A K-Fold method for Baseline estimation in Policy Gradient algorithms

Sunil Srinivasa

Nithyanand Kota

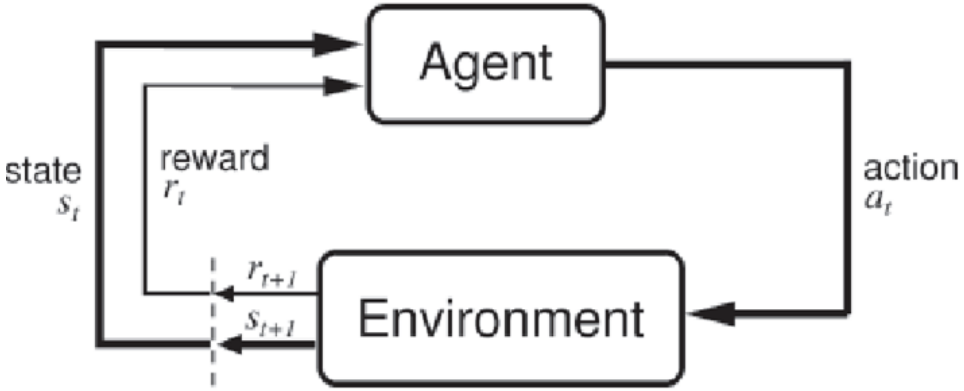
Abhishek Mishra

Presented at 2016 NIPS Deep RL Workshop

AGENDA

- What is Reinforcement Learning?
- What are policy gradient algorithms?
- What is a Baseline?
- Current Baseline estimation issues.
- K-Fold method for Baseline estimation.
- Experimental Results.

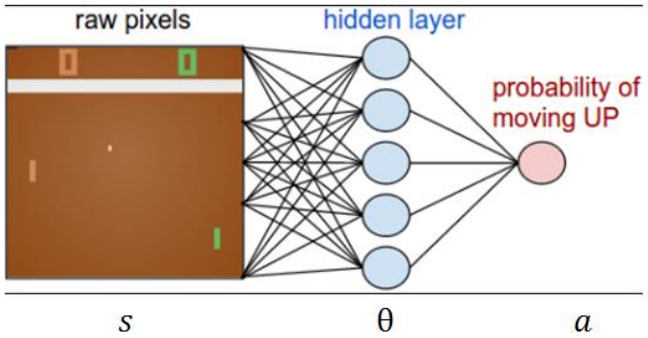
Reinforcement Learning (RL)



- **Agent** learns from repeated interactions with **environment**.
- The goal is to determine the set of actions that maximize cumulative reward (also called **return**).
- Has seen tremendous practical success of late:
  - Autonomous vehicles
  - Google datacenter PUE
  - Robotic control
  - Alpha-go, ATARI games

Policy Gradient Algorithms

- The agent's behavior is called **policy**, which is a mapping from state to action. Typically, the policy is represented as a **deep network** parameterized by θ.



- The **expected return** $E_{s,a}[R_\theta(s,a)]$ is notated simply as $J(\theta)$, which needs to be **maximized**.

- PG algorithms simply use the **gradient of the return** $\nabla_\theta J(\theta)$ to optimize the policy parameters θ directly.

Policy Gradient Algorithms (Contd.)

- The **policy gradient** $\nabla_\theta J(\theta) = \nabla_\theta E_{s,a}[R_\theta(s,a)]$

- Policy gradient algorithms search for a **local maximum** in $J(\theta)$ by **ascending the gradient of the policy** w.r.t the parameters θ **iteratively**, for e.g., in Vanilla Policy Gradient (VPG):
  - $\theta_{k+1} = \theta_k + \alpha\nabla_\theta J(\theta)$, where α is a step-size parameter.

- The PG $\nabla_\theta J(\theta)$ is estimated in practice via **Monte Carlo** (MC) methods.
  - Though the MC estimates are unbiased, they suffer from **high variance**.
  - This means the convergence time of the algorithm becomes very large.

