Classification-based reinforcement learning

Vincent Antaki

McGill University

Classification-based reinforcement learning

Approach based on :

- Lagoudakis & Parr, *Reinforcement Learning as Classification : Leverating Modern Classifiers*
- Riedmiller, *Neural Fitted Q Iteration - First Experiences with a Data Efficient Neural Reinforcement Learning Method*
- Farahmand, Precup, Barreto, Ghavamzadeh *Classification-based Approximate Policy Iteration : Experiments and Extended Discussions*
- Lagoudakis & Parr, *Least-Squares Policy Iteration*

Classification-based reinforcement learning

Why classification-based reinforcement learning?

- Attempt to leverage advantages of supervised learning for reinforcement learning problems (ex. data efficiency, handling of non-linearity)
- Find structure directly in the action space

Policy iteration

```
>>> While not satisfied with the policy :
...         policy_eval()
...         policy_update()
```

Figure: The underlying principles behind policy iteration

CAPI

Algorithm CAPI(Π, ν, K)

Input: Policy space Π, State distribution ν, Number of iterations K

Initialize: Let $\pi_{(0)} \in \Pi$ be an arbitrary policy

for $k = 0, 1, \ldots, K-1$ **do**

Construct a dataset $\mathcal{D}_n^{(k)} = \{X_i\}_{i=1}^n,\ X_i \overset{\text{i.i.d.}}{\sim} \nu$

$\hat{Q}^{\pi_k} \leftarrow \text{PolicyEval}(\pi_k)$

$\pi_{k+1} \leftarrow \text{argmin}_{\pi \in \Pi} \hat{L}_n^{\pi_k}(\pi)$ (action-gap-weighted classification)

end for

Figure: The CAPI framework

* Image from Farahmand, Precup, Barreto, Ghavamzadeh *Classification-based Approximate Policy Iteration : Experiments and Extended Discussions*

CAPI

Action gap :

- Given state X_i and an action a, we consider the absolute difference between $\hat{Q}^{\pi_*}(X_i, a)$ and $\hat{Q}^{\pi_*}(X_i, a^*)$
- When action gap is low, regret for choosing the non-optimal action is low and confusion is more likely to happen.

Very important :

$$\hat{L}_n^{\pi_k}(\pi) = \sum_{X_i \in \mathcal{D}_n^{(k)}} \mathbf{g}_{\hat{Q}^{\pi_k}}(X_i) \mathbb{I}\{\pi(X_i) \neq \operatorname*{argmax}_{a \in \mathcal{A}} \hat{Q}^{\pi_k}(X_i, a)\}$$

Policy evalution

We want to approximate π^* by learning Q-values from our samples. Our options include :

- Rollout
- Least-square Temporal Difference Q-learning
- Neural Fitted Q iteration

Rollout

Rollout $(\mathcal{M},\ s,\ a,\ \gamma,\ \pi,\ K,\ T)$

// $\mathcal{M}$: Generative model
// (s, a) : State-action pair whose value is sought
// γ : Discount factor
// π : Policy
// K : Number of trajectories
// T : Length of each trajectory

for $k = 1$ **to** K
  $(s', r) \leftarrow \text{SIMULATE}(\mathcal{M}, s, a)$
  $\widetilde{Q}_k \leftarrow r$
  $s \leftarrow s'$
  for $t = 1$ **to** $T - 1$
    $(s', r) \leftarrow \text{SIMULATE}(\mathcal{M}, s, \pi(s))$
    $\widetilde{Q}_k \leftarrow \widetilde{Q}_k + \gamma^t r$
    $s \leftarrow s'$

$$\widetilde{Q} \leftarrow \frac{1}{K} \sum_{k=1}^{K} \widetilde{Q}_k$$

return $\widetilde{Q}$

Figure: The rollout algorithm

- Simulation-based approximation.

