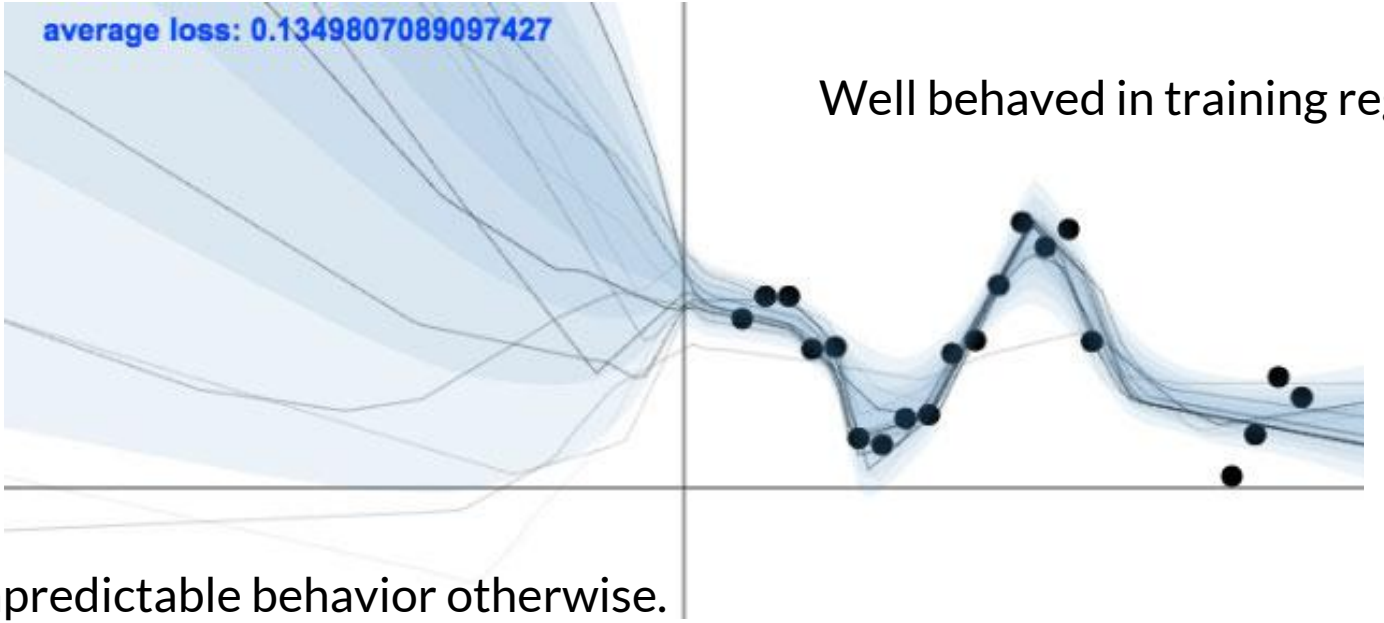
Assumes training data *perfectly represents* the variety of data experienced at run-time.

Does not have mechanism to incorporate new data outset of training.

Will probably have unexpected / dangerous behavior when operating out of spec.

Well behaved in training regime...





Well behaved in training regime

Unpredictable behavior otherwise.

http://mlg.eng.cam.ac.uk/yarin/blog_2248.html

How does RL help?

The learner is an *agent*, that takes actions to achieve a high reward / low cost.

Actions can take the agent to new states (new data), that will be incorporated into the agent's behavior.

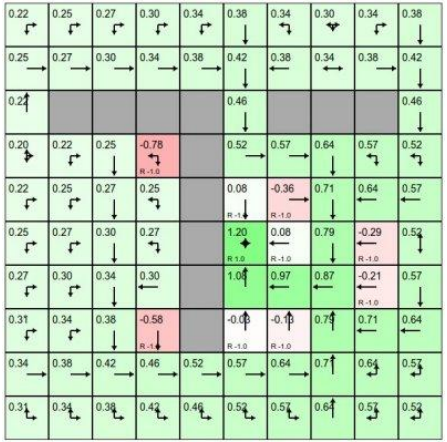
RL can be viewed as a superset of Supervised Learning.

Some Terms

Trajectory : A sequence of states and actions, time indexed, where the state at some time *t*, depends on the state and action taken at time *t-1*.

Cost function : a function that takes in a state-action pair, and returns a value denoting ‘good’, ‘bad’, or how well that pair accomplishes some goal.

Some Terms in RL



V(s) : Value function. Represents the *value* of being in some state ‘x’ relative to the current task.

$$V(x_t) = R(x_t) + V(x_{t+1})$$

Q(s,a) : Q-function. Represents the *value* of being in a state ‘x’ *and* taking an action ‘u’.