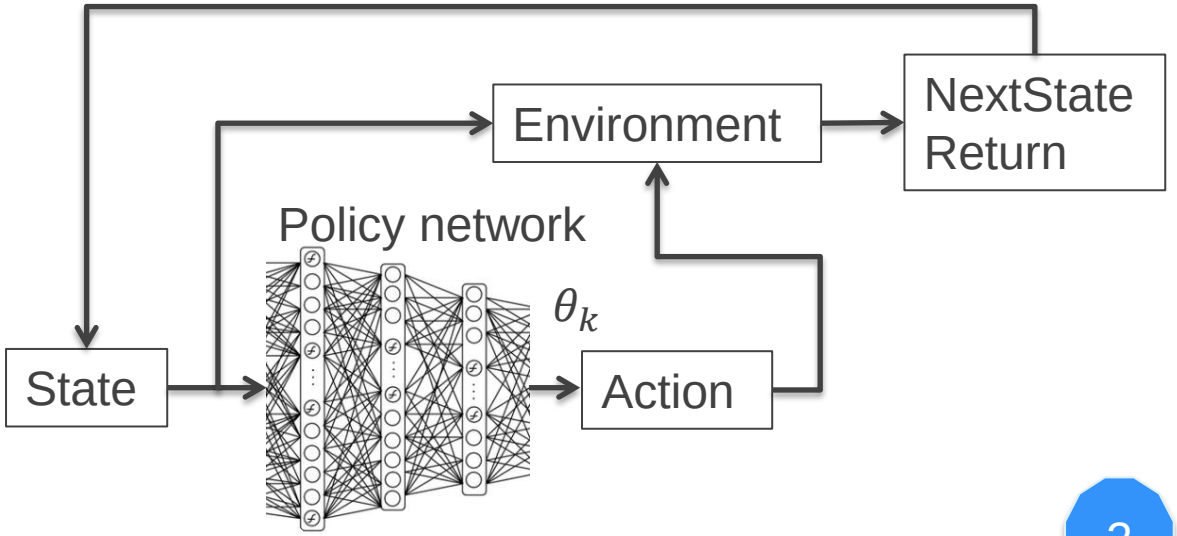
Baseline

- In order to reduce the variance in the PG estimates, a **baseline b** is added.
  - $\nabla_\theta J(\theta) = \nabla_\theta J(\theta) = \nabla_\theta E_{s,a}[R_\theta(s,a) - b(s)]$

- Typically, $b(s) = V(s)$, the **state-value function**, which is the expected return starting from state s.
  - This is *somewhat* optimal in terms of reducing the variance of PG.

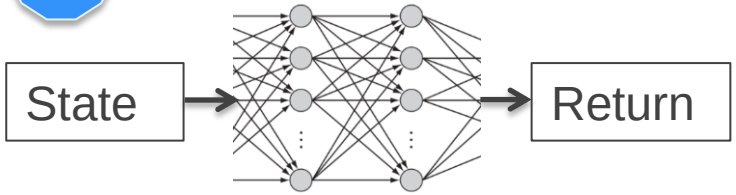
- In practice, the baseline is estimated using a regression on (fitting) states versus the returns obtained starting from those states.

What happens in each iteration?

