Policy Gradient Methods

CSE599G: Deep Reinforcement Learning

Aravind Rajeswaran and Kendall Lowrey

University of Washington Seattle

April 11, 2018

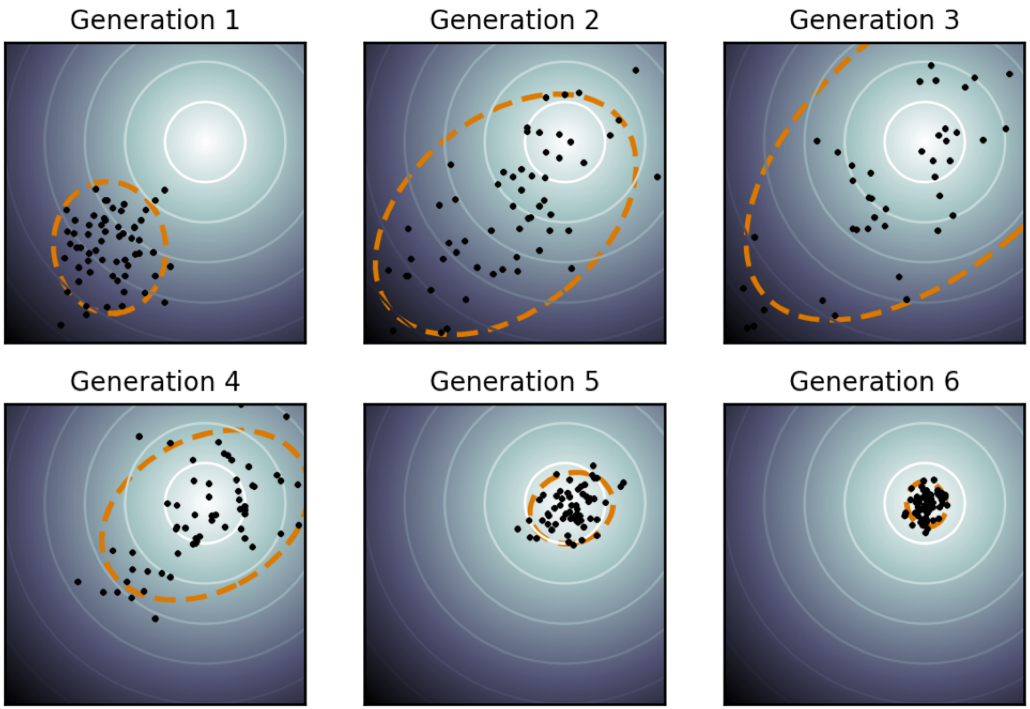
Score Function Estimator

- Let us pick some sampling distribution for the parameters: $\theta \sim P(.|\mu)$
- Here μ are parameters of sampling distribution (e.g. mean, variance of Gaussian)
- We want a sampling distribution that samples more often in the “good” parts
- Optimize over parameters of the sampling distribution

$$\max_{\mu \in \mathbb{R}^d} \eta(\mu) := \mathbb{E}_{\boldsymbol{s_0 \sim \rho_0}, \boldsymbol{\theta \sim P(.|\mu)}} \left[\sum_{t=0}^{T} \gamma^t \, r(s_t, \pi(.|s_t; \theta)) \right]$$

Note: the distribution for collecting samples depends on the values of decision variables. Very different from stochastic optimization problems in supervised learning where the data comes from a fixed distribution.

Changing the sampling distribution



Plots above use CMA-ES, we use it here only for illustrative purposes

Score Function Estimator

For the scoring function: $\eta(\mu) \coloneqq \mathbb{E}_{x\sim P(.|\mu)}[f(x)]$

We derived the gradient to be:

$$\nabla_\mu \eta(\mu) = \nabla_\mu \mathbb{E}_{x\sim P(.|\mu)}[f(x)] = \mathbb{E}_{x\sim P(.|\mu)}\left[f(x)\nabla_\mu \ln P(x|\mu)\right]$$

Since this has the form of expected value of some quantity, we can get unbiased estimates based on sampling. The sample based estimator is:

$$\nabla_\mu \eta(\mu) = \frac{1}{K}\sum_{i=1}^{K} f(x_i)\,\nabla_\mu \ln P(x = x_i|\mu)$$

Score Function Estimator

Final Algorithm

In a loop:

- Sample policy parameters $\theta^1, \theta^2, \theta^3, \dots, \theta^K$ using $P(.|\mu)$
- Compute respective evaluations or "fitness": $R(\theta^1), R(\theta^2), \dots, R(\theta^K)$
- Compute gradient: $\nabla_\mu \eta(\mu) = \frac{1}{K}\sum_{i=1}^{K} R\left(\theta^i\right)\nabla_\mu \ln P(\theta = \theta^i|\mu)$
- Gradient ascent: $\mu \leftarrow \mu + \alpha\nabla_\mu \eta(\mu)$

- Similar to weighted maximum likelihood estimation
- Theoretical analysis for the Gaussian sampling distribution by Nesterov
- We are optimizing for a ”smoothed” objective
- Has a terribly high variance (we will return to variance reduction later)

Understanding the expression

For a Gaussian sampling distribution, this looks basically like stochastic finite differencing, random projected gradient descent etc. (see Ben Recht's blog post)

Let us pick a Gaussian sampling distribution with mean μ and covariance I

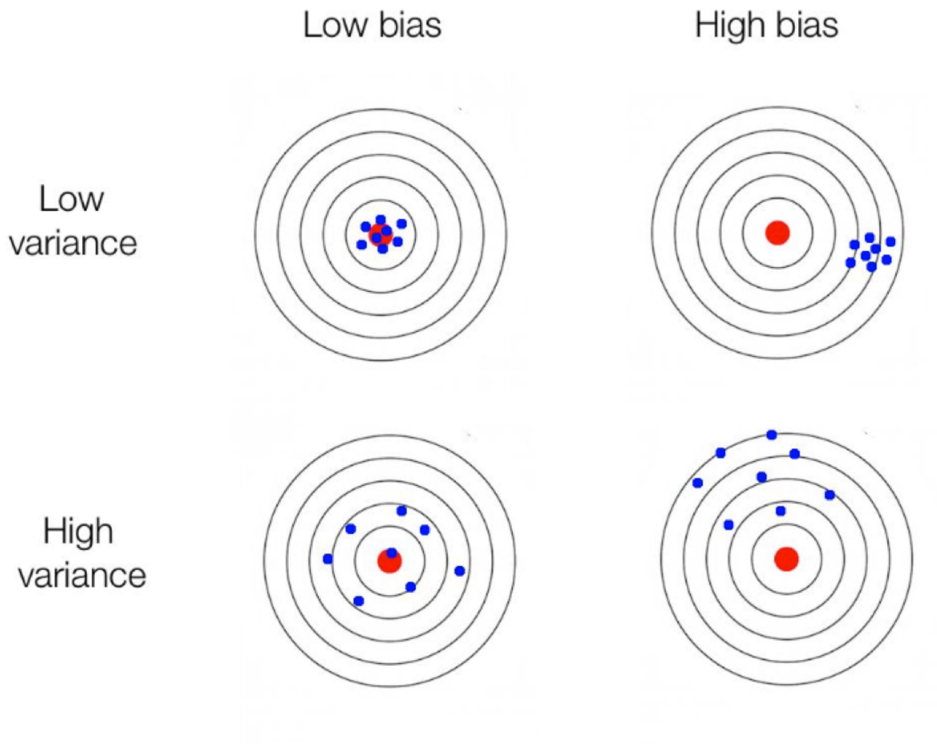
$$\nabla_\mu \ln P\left(\theta = \theta^i \middle| \mu\right) = \theta^i - \mu$$

The score function gradient is thus:

$$\nabla_\mu \eta(\mu) = \frac{1}{K} \sum_{i=1}^{K} R\left(\theta^i\right)\left(\theta^i - \mu\right)$$

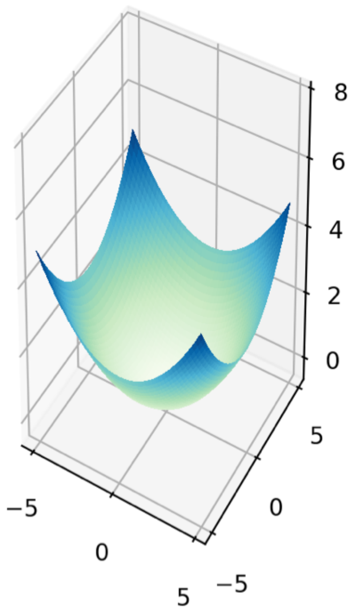
Pick k random directions, and perform finite differencing (almost) along those directions. Has the correct expected value but what about the variance?