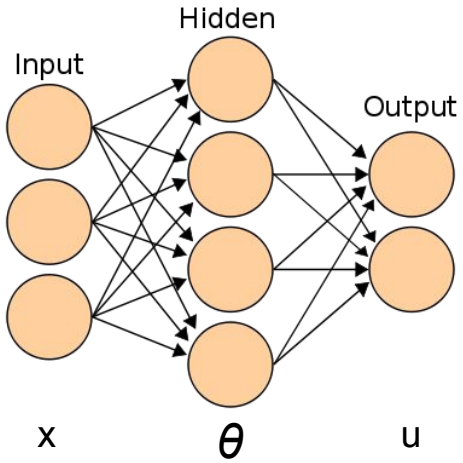
$$Q(x_t,u_t) = R(x_t) + Q(x_{t+1},\pi(x_{t+1}))$$

$a=\pi(s)$: Policy function. Returns an action for a given state. The optimal policy is the highest scoring action in the Q-function, given an optimal Q function.

https://cs.stanford.edu/people/karpathy/reinforcejs/

Function Approx. of Policy

$u=\pi_\theta(x)$: here we say policy π is parameterized by θ.



Simple Policy Improvement

Cross-Entropy Method (one iteration)

$$\boldsymbol{\theta}_{k=1\dots K} \sim \mathcal{N}(\boldsymbol{\theta}, \Sigma) \qquad \textbf{sample} \quad (1)$$

$$J_k = J(\boldsymbol{\theta}_k) \qquad \textbf{eval.} \quad (2)$$

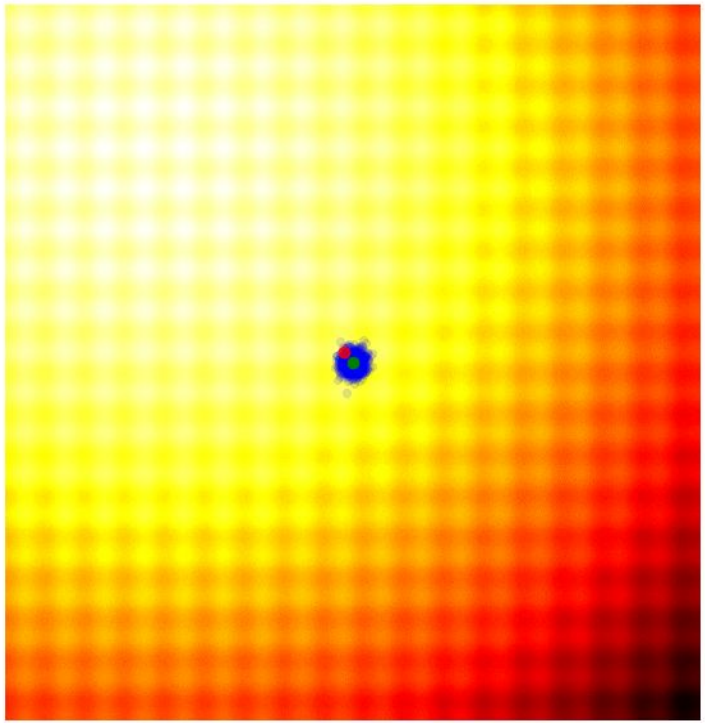
$$\boldsymbol{\theta}_{k=1\dots K} \leftarrow \texttt{sort}\ \boldsymbol{\theta}_{k=1\dots K}\ \texttt{w.r.t}\ J_{k=1\dots K} \qquad \textbf{sort} \quad (3)$$

$$\boldsymbol{\theta}^{new} = \sum_{k=1}^{K_e} \frac{1}{K_e} \boldsymbol{\theta}_k \qquad \textbf{update} \quad (4)$$

$$\Sigma^{new} = \sum_{k=1}^{K_e} \frac{1}{K_e} (\boldsymbol{\theta}_k - \boldsymbol{\theta})(\boldsymbol{\theta}_k - \boldsymbol{\theta})^\intercal \qquad \textbf{update} \quad (5)$$

1. Sample *K* policies from a distribution.
2. Evaluate the *1-step score* of each policy with cost function *J(θ)*
3. Rank each policy θ_k based on its score
4. Make a new policy; here we average the K_e best performing policies.
5. Can also update other parameters with this scoring.

CMA-ES



http://blog.otoro.net/2017/10/29/visual-evolution-strategies/

PI²

1. Sample *K* policies from a distribution.
2. *Rollout* each policy for *N* timesteps, and store states and actions.
3. Evaluate the *performance* of each policy with cost function *J(θ)*
4. Weight each trajectory's score against all others, and use them to make a weighted average of our *K* policies.

PI^2

for $k = 1 \dots K$ **do**

| $\boldsymbol{\theta}_{k,i=1\dots N} \sim \mathcal{N}(\boldsymbol{\theta}, \Sigma)$

| $\tau_{k,i=1\dots N} = \text{executepolicy}(\boldsymbol{\theta}_{k,i=1\dots N})$

for $i = 1 \dots N$ **do**

| for $k = 1 \dots K$ **do**

| | $S_{k,i} \equiv S(\boldsymbol{\tau}_{k,i}) = \sum_{j=i}^{N} J(\boldsymbol{\tau}_{j,k})$

| | $P_{k,i} = \frac{e^{-\frac{1}{\lambda} S_{k,i}}}{\sum_{k=1}^{K} [e^{-\frac{1}{\lambda} S_{k,i}}]}$

| $\boldsymbol{\theta}_i^{new} = \sum_{k=1}^{K} P_{k,i} \boldsymbol{\theta}_k$

| $\Sigma_i^{new} = \sum_{k=1}^{K} P_k (\boldsymbol{\theta}_{k,i} - \boldsymbol{\theta})(\boldsymbol{\theta}_{k,i} - \boldsymbol{\theta})^\intercal$

$\boldsymbol{\theta}^{new} = \frac{\sum_{i=0}^{N} (N-i) \boldsymbol{\theta}_i^{new}}{\sum_{l=0}^{N} (N-l)}$

$\Sigma^{new} = \frac{\sum_{i=0}^{N} (N-i) \Sigma_i^{new}}{\sum_{l=0}^{N} (N-l)}$

Limitations of PI²

K needs to be somewhat large; takes a lot of compute to do many rollouts.