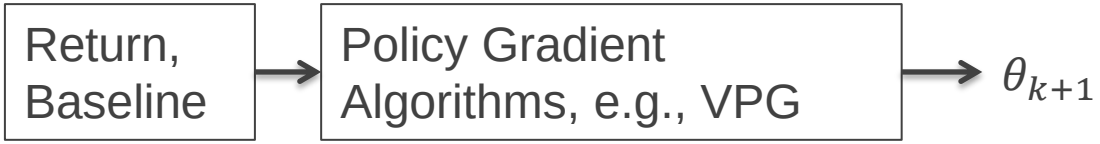


1 Collect a lot of (state, return) data

2 Train a Baseline network

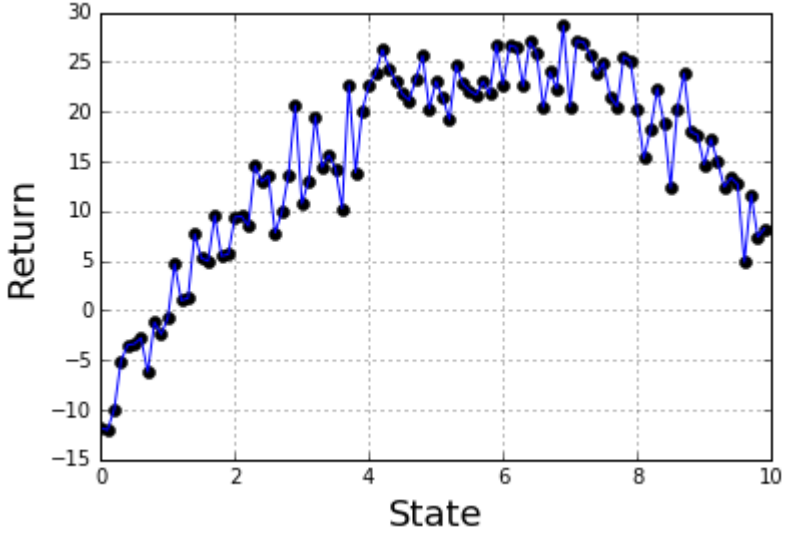


3 Update the policy parameters



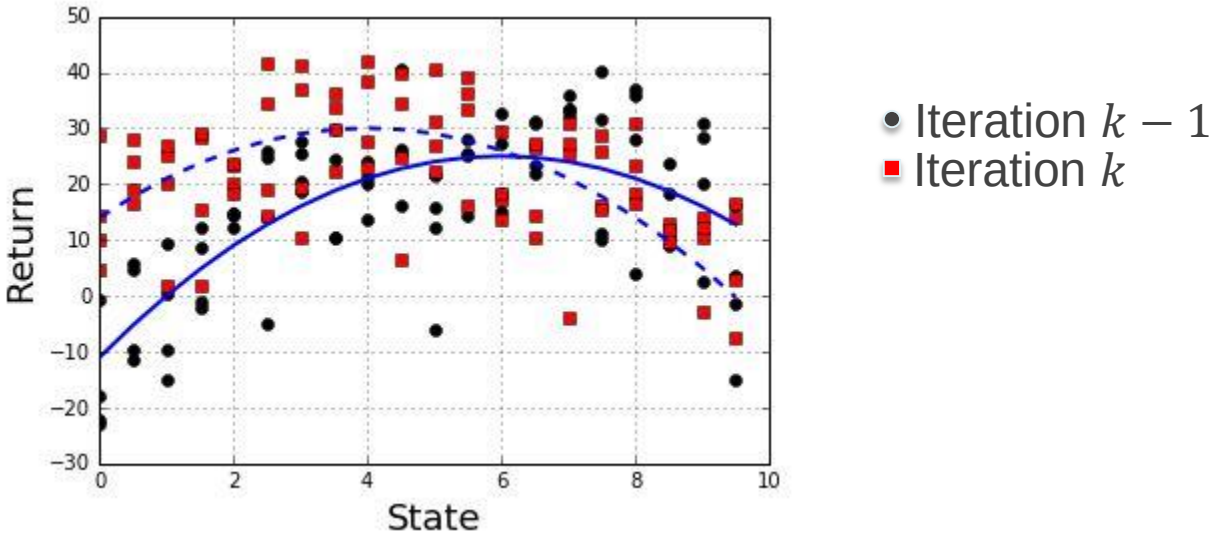
Case 1: Baseline fitting is performed in iteration k and the baseline is used for computing PG in iteration k

- This causes the baseline fit to become **biased** towards the states and returns that are collected for fitting.
- In the **extreme case**, if we have only one return data sample coming from each state, and if the baseline fits this data perfectly**,** the **PG** would be **zero**.



Case 2: Baseline fitting is performed in iteration $k-1$ and the baseline is used for computing PG in iteration k.

- When the policy changes drastically between iterations, the baseline can be a poor estimate of the state-value function, resulting in a **poor fitting.**
- If the policy doesn't change by much, the **learning** will be **slow**.