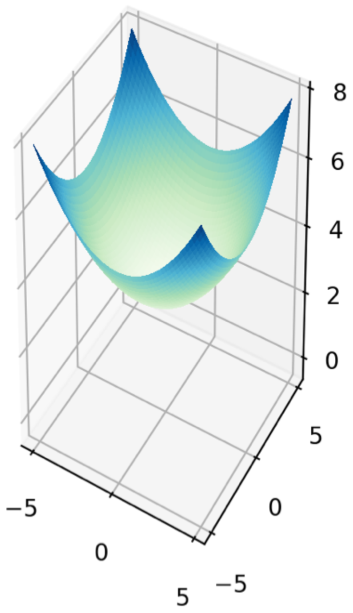
Bias-Variance tradeoff



Variance in PG

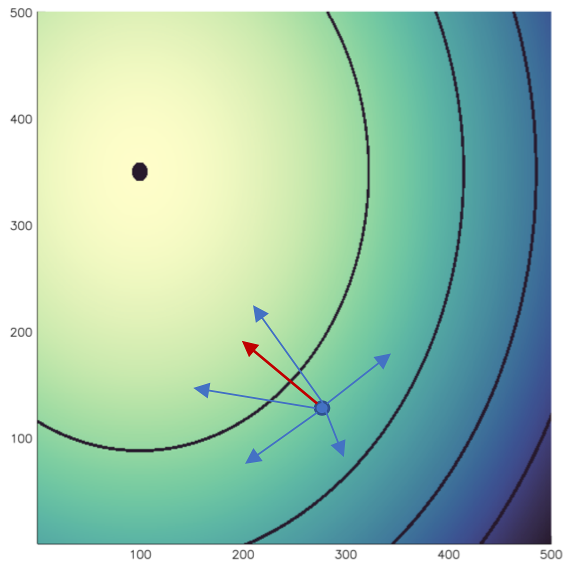
X and Y axis are different state/action dimensions. Z axis is the total reward

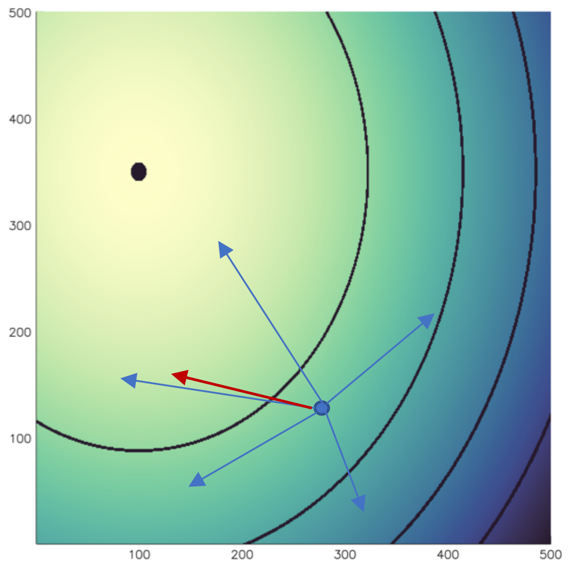




No optimization algorithm worth its salt should be sensitive to constant offsets

Variance in PG





Having a constant offset tells the algorithm that all directions are nearly equally good:
It makes it hard to distinguish between good directions and bad directions, which manifests as high variance in the gradient estimates.

Variance in PG

- The high variance is because we are not doing finite differences “properly”
- We can fix this by introducing a “reinforcement baseline”:

$$\nabla_\mu \eta(\mu) = \frac{1}{K}\sum_{i=1}^{K}\left(R(\theta^i) - \boldsymbol{R(\mu)}\right)\nabla_\mu \ln P(\theta = \theta^i|\mu)$$

- In the Gaussian case, this is:

$$\nabla_\mu \eta(\mu) = \frac{1}{K}\sum_{i=1}^{K}\left(R(\theta^i) - \boldsymbol{R(\mu)}\right)(\theta^i - \mu)$$

- Is this still unbiased? Yes!

