Finding curvature in RL objective surfaces

Ben Black

1 Outline

The first step to understanding curvature in the objective surface is to compute the Hessian of that surface around a point. The first section derives a formula to estimate that local Hessian. The formalism follows the derivation of the policy gradient very closely.

The second step is to create an algorithm that computes the Hessian linear operation (i.e. can compute a Hessian-vector dot product) efficiently. In particular, it should allow for dense, batched estimation on a GPU.

The third step is to identify the maximally curved directions in the space, by computing the minimum and maximum eigenvalues and their corresponding eigenvectors.

2 Policy Hessian Theorem

Notation taken from `https://sites.google.com/view/icml17deeprl`

In particular, J is the objective function, θ are the parameters of the RL policy, τ is all the states and actions in an episode trajectory, S_i is the state at timestep i, a_i is the action at step i.

Def: The policy hessian is

$$H_\theta(J(\theta)) = \nabla_\theta \nabla_\theta J(\theta)$$

In order to derive an efficient algorithm we will use the following two lemmas:

2.1 calculus trick lemma

$$\nabla_x P(x) = P(x) \nabla_x \log(P(x))$$

proof:

$$P(x)\nabla_x \log(P(x)) = P(x)\frac{1}{P(x)}\nabla_x P(x) = \nabla_x P(x)$$

2.2 Policy gradient lemma

$$\nabla_\theta \log(P(\tau)) = \sum_{t=0}^{t_\tau} \nabla_\theta \log(\pi_\theta(s,a))$$

$$\nabla_\theta \log(P(\tau)) = \nabla_\theta(\log(P(S_0))) + \sum_{t=0}^{t_\tau} \nabla_\theta(\log(P(a_t|S_t))) + \nabla_\theta(\log(P(S_{t+1}|a_t,S_t)))$$

buf the transition function does not depend on theta, so the gradient is zero

$$= \sum_{t=0}^{t_\tau} \nabla_\theta(\log(P(a_t|S_t)))$$

$$= \sum_{t=0}^{t_\tau} \nabla_\theta \log(\pi(a_t,S_t))$$

2.3 Full derivation of formula

Def: The policy hessian is

$$\begin{aligned}
H_\theta(J(\theta)) &= \nabla_\theta \nabla_\theta J(\theta) \\
&= \nabla_\theta \nabla_\theta \mathbb{E}_{\tau\sim\pi_\theta} R(\tau) \\
&= \nabla_\theta \nabla_\theta \sum_\tau P_\theta(\tau) R(\tau) \\
&= \text{applying calculus trick lemma} \\
&= \nabla_\theta \sum_\tau P_\theta(\tau) \nabla_\theta \log(P_\theta(\tau)) R(\tau) \\
&= \sum_\tau ((\nabla_\theta P_\theta(\tau)) \nabla_\theta \log(P_\theta(\tau)) + P_\theta(\tau) \nabla_\theta \nabla_\theta \log(P_\theta(\tau))) R(\tau) \\
&= \text{applying calculus trick lemma} \\
&= \sum_\tau (P_\theta(\tau) \nabla_\theta \log(P_\theta(\tau)) \nabla_\theta \log(P_\theta(\tau)) + P_\theta(\tau) \nabla_\theta \nabla_\theta \log(P_\theta(\tau))) R(\tau) \\
&= \sum_\tau P_\theta(\tau) ((\nabla_\theta \log(P_\theta(\tau)))^2 + \nabla_\theta \nabla_\theta \log(P_\theta(\tau))) R(\tau) \\
&= \mathbb{E}_{\tau\sim\pi_\theta} ((\nabla_\theta \log(P_\theta(\tau)))^2 + \nabla_\theta \nabla_\theta \log(P_\theta(\tau))) R(\tau) \\
&\text{policy gradient lemma} \\
&= \mathbb{E}_{\tau\sim\pi_\theta} \left(\left(\sum_{t=0}^{t_\tau} \nabla_\theta \log(\pi_\theta(s,a)) \right)^2 + \nabla_\theta \left(\sum_{t=0}^{t_\tau} \nabla_\theta \log(\pi_\theta(s,a)) \right) \right) R(\tau) \\
&= \mathbb{E}_{\tau\sim\pi_\theta} \left(\left(\sum_{t=0}^{t_\tau} \nabla_\theta \log(\pi_\theta(s,a)) \right)^2 + \left(\sum_{t=0}^{t_\tau} \nabla_\theta \nabla_\theta \log(\pi_\theta(s,a)) \right) \right) R(\tau)
\end{aligned}$$