* Image from Lagoudakis & Parr, *Reinforcement Learning as Classification : Leverating Modern Classifiers*

LSTDQ

LSTDQ $(D,\ k,\ \phi,\ \gamma,\ \pi)$ // Learns $\widehat{Q}^{\pi}$ from samples

// D : Source of samples (s, a, r, s')
// k : Number of basis functions
// ϕ : Basis functions
// γ : Discount factor
// π : Policy whose value function is sought

$\widetilde{\mathbf{A}} \leftarrow \mathbf{0}$ // $(k \times k)$ matrix
$\widetilde{b} \leftarrow \mathbf{0}$ // $(k \times 1)$ vector

for each $(s, a, r, s') \in D$
$\widetilde{\mathbf{A}} \leftarrow \widetilde{\mathbf{A}} + \phi(s, a)\Big(\phi(s, a) - \gamma\phi\big(s', \pi(s')\big)\Big)^{\mathsf{T}}$
$\widetilde{b} \leftarrow \widetilde{b} + \phi(s, a)r$

$\widetilde{w}^{\pi} \leftarrow \widetilde{\mathbf{A}}^{-1}\widetilde{b}$

return $\widetilde{w}^{\pi}$

Figure: The LSTDQ algorithm

- Linear function approximation.
- Need to use basis functions.
- Requires pseudo-matrix inversions.

* Image from Lagoudakis & Parr, *Least-Squares Policy Iteration*

NFQ

NFQ_main() {
input: a set of transition samples D; output: Q-value function Q_N
k=0
init_MLP() $\rightarrow Q_0$;
Do {
generate_pattern_set $P = \{(input^l, target^l), l = 1, \ldots, \#D\}$ where:
$input^l = s^l, u^l,$
$target^l = c(s^l, u^l, s'^l) + \gamma \, min_b Q_k(s'^l, b)$
Rprop_training$(P) \rightarrow Q_{k+1}$
k:= k+1
} WHILE $(k < N)$

- Use a neural net for regression
- Trained with SGD or an RProp variant.
- If using RProp, train in big batch to avoid frequent gradient sign change

* Image from Riedmiller, *Neural Fitted Q Iteration - First Experiences with a Data Efficient Neural Reinforcement Learning Method*

Policy update

Using our approximation of π^* and our samples, we generate multiple examples from each seen state.

- We give label 0 to for pairs (s, a^*)
- We give label 1 to for pairs (s, a) $\forall a \neq a^*$

Then, we train a classifier on the dataset, set a tie-breaker policy and we have a new policy.

Technical considerations

We should technically end our algorithm when the policy converges or when the preset maximum number of iterations is reached.

- Threshold on empirical similarity between policies as stopping criterion.

Having very different policy evaluation and policy update methods can induce policy instability and other wierd behaviors.

Advantages

What are the possibles advantages to use a classifier for the policy update?

- Data-efficient methods.
- Lots of option to handle non-linearity.
- Can detect structure inherent to the action space.

Advantages

What are the possibles advantages to use a classifier for the policy update?

- Data-efficient methods.
- Lots of option to handle non-linearity.
- Can detect structure inherent to the action space.
- We can make our implementation Scikit compatible.

Load balancing problem

- The agent has 2 'servers' at its disposition and needs to dispatch them tasks it receives. Tasks arrive randomly following a Poisson distribution with $\lambda = 2$.
- Tasks requires a certain amount of work to be completed. This amount of work required to complete a task is equal to $1 + T$ where T is a random poisson variable with $\lambda = 3$. The agent never knows the associated workload with a task.
- All server queues have a maximum length of 10. If the agent tries to add a task to an already full queue, the task is discarded and the agents receive a -5 points reward.
- At every timestep, all servers accomplish one unit of work on the first task in their queue.
- Upon the completion a task by a server, the agent receives a reward equal to $\frac{5}{\#\text{ of iteration to complete task}}$.

Considerations

For simplicity, we consider the state to be defined as follows :

- Current timestep
- Queue length of all servers
- Time since the current task has been added to the queue for all servers
- Number of timesteps spent on current task for all servers.

Considerations

Difficulties inherent to that problem :

- Quite delayed reward
- A bit of stochasticity
- Partially observable state
- Decision unevenly spread through time
- Better formulated as an average reward problem?

Difficulties inherent to our approaches:

- NFQ : Hyperparameters tuning required.
- LSTDQ : Need to find the appropriate set of basis function. Does not scale well.

Considerations

The optimal policy is however very simple.