Variance in PG

Theorem: For any constant b, we have:

$$\mathbb{E}_{\theta\sim P(.|\mu)}\left[b\,\nabla_\mu \ln P(\theta|\mu)\right] = 0$$

Proof: Let's apply the general form of the score function backwards. The score function estimator was:

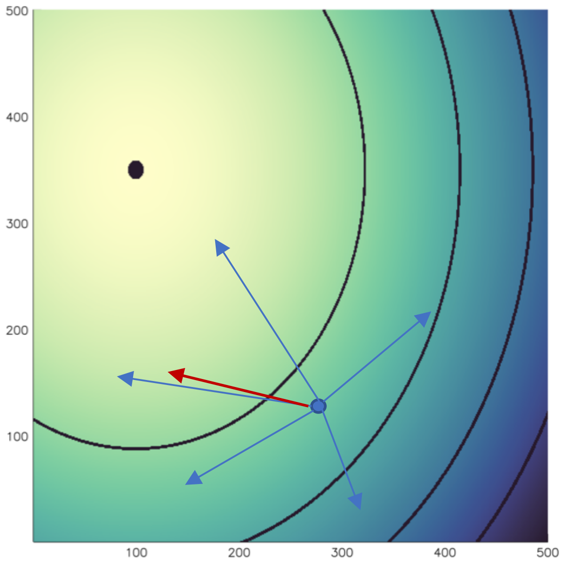
$$\nabla_\mu \mathbb{E}_{\theta\sim P(.|\mu)}[\eta(\theta)] = \mathbb{E}_{\theta\sim P(.|\mu)}[\eta(\theta)\nabla_\mu \ln P(\theta|\mu)]$$

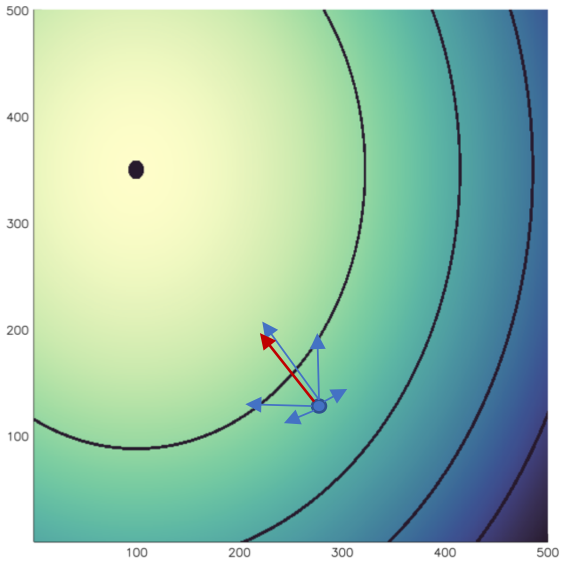
Using this backward, we have:

$$\mathbb{E}[b\nabla_\mu \ln P(\theta|\mu)] = \nabla_\mu \mathbb{E}[b] = \nabla_\mu[constant] = 0$$

A slightly sub-optimal, but a good choice for the value of b is $\eta(\mu)$ or $R(\mu)$

With reinforcement baseline





Compare how good each direction is to some “average” or baseline value as opposed to comparing the value absolutely to zero. This reduces variance tremendously.