K-fold method for Baseline estimation

- Break the data samples (states and rewards) into K **partitions**

- For each partition, a baseline is trained using data from all the other partitions, and the same baseline is used for predicting the value function for the current partition.

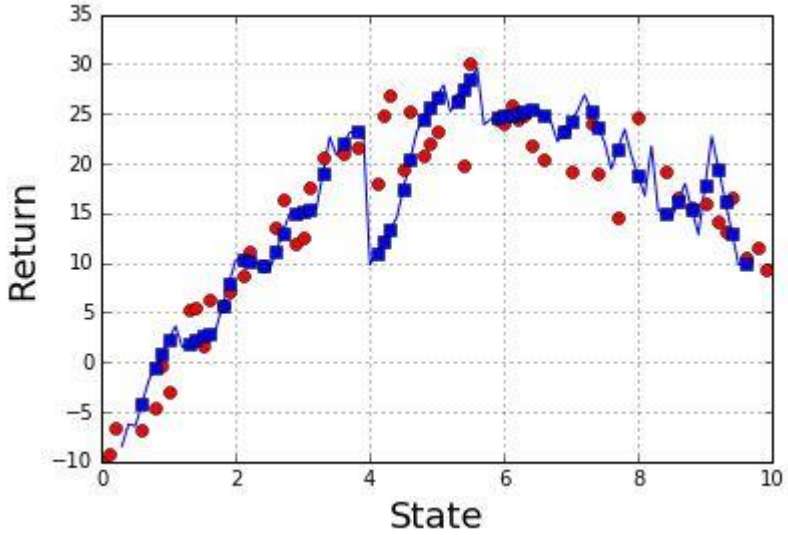


Illustration for $K = 2$

K-fold method for Baseline estimation (Contd.)

- Since we do not directly fit on `the current partition's data samples, we also avoid the issue of gradient becoming small and quicken learning (case 1) .
- Since the baseline fitting is performed using samples from the current policy, we mitigate the problem of poor fitting (case 2).
- $\boldsymbol{K}$ is a **hyper-parameter** that trades-off between case 1 ($K = data\_sample\_size$) and case 2 (K =1).
- By varying K, we can potentially **quicken learning** and improve performance.
- The **optimal value** of K may be found using standard **hyper-parameter tuning** techniques.

K-fold method for Baseline estimation (Contd.)

- Since the K-fold method operates on K partitions of the data, we obtain
  - K different baselines and therefore, K different policy gradients,

  How do we combine them to update the parameters θ?

- Recall: $\theta_{k+1} = \theta_k + \alpha \nabla_\theta J(\theta)$,
  where $\nabla_\theta J(\theta) = \nabla_\theta E_{s,a}[R_\theta(s,a)]$.

- Now, we have $\nabla_\theta J_1(\theta)$, $\nabla_\theta J_2(\theta)$, ... , $\nabla_\theta J_K(\theta)$ – one from each partition.

- There are at least two different ways to perform policy optimization:
  - **Parameter-based**: Average the updated parameters across the partitions to obtain the policy's new parameters.
  - **Gradient-based**: Average the PG across the partitions, and use the averaged gradient to obtain the policy's new parameters.

Method 1: Averaging the policy parameters

Algorithm 1 Parameter-based K-Fold Baseline Estimation for Policy Optimization

Initialize: For iteration $i = 0$, initialize the policy parameter θ_0 randomly.
Iterate: Repeat for each iteration $i, i \in 1, 2, \ldots$ until convergence:

1: Sample N trajectories from policy $\pi(\cdot|\cdot, \theta_{i-1})$: $\tau_{j:1,\ldots,N}$.
2: Partition the N trajectories into $K < N$ disjoint folds: $\tau_{j_1}, \tau_{j_2}, \ldots, \tau_{j_K}, \cup_{i=1}^{N} \tau_{j_i} = \tau_j$
3: **for** each partition k in $\{1, \ldots, K\}$ **do**
4: For each state $s_t^{j_k}$, compute the discounted returns $R_t^{j_k} = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}^{j_k}$
5: Fit a baseline regression model $b_k^{(i)}(\cdot)$ for partition k using data samples from all the remaining partitions, i.e., with inputs $s_t^{j_l}$ and outputs $R_t^{j_l}, l = 1, \ldots, K, l \neq k$.
6: For each partition k, initialize the policy with parameters θ_{i-1} and use any policy optimization algorithm to find optimized policy parameters θ_i^k: $\theta_{i-1} \rightarrow \theta_i^k$.
7: **end for**
8: Update the policy parameters $\theta_{i-1} \rightarrow \theta_i$ as the average of all the optimized policy parameters obtained, i.e., $\theta_i = \frac{1}{K} \sum_{k=1}^{K} \theta_i^k$. =0