θ may also have many parameters, contributing to compute issues.

Trajectory optimization is a hard problem; difficult to find a path from start to finish effectively.

Do we even need a policy?

MPPI

MPC : Instead of a long, perfect trajectory, we constantly recalculate short trajectories whilst simultaneously using the results. Can make up for modelling errors, run-time perturbations, etc.

Given that a policy & model produces a sequence of controls; can we just optimize a sequence of controls?

```
for t=1:T
    u_t=π(x_t)
    x_{t+1}=F(x_t,u_t)
end
```

Algorithm 2: MPPI

Given: $\mathbf{F}$: Transition Model;
K: Number of samples;
T: Number of timesteps;
$(\mathbf{u}_0, \mathbf{u}_1, ...\mathbf{u}_{T-1})$: Initial control sequence;
Σ, ϕ, q, λ: Control hyper-parameters;
while *task not completed* **do**
  $\mathbf{x}_0 \leftarrow$ GetStateEstimate();
  for $k \leftarrow 0$ **to** $K-1$ **do**
    $\mathbf{x} \leftarrow \mathbf{x}_0$;
    Sample $\mathcal{E}^k = \{\epsilon_0^k, \epsilon_1^k, \dots \epsilon_{T-1}^k\}$;
    for $t \leftarrow 1$ **to** T **do**
      $\mathbf{x}_t \leftarrow \mathbf{F}(\mathbf{x}_{t-1}, \mathbf{u}_{t-1} + \epsilon_{t-1}^k)$;
      $S(\mathcal{E}^k) \mathrel{+}= \mathbf{q}(\mathbf{x}_t) + \lambda \mathbf{u}_{t-1}^{\mathrm{T}} \Sigma^{-1} \epsilon_{t-1}^k$;
    $S(\mathcal{E}^k) \mathrel{+}= \phi(\mathbf{x}_T)$;
  $\beta \leftarrow \min_k [S(\mathcal{E}^k)]$;
  $\eta \leftarrow \sum_{k=0}^{K-1} \exp\left(-\frac{1}{\lambda}(S(\mathcal{E}^k) - \beta)\right)$;
  for $k \leftarrow 0$ **to** $K-1$ **do**
    $w(\mathcal{E}^k) \leftarrow \frac{1}{\eta} \exp\left(-\frac{1}{\lambda}(S(\mathcal{E}^k) - \beta)\right)$;
  for $t \leftarrow 0$ **to** $T-1$ **do**
    $\mathbf{u}_t \mathrel{+}= \sum_{k=1}^{K} w(\mathcal{E}^k) \epsilon_t^k$;
  SendToActuators($\mathbf{u}_0$);
  for $t \leftarrow 1$ **to** $T-1$ **do**
    $\mathbf{u}_{t-1} \leftarrow \mathbf{u}_t$;
  $\mathbf{u}_{T-1} \leftarrow$ Intialize($\mathbf{u}_{T-1}$);

Instead of a policy with parameters θ, we instead just keep a sequence of controls $U=[u_0,u_1,u_2...]$; instead of θ_k we have $U_k=[u_0+\epsilon^k_0, u_1+\epsilon^k_1, u_2+\epsilon^k_2...]$.

We rollout each U_k and score each trajectory. Score is $S(\epsilon^k)$, and cost function is $q()$.

The weighting is the same as PI^2, with a baseline subtraction to scale things.

The weights give us a new U instead of new policy.

Implementation Details

Task: Go to Pose. When close enough, consider task done.

Avoid for-loops. Use PyTorch's Tensor math. Look up 'mm', 'bmm', and others…

Chance are you will need to smooth your control outputs; you can smooth before you update, or smooth the values sent to the actuators. When adding noise to controls, you should clip to the values your NN model was trained at.

T=30; K=1000 at 10hz is easily achievable.

Test.

Debug each component in parts. Does your NN correctly ‘drive’ forward? Backwards? Left and right? Sitting still?

Does each part of MPPI do the expected thing? Set T and K to small values, and look at all the numbers. Convince yourself it’s doing the correct thing.

Useful References

http://blog.otoro.net/2017/10/29/visual-evolution-strategies/ GIFs

https://icml.cc/2012/papers/171.pdf PI^2 with CMA

https://www.cc.gatech.edu/~bboots3/files/InformationTheoreticMPC.pdf

https://arxiv.org/abs/1707.04540 newest MPPI work (has some tips)