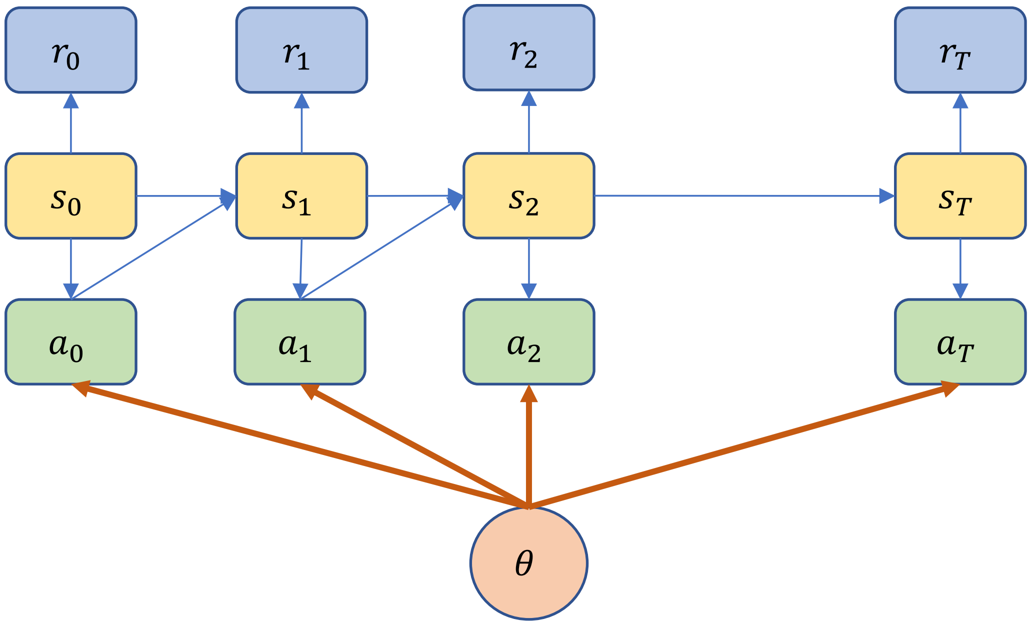
Flow graph of RL

Which parts are differentiable in the general case?

$$\begin{tikzcd}
r_0 & r_1 & r_2 & r_T \\
s_0 \arrow[u] \arrow[d] \arrow[r] & s_1 \arrow[u] \arrow[d] \arrow[r] & s_2 \arrow[u] \arrow[d] \arrow[r] & s_T \arrow[u] \arrow[d] \\
a_0 \arrow[ur] & a_1 \arrow[ur] & a_2 & a_T \\
 & & \theta \arrow[ull] \arrow[ul] \arrow[u] \arrow[ur] &
\end{tikzcd}$$

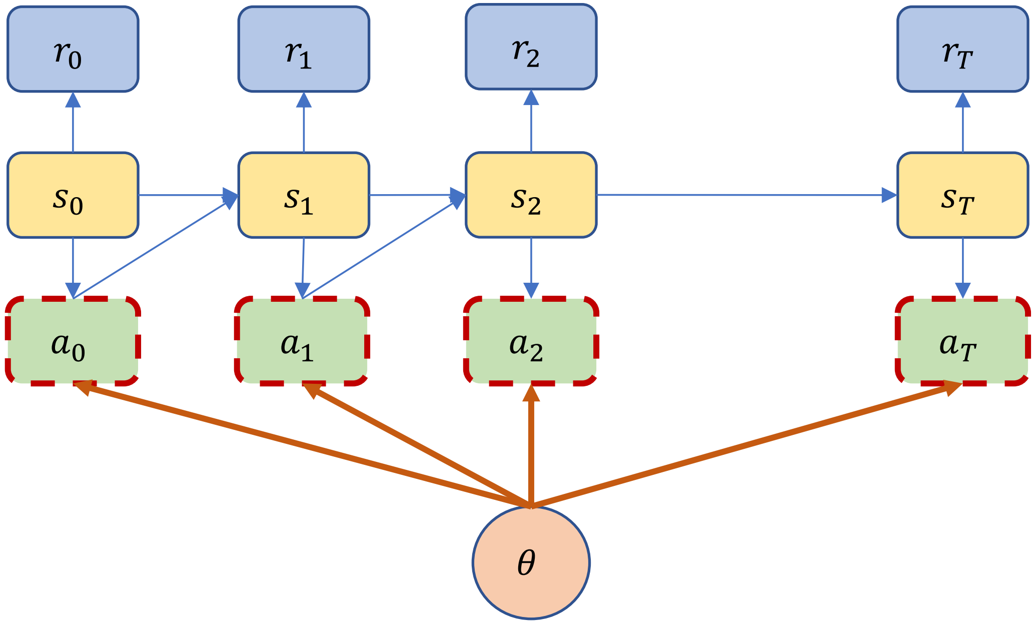
Flow graph of RL

Policy parameterization is known to us: we can certainly differentiate through it!



Flow graph of RL

Instead of sampling the policy parameters, sample the actions!