Method 2: Averaging the gradients

Algorithm 2 Gradient-based K-Fold Baseline Estimation for Policy Optimization

Initialize: For iteration $i = 0$, initialize the policy parameter θ_0 randomly.

Iterate: Repeat for each iteration $i, i \in 1, 2, \ldots$ until convergence:

1: Sample N trajectories from policy $\pi(\cdot|\cdot, \theta_{i-1})$: $\tau_{j:1,\ldots,N}$.

2: Partition the N trajectories into $K < N$ disjoint folds: $\tau_{j_1}, \tau_{j_2}, \ldots, \tau_{j_K}, \cup_{i=1}^{N} \tau_{j_i} = \tau_j$

3: **for** each partition k in $\{1, \ldots, K\}$ **do**

4: For each state $s_t^{j_k}$, compute the discounted returns $R_t^{j_k} = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}^{j_k}$

5: Fit a baseline regression model $b_k^{(i)}(\cdot)$ for partition k using data samples from all the remaining partitions, i.e., with inputs $s_t^{j_l}$ and outputs $R_t^{j_l} o$, $l = 1, \ldots, K, l \neq k$.

6: Evaluate the gradient for fold k, g_k using predictions from the baseline $b_k^{(i)}(\cdot)$.

7: **end for**

8: Compute the average gradient $g = \frac{1}{K} \sum_{k=1}^{K} g_k$.

9: Use the average gradient g in any policy optimization algorithm to update the policy parameters $\theta_{i-1} \rightarrow \theta_i$. =0

Environments

Walker

Make a bipedal robot walk forward as fast as possible.

21-D state space
6 actuated joints

Hopper

Make a one-legged robot hop forward as fast as possible.

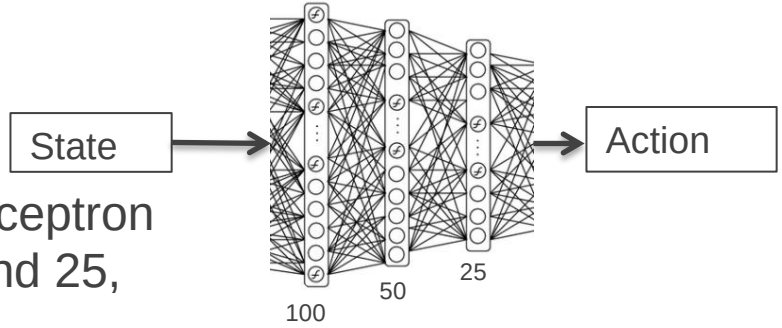
20-D state space
3 actuated joints

Cheetah

Make a two-legged robot run forward as fast as possible.

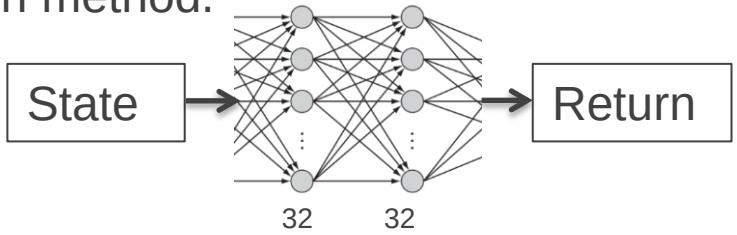
20-D state space
6 actuated joints

Experimental setup



- **Policy network**: feed-forward Multi-layer Perceptron (MLP) with 3 hidden layers of sizes 100, 50 and 25, and tanh nonlinearity after the first two layers.
  The policy network maps states to the mean of a Gaussian distribution.