- Give the task to the server which is expected to complete it the soonest.

For the first variant, this means :

- Give the task to the server with the shortest queue.
- If multiples servers have the shortest length of queue, give the task to the one which been running its task for the longest amount of time.

Methodology and other technical considerations

- We terminate simulation after a maximum of 500 decisions.
- We generate batches of 50 episodes for iteration of the main loop (with $\epsilon = 0.2$).
- Examples are discarded after each policy update.
- We monitor the reward for each episode and the number of time the agent tries to add a task to an already filled queue.
- We use a onehot encoding of the action when learning the classifier

Approach

We consider four settings :

- Policy eval with NFQ using 2 layers MLP with relu activation, no activation on the output neuron, 100 neurons by hidden layer, L1 and L2 regularisation. Decision Tree for policy update.
- Policy eval with NFQ, policy update with a similar MLP.
- Policy eval with NFQ, policy update with logistic regression.
- Policy eval with NFQ, policy update with KNN and $k = 5$.

Results

To quickly summarize results :

- None worked. All performs significantly worst than the random agent.
- Also tried policy iteration with only NFQ. Did not work either.

Possible failure causes :

- Insufficiant tuning of hyperparameters (mostly tried to do it by hand).
- Discrepancy between policy evaluation model and policy update model?
- Full batch learning and RProp+ may be good for stability, but they may hinder speed of convergence.
- Insufficiant exploration (i.e ϵ too low) ?

Approaches

Possibles add-on to our approach :

- Shared structure between policy eval and policy update MLP.
- Keep, and update, a subset of examples through iterations (experience replay?).

Alternatives to consider :

- Consider alternative policy evaluation methods. For instance, try to use regression tree in the same fashion as we the MLP in NFQ.
- Critic-based approach

Gabillon, Lazaric, Ghavamzadeh and Scherrer.
Classification-based Policy Iteration with a Critic (2011)
https://hal.inria.fr/hal-00590972/PDF/dpi-critic-techReport.pdf

Linewalk

- Maybe the previous environment was a bit hard. Let's try our method on a very simple environment.
- We'll use the linewalk environment with 10 states, a default reward of -1 per timestep, a 100 points terminal reward and a -2 reward for hitting the wall.

We use the following settings :

- 5 episodes for every policy update
- Maximum of 50 iteration by episodes.
- Behavior policy is ϵ-greedy with $\epsilon = 0.2$
- $\gamma = 0.99$
- We use a single-layer percreptron instead of a MLP for the NFQ algorithm.

Results

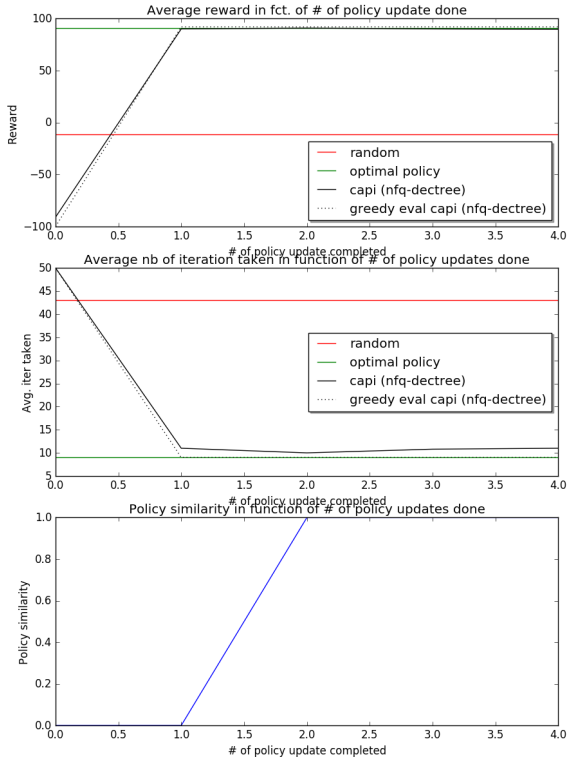

Figure: It's super effective!

- All variants are able to learn the optimal policy in one update.
- Linewalk is a very easy environment.

The End

Thank you!