REINFORCE

Sample in the action space and use the score function estimator. In this case, we will pick a stochastic policy, which defines the sampling distribution.

$$\nabla_\theta \eta(\theta) = \mathbb{E}\left[\sum_{t=0}^{T} \nabla_\theta \ln \pi(a_t|s_t) \left(\sum_{t'=t}^{T} \gamma^{t'} r_{t'}\right)\right]$$

- Sample the action a_t at time t
- Compute the probability of the specific sample $\nabla_\theta \ln \pi(a_t|s_t)$
- An estimate of the long term consequence is $\left(\sum_{t'=t}^{T} \gamma^{t'} r_{t'}\right)$ which serves like the weight in the the weighted MLE form of the estimator

Monte Carlo REINFORCE

In a loop:

- Sample K trajectories: $\tau^1, \tau^2, \tau^3, \dots, \tau^K$ using π_θ
  where $\tau^i = (s_0^i, a_0^i, r_0^i, s_1^i. a_2^i, r_2^i, \dots, s_T^i, a_T^i, r_T^i)$
- For every trajectory and time-step, compute future performance:
  $R_t^i = \sum_{t'=t}^{T} \gamma^{t'} r_t^i$ which serves as the fitness or score.
- Compute gradient: $\nabla_\theta \eta(\theta) = \frac{1}{K}\frac{1}{T}\sum_{i=1}^{K}\sum_{t=0}^{T} R_t^i \, \nabla_\theta \ln \pi(a_t^i | s_t^i; \theta)$
- Gradient ascent: $\theta \leftarrow \theta + \alpha \nabla_\theta \eta(\theta)$

A good choice for reinforcement baseline is the value function, i.e. $b(s_t) \approx V^\pi(s_t)$

$$\nabla_\theta \eta(\theta) = \frac{1}{K}\frac{1}{T}\sum_{i=1}^{K}\sum_{t=0}^{T}\left(R_t^i - b(s_t)\right) \nabla_\theta \ln \pi(a_t^i | s_t^i; \theta)$$

Is there a better search direction?

How can we understand ”steepest” descent?

We want some local iterative optimization procedure for objective $\eta(\theta)$. Let’s say we are at iterate θ_k and we have some distance measure $d(\theta, \theta_k)$. We wish to consider all points that are distance ϵ away and move to the best point.

$$\max_{\theta} \ \eta(\theta)$$
$$s.t. \ \ d(\theta, \theta_k) = \epsilon$$

Then, we can interpret the steepest descent direction as one that points towards the maximizer of the above local optimization problem.

Is there a better search direction?

For a lot of optimization problems, we can locally approximate the distance measure by a quadratic function and the objective by a linear function. This is because we don't want to move too much in an iterative optimization procedure.

$$\eta(\theta) \approx \eta(\theta_k) + g^T(\theta - \theta_k)$$

$$d(\theta, \theta_k) \approx \frac{1}{2}(\theta - \theta_k)^T B(\theta - \theta_k)$$

Note: Sensible distance functions have the property that $d(\theta, \theta_k) = 0$ and $\theta = \theta_k$ is a minimizer of the distance function, i.e. ${}^{\partial d}/_{\partial \theta}\,|_{\theta=\theta_k} = 0$. So pure quadratics are a reasonable approximation, without the linear form.

Is there a better search direction?

$$\max_{\theta} \; g^T(\theta - \theta_k)$$

$$s.t. \; \frac{1}{2}(\theta - \theta_k)^T B(\theta - \theta_k) = \epsilon$$

The solution is:

$$\theta_{k+1} = \theta_k + \sqrt{\frac{2\epsilon}{g^T B^{-1} g}} \, B^{-1} g = \alpha B^{-1} g$$

Best search direction is $B^{-1}g$ and depends on the choice of distance function.

- $d(\theta, \theta_k) = \left|\left|\theta - \theta_k\right|\right|_2^2$ implies $B = I$ in which case we get gradient descent
- $d(\theta, \theta_k) = \frac{1}{2}(\theta - \theta_k)^T H(\theta - \theta_k)$ implies we get a Newton-like method

Is there a better search direction?

In PG, we care about the sampling distribution, not the parameters themselves. So, the correct distance measure should be something that acts on the resulting distribution, as opposed to directly on the weights of the network.