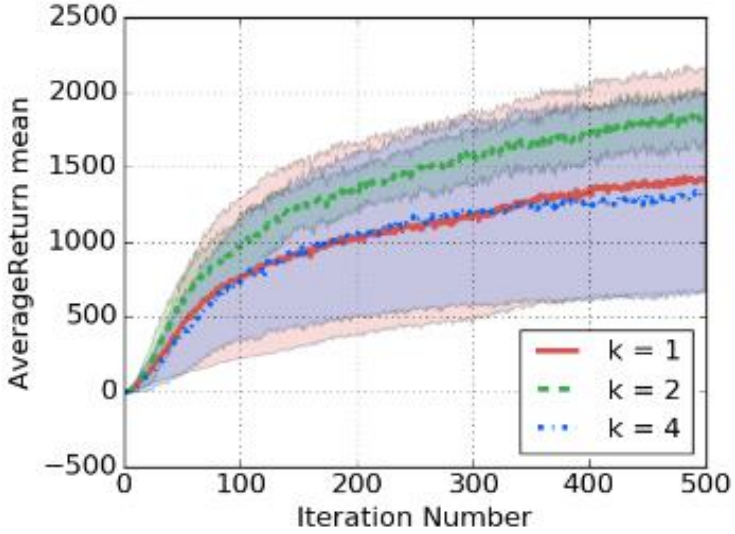
- **Baseline network**: feed-forward MLP with 2 hidden layers of size 32 each.
  The baseline uses ADAM first-order optimization method.



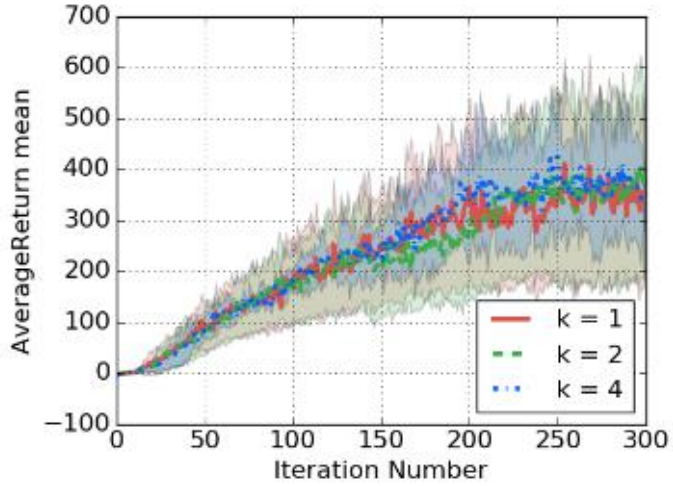
- Two policy gradient algorithms:
  - **TRPO**: Trust Region Policy Optimization
  - **TNPG**: Truncated Natural Policy Gradient.