Reformatting the function to a useful format for estimation:

$$\begin{aligned}
&\mathbb{E}_{\tau\sim\pi_\theta} \left(\left(\sum_{t=0}^{t_\tau} \nabla_\theta \log(\pi_\theta(s,a)) \right)^2 + \nabla_\theta \left(\sum_{t=0}^{t_\tau} \nabla_\theta \log(\pi_\theta(s,a)) \right) \right) R(\tau) \\
&= \mathbb{E}_{\tau\sim\pi_\theta} \left(\sum_{t=0}^{t_\tau} \nabla_\theta \log(\pi_\theta(s,a)) \right) \left(\sum_{t=0}^{t_\tau} \nabla_\theta \log(\pi_\theta(s,a)) R(\tau) \right) + \nabla_\theta \left(\sum_{t=0}^{t_\tau} \nabla_\theta \log(\pi_\theta(s,a)) R(\tau) \right)
\end{aligned}$$

This formula will be used to create the policy hessian vector dot product algorithm below.

3 Policy Hessian vector dot product

Now, like all hessians over neural networks, it is large and inefficient to compute. Except worse because it is over an expectation of possibly sparse samples.

So instead, we can learn something about the Hessian by computing the Hessian vector dot product i.e. for vector v of the same shape as the parameter vector θ, compute

$$H(J(\theta))v$$

Given the rearranged version of the policy gradient above we have

$$H(J(\theta))v = \mathbb{E}_{\tau\sim\pi_\theta} \left(\sum_{t=0}^{t_\tau} \nabla_\theta \log(\pi_\theta(s,a)) \right) \left(\sum_{t=0}^{t_\tau} \nabla_\theta \log(\pi_\theta(s,a)) v R(\tau) \right) + \nabla_\theta \left(\sum_{t=0}^{t_\tau} \nabla_\theta \log(\pi_\theta(s,a)) v R(\tau) \right)$$

Which gives a natural, efficient batchable (within episodes) evaluation algorithm

Algorithm 1 Policy Hessian vector dot product

```
procedure DotHJ($\theta, v$)
    acc = $0 \cdot \theta$
    for $\tau \sim \pi_\theta$ do
        gradacc = $0 \cdot \theta$
        gradvecacc = $0$
        heshvecacc = $0 \cdot \theta$
        for Batched $S, a,$ ret in $\tau$ do
            logprobs = $\log(\pi_\theta(a, S))$
            grad = $\nabla_\theta \sum_b^{\text{batch}} \text{logprobs}_b$
            gradvec = $(\nabla_\theta \sum_b^{\text{batch}} \text{logprobs}_b \text{ret}_b)) \cdot v$
            heshvec = $\nabla_\theta$gradvec    ▷ During implementation, make sure to compute gradient using
autograds's create_graph=true so this line is computed correctly
            gradacc + = grad
            gradvecacc + = gradvec
            heshvecacc + = heshvec
        acc + = gradvecacc · gradacc + heshvecacc
    return acc / num_episodes
```

4 Eigenvalue/Eigenvector estimation

The next step is to estimate the eigenvectors and eigenvalues of the Hessian using the hessian vector dot product algorithm above. Note that Hessian is a linear operation that we can compute. Since we can compute this linear operation, standard linear algebra methods of finding the eigenvectors and eigenvalues of linear operations can be used

The ARPACK library[2] implements a number of efficient methods to solve linear algebra problems on arbitrary linear operations without a full matrix. In particular it can find estimates of the eigenvectors and eigenvalues of a linear operation using the Implicitly Restarted Lanczos Method, without having to compute all the entries of the matrix defining the linear operation. Scipy's python based `eigsh` interface offers a convenient way of accessing ARPACK functionality. Also note that [1]'s code [1] was helpful to develop this method of finding the eigenvectors and eigenvalues.

Note that this method does not guarantee an unbiased estimate of the eigenvector, in particular since we only have estimates of the actual linear operation under study. So the results should be treated with some care.

References

[1] Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. Visualizing the loss landscape of neural nets. In *Neural Information Processing Systems*, 2018.

[2] D. C. Sorensen R. B. Lehoucq and C. Yang. ARPACK USERS GUIDE: Solution of large scale eigenvalue problems by implicitly restarted arnoldi methods, 1998.

[1] `https://github.com/tomgoldstein/loss-landscape/blob/master/hess_vec_prod.py`