A sensible distance measure for probability distributions is KL divergence.

$$KL(\theta_1||\theta_2) = \sum_i P_{\theta_1}(i)\left[\ln P_{\theta_1}(i) - \ln P_{\theta_2}(i)\right]$$

Quadratic approximation to the KL divergence gives us the Fisher matrix.

$$KL(\theta_1||\theta_2) \approx \frac{1}{2}(\theta_1 - \theta_2)^T F(\theta_1 - \theta_2)$$

Natural Policy Gradient

Using the distance measure appropriate for probability distributions, we have:

$$\max_{\theta} \; g^T(\theta - \theta_k)$$

$$s.t. \; \frac{1}{2}(\theta - \theta_k)^T F(\theta - \theta_k) = \epsilon$$

The best search direction: $F^{-1}g$ is called the natural (policy) gradient.

Natural gradient works much better than the vanilla gradient g in RL.

We can also employ the step size derived earlier (normalized NPG or TRPO)

$$\theta_{k+1} = \theta_k + \sqrt{\frac{2\epsilon}{g^T F^{-1} g}} \; F^{-1} g$$

Natural Policy Gradient

How to estimate the Fisher matrix? There are two estimators.

$$F = \nabla_\theta^2 KL(\theta|\theta_k)\Big|_{\theta=\theta_k}$$

Which comes directly from the definition. If we know an analytical expression for KL, we can use it and derive the hessian of the expression. Possible for many known probability distributions like Gaussian, Categorical etc. But in general, a bit complex.

Use a sample based estimator for the Fisher matrix, which is an unbiased estimator. Easier in general and approximation quality is not that bad.

$$F = \mathbb{E}[\nabla_\theta \ln \pi(a|s;\theta_k)\ \nabla_\theta \ln \pi(a|s;\theta_k)^T]$$

Natural Policy Gradient

In a loop:

- Sample K trajectories: $\tau^1, \tau^2, \tau^3, \dots, \tau^K$ using π_θ
  where $\tau^i = (s_0^i, a_0^i, r_0^i, s_1^i. a_2^i, r_2^i, \dots, s_T^i, a_T^i, r_T^i)$
- For every trajectory and time-step, compute future performance:
  $R_t^i = \sum_{t'=t}^{T} \gamma^{t'} r_{t'}^i$ which serves as the fitness or score.
- Compute gradient: $g = \frac{1}{K}\frac{1}{T}\sum_{i=1}^{K}\sum_{t=0}^{T}(R_t^i - b\left(s_t^i\right))\, \nabla_\theta \ln \pi(a_t^i|s_t^i;\theta)$
- Compute Fisher matrix: $F =$
  $\frac{1}{K}\frac{1}{T}\sum_{i=1}^{K}\sum_{t=0}^{T} \nabla_\theta \ln \pi(a_t^i|s_t^i;\theta)\, \nabla_\theta \ln \pi\left(a_t^i\middle|s_t^i;\theta\right)^T$
- Compute NPG: $\beta = F^{-1} g$
- Gradient ascent: $\theta \leftarrow \theta + \sqrt{\frac{2\delta}{g^T\beta}}\beta$

Finer implementation details

- Existing auto grad frameworks are not very efficient at computing per sample gradients. You will implement per sample gradient in the homework, but you shouldn't do it for larger projects.
- You can subsample and still get a good estimate of Fisher matrix – for example, use all points to compute g but only (say) 10% of points for F
- In general, we don't want to invert a big matrix to solve for $F^{-1}g$. We will use the **conjugate gradient** algorithm. This is an incremental procedure for solving matrix equations that only require knowing: Fv for any v. The algorithm incrementally updates v so that we eventually get $Fv = g$.

Finer implementation details

- To get Fv we don't actually need to fully form F
- See that $F = {}^1/_N \; LL^T$ where L is the matrix appropriately stacking $\nabla_\theta \ln \pi(a_t^i | s_t^i; \theta_k)$ for different samples (i, t).
- We have $Fv = {}^1/_N \; LL^T v = {}^1/_N \; L\tilde{v}$ so we can avoid a big LL^T computation
- If we know the analytical KL divergence expression, we can use autograd: $Fv = (\nabla^2 KL)v = \nabla(\nabla KL^T v)$ – we do a backward pass through a forward + backward pass. The second backward pass requires storing the first forward and backward passes, so a bit memory intensive, but not that bad.
- Use 10-20 conjugate gradient iterations. Doing more seems to make things a bit unstable – some effect similar to overfitting. It's possible to mitigate this with regularization, but needs proper tuning.