- **K** = 1, 2 and 4

Results with Walker

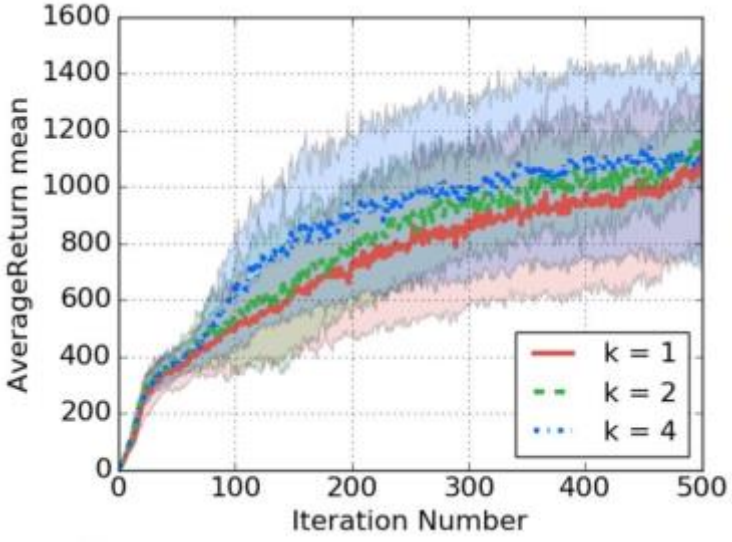


Walker with TRPO, and a data size of 50000
Lower variance

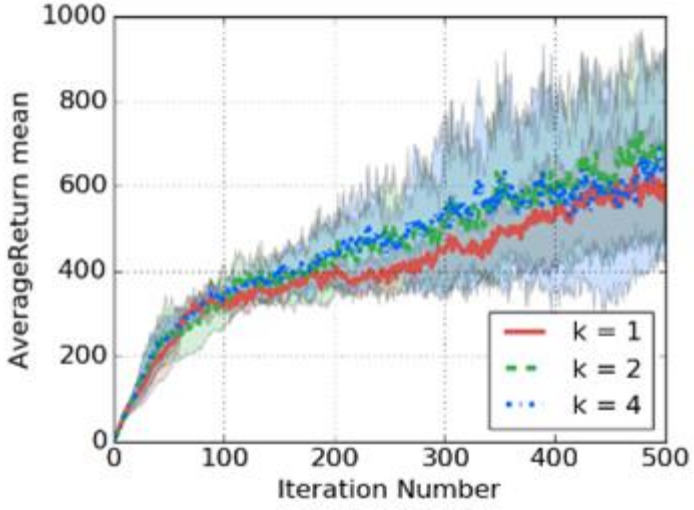


Walker with TNPG, and a data size of 5000
Stable learning

Results with Hopper



Hopper with TRPO, and a data size of 50000
Significantly faster learning



Hopper with TNPG, and a data size of 5000
Improved performance

Result Tables

Method 1 with TRPO and data size of 50,000.			
Task	$K = 1$	$K = 2$	$K = 4$
Walker	911.0 ± 681.0	$\mathbf{1015.7 \pm 327.3}$	938.7 ± 462.1
Hopper	727.7 ± 242.6	723.7 ± 190.5	$\mathbf{721.4 \pm 149.5}$
Cheetah	$\mathbf{1595.1 \pm 404.4}$	1528.5 ± 406.6	1383.8 ± 356.1

Method 2 with TRPO and data size of 50,000.			
Task	$K = 1$	$K = 2$	$K = 4$
Walker	911.0 ± 681.0	1035.0 ± 491.1	$\mathbf{1092.8 \pm 401.2}$
Hopper	727.7 ± 242.6	$\mathbf{786.0 \pm 171.1}$	847.7 ± 274.0
Cheetah	1595.1 ± 404.4	1664.1 ± 337.1	$\mathbf{1676.1 \pm 333.4}$

Method 2 with TNPG and data size of 5,000.			
Task	$K = 1$	$K = 2$	$K = 4$
Walker	299.4 ± 154.0	316.6 ± 164.6	$\mathbf{336.7 \pm 91.9}$
Hopper	331.4 ± 42.6	317.3 ± 29.5	$\mathbf{344.7 \pm 31.9}$
Cheetah	$\mathbf{609.5 \pm 215.3}$	445.9 ± 228.8	445.9 ± 181.9

Numbers indicate mean$\pm$std of the obtained returns.

The numbers with the **best lower bound**, i.e., highest mean-std are in bold face.

Summary

- Proposed K-fold method provides an **additional degree of freedom** to further the performance of Policy Gradient algorithms.
- Our algorithms exhibit **promising performance improvements** on robotic tasks, and with two policy gradient algorithms.
- The K-fold method is **generic** enough to be used with any policy gradient algorithm such as VPG/TRPO/TNPG.
- Interesting to study the benefit with the K-fold method applied to other **practical environments** and policy gradient algorithms.

Acknowledgements

- Thanks to our collaborators from open AI and UC Berkeley, **Xi (Peter) Chen** and **Pieter Abbeel**.

- Thanks to **Girish Kathalagiri** and **Aleksander Beloi** for their help during various stages of this work.

- Thanks to **Luis Carlos Quintela** and **Atul Varshneya** for their suggestions and feedback.

- Workshop website: **https://sites.google.com/site/deeprlnips2016/**