conjugate_gradient

April 25, 2018

연구자: 정회희 (2018/04/24)

해당 코드는 Understanding Black-box Predictions via Influence Functions 논문의 Conjugate Gradient를 구현한 코드이다.

0.1 Theoretical Backgrounds

Preliminaries

- Krylov Subspace
- Caley-Hamilton Theorem

Settings Given,

$$\hat{\theta} \in \mathbb{R}^p$$
$$v = \nabla L(z_{test}, \hat{\theta}) \in \mathbb{R}^p$$
$$H = \frac{1}{n} \sum_{i=0}^{n} \nabla^2 L(z_i, \hat{\theta}) \in \mathbb{R}^{p^2}$$

Our goal is to find $x^* \rightarrow H^{-1}v$

Algorithm

$x := 0, r := v, \rho_0 := ||r||^2$

for $ k=1,..., N_{max} $

if $\sqrt{\rho_{k-1}} \leq \epsilon ||v||$: break

if $k = 1$ then $p := r$; else $p := r + \left(\frac{\rho_{k-1}}{\rho_{k-2}}\right) p$

$w := Hp$

$\alpha := \frac{\rho_k}{p^T w}$

$x := x + \alpha p$

$r := r - \alpha w$

$\rho_k := ||r||^2$

Remark) - $w := Hp$ 진행할 때 HVP를 사용하면 $O(np)$만큼 computational complexity 듬. - 실제 code에서는 모든 training dataset에 대해서 HVP를 하려면 한번에 모든 training set에 대해서 feed_dict해줘야하기 때문에 메모리 문제로 HVP_minibatch를 구현함. (밑 코드 참고)

0.2 Implementations

실제 code에서는 모든 training dataset에 대해서 HVP를 하려면 한번에 모든 training set에 대해서 feed_dict 해줘야하기 때문에 메모리 문제로 HVP_minibatch를 구현함. (밑 코드 참고)

```
In [8]:     def minibatch_hessian_vector_val(self, v):

                num_examples = self.num_train_examples
                if self.mini_batch == True:
                    batch_size = 100
                    assert num_examples % batch_size == 0
                else:
```

```
                batch_size = self.num_train_examples

            num_iter = int(num_examples / batch_size)

            self.reset_datasets()
            hessian_vector_val = None
            for i in xrange(num_iter):
                feed_dict = self.fill_feed_dict_with_batch(self.data_sets.train,
        batch_size=batch_size)
                # Can optimize this
                feed_dict = self.update_feed_dict_with_v_placeholder(feed_dict, v)
                hessian_vector_val_temp = self.sess.run(self.hessian_vector,
        feed_dict=feed_dict)
                if hessian_vector_val is None:
                    hessian_vector_val = [b / float(num_iter) for b in
        hessian_vector_val_temp]
                else:
                    hessian_vector_val = [a + (b / float(num_iter)) for (a,b) in
        zip(hessian_vector_val, hessian_vector_val_temp)]

            hessian_vector_val = [a + self.damping * b for (a,b) in zip(hessian_vector_val,
        v)]

            return hessian_vector_val
```

원래 conjugate gradient method 방법은 $x^* = H^{-1}v$를 구하고 싶은데, H 차원이 너무 커서 직접적으로 matrix를 계산하고, 역행렬을 얻는 것이 힘들 때

$$x^* = \min_x \frac{1}{2} x^T H x - v^T x$$

를 풀어서 얻는 방법.

때문에 quadratic programming 형태여야만 이론적 근거를 충족시킴. (refer: Krylov space)

scipy package에서 제공하는 fmin_ncg는 임의의 convex 함수를 풀 때 conjugate gradient의 iteration 방법을 써서 풀겠다는 것. 주어진 objective function이 convex function 이라면 2nd order Taylor approximation을 할 수 있을 것이고 approximation을 하고 나면 정확하지는 않겠지만 어느정도 비슷하고 convergent한 해를 얻을 수 있다는 것. 따라서 fmin_ncg는 conjugate gradient의 iteration 방법으로 optimization 문제를 풀겠다는 것.

이렇게 풀 경우에는 Hessian을 직접적으로 구하지 않고 (HVP만 사용해서) Newton's method 방식으로 optimize가 될 것.

cf) 좀 더 general하게 설계되어있다 정도만 알면 됨. 어차피 우리는 objective function에 quadratic function을 넣을거고 그러면 전혀 문제 없음.

Scipy package의 fmin conjugate gradient는 HVP 함수를 explicit하게 제공하면 더 정확하게 풀 수 있음. 따라서 위의 HVP를 사용할 예정.

사실 HVP를 default로 해도 알아서 계산해주기 때문에 상관없지만, 우리의 경우 network parameter가 복잡하기 때문에 (layer별로 list화 되어있기 때문에), 그리고 한번에 많은 data point를 feed_dict하기 때문에 추가적인 작업이 필요함. 때문에 여기서는 명확하게 HVP를 제공함.

아래 코드는 scipy.optimize.fmin_ncg를 사용하여 conjugate gradient를 하는 것.

```
In [11]:     from scipy.optimize import fmin_ncg

             def get_inverse_hvp_cg(self, v, verbose):
                 fmin_loss_fn = self.get_fmin_loss_fn(v)
                 fmin_grad_fn = self.get_fmin_grad_fn(v)
                 cg_callback = self.get_cg_callback(v, verbose)

                 fmin_results = fmin_ncg(
                     f=fmin_loss_fn,
                     x0=np.concatenate(v),
                     fprime=fmin_grad_fn,
                     fhess_p=self.get_fmin_hvp,
                     callback=cg_callback,
                     avextol=1e-8,
                     maxiter=100)

                 return self.vec_to_list(fmin_results)
```

fmin_ncg의 input으로는

- f: objective function to be minimized ~ $f(x) = \frac{1}{2}x^T Hx - v^T x$ where $H \triangleq \frac{1}{n}\sum_{i=0}^{n} \nabla^2 L(z_i, x)$
- x0: initial guess
- fprime: gradient of f ~ $f(x) = Hx - v$
- fhess_p: function to compute the Hessian matrix of f ~ $f(x, p) = Hp$
- callback: an optional user-supplied function which is called after each iteration
- avextol: hyperparameter epsilon
- maxiter: maximum number of iterations to perform

이 있음.

위에서 설명했듯 f, fprim, fhess_p는 특정 값이 아니고 input을 받으면 output을 내보내는 함수여야 함.

code에서는 다음과 같이 정의함.

```
In [5]:     def get_fmin_loss_fn(self, v):

                def get_fmin_loss(x):
                    hessian_vector_val = self.minibatch_hessian_vector_val(self.vec_to_list(x))

                    return 0.5 * np.dot(np.concatenate(hessian_vector_val), x) -
```

```
        np.dot(np.concatenate(v), x)
                return get_fmin_loss


            def get_fmin_grad_fn(self, v):
                def get_fmin_grad(x):
                    hessian_vector_val = self.minibatch_hessian_vector_val(self.vec_to_list(x))

                    return np.concatenate(hessian_vector_val) - np.concatenate(v)
                return get_fmin_grad


            def get_fmin_hvp(self, x, p):
                hessian_vector_val = self.minibatch_hessian_vector_val(self.vec_to_list(p))

                return np.concatenate(hessian_vector_val)
```