# Implementations

April 25, 2018

연구자: 정회희 (2018/04/24)

이 페이지는 Understanding Black-box Predictions via Influence Functions 논문의 Conjugate Gradient와 Stochastic Estimation에 관한 추가 설명과 이를 Tensorflow로 구현한 내용을 담고 있다.

## 1 Conjugate Gradient

### 1.1 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)

## 1.2 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 [1]:     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 [2]:     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 [3]:     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)
```

# 2 Stochastic Estimation

## 2.1 Theoretical Backgrounds

Taylor series expansion을 통해서

$$H_j^{-1} \triangleq \sum_{i=0}^{j} (I - H)^i$$

Hessian matrix의 inverse를 approximate할 수 있다. 이 식을 새로 정리하면 다음과 같은 점화식을 얻을 수 있다.

$$H_j^{-1} = I + (I - H)H_{j-1}^{-1}$$

우리는 이 점화식을 반복하여 진행하며 Hessian의 inverse를 추정할 것이다. 이 점화식을 반복하기 위해서는 H를 계산해야한다. 이 때 H는 $\frac{1}{n}\sum_{i=0}^{n} \nabla^2 L(z_i, \hat{\theta})$ 값으로 이 값 정확하게 구하려면 지나치게 많은 계산량을 필요로 한다. 때문에 이를 unbiased estimator인 $\nabla^2 L(z_i, \hat{\theta})$로 대치하여 iteration을 진행할 것이라는게 논문의 내용이다. 이 때 unbiased estimator of Hessian을 $\tilde{H}$라고 하자. 실제 구현 코드에서는 unbiased estimator $\tilde{H}$를 좀 더 잘 (낮은 variance로 추정하기) 구하기 위해서 몇 가지 방법을 제시하고 있다. 나는 이 방법을 임의로 minibatching, scaling이라고 명명했다.

**Minibatching** 하나의 sample만을 가지고 unbiased estimator를 사용하는 대신, 여러 sampling에 대해서 평균취하는 방법이다. 평균을 취했기 때문에 mean값에는 변화가 없고, variance는 줄어들 것으로 기대할 수 있다. 이 때 sample 수는 $m$개이고 당연히 $m << n$이다. e.g., Gaussian random variable이라고 가정할 경우 mean은 변함없고, variance는 $\frac{1}{m^2}$로 줄어들 것이다.

$$\begin{aligned} \tilde{H} &= \nabla^2 L(z_i, \hat{\theta}) \\ &\rightarrow \frac{1}{m}\sum_{i=0}^{m} \nabla^2 L(z\_i, \hat{\theta}) \end{aligned}$$

**Scaling** H를 추정하는 대신 $H' \triangleq \frac{H}{Scale}$을 추정하는 것이다. 상수배로 나뉘었기 때문에 variance가 줄어들 것이라고 예측할 수 있다. e.g., Gaussian random variable이라고 가정할 경우 variance가 $\frac{1}{Scale^2}$ 만큼 줄어들 것이다. 이는 minibatching과 다르게 하나의 step에서 실질적인 계산량은 변함없이 정확도를 향상시킬 수 있다. 하지만 수렴을 위해서 더 많은 recursion step (j)을 요구한다. 때문에 scaling factor는 learning rate와도 연관지어 생각할 수 있다.

$$\begin{aligned} {H'}_j^{-1} &= I + (I - H'){H'}_{j-1}^{-1} \\ &= I + (I - \frac{H}{Scale}){H'}_{j-1}^{-1} \end{aligned}$$

그리고 이 값은 ${H'}_j^{-1} = \left(\frac{H_j}{Scale}\right)^{-1}$를 추정한 것이기 때문에 $H_j^{-1}$을 얻기위해선 iteration을 통해 얻은 최종값을 Scale값으로 나눠야한다.

cf) Talor expansion이기 때문에 initialization 값은 변함이 없다, i.e., ${H'}_0^{-1} = I$.

**Damping** 마지막으로 code에는 damping term이 있다. 이 term은 앞서 Influence function을 얻을 때 strictly convex와 twice differentiable이라는 두 조건 중 strictly convex 조건을 더 강력하게 맞추기위해서 생겨난 regularization term이다. 특히 deep neural network 같은 경우에는 비용함수가 strictly convex라는 보장이 전혀 없기 때문에 꼭 필요한 term이다.

정확하게 말하자면 positive eigenvalue를 얻기위해 임의로 identity matrix를 더해주는 것으로 수식으로 표현하자면 다음과 같다.

$$\bar{H} = \tilde{H} + \lambda I$$

이 $\bar{H}$를 점화식에 넣게 되면 다음과 같은 식을 얻을 수 있다.

$$\begin{aligned} H_j^{-1} &= I + (I - \bar{H}) H_{j-1}^{-1} \\ &= I + \left( (1-\lambda) I - \tilde{H} \right) H_{j-1}^{-1} \end{aligned}$$

## 2.2 Implementations

이 방법을 통해서 찾고자하는 최종 값은 $\theta$와 차원이 같은 임의의 vector $v$와 Inverse of Hessian matrix의 곱, $H^{-1}v$ 값이다. 따라서 위 점화식의 양 변에 $v$를 곱하고 추정할 vector를 $s_j$로 새로 정의하면, i.e., $s_j \triangleq H_j^{-1}v$, 다음과 같은 반복식을 얻을 수 있다.

$$s_j = v + s_{j-1} + \bar{H} s_{j-1}$$

Strictly convex 조건을 만족하기위해서 damping term을 풀면 다음과 같은 식을 얻을 수 있다.

$$s_j = v + (1-\gamma) s_{j-1} + \tilde{H} s_{j-1}$$

위에서 설명한 minibatching과 scaling을 포함하여 점화식을 얻으면 다음과 같다.

$$s'_j = v + (1-\gamma) s'_{j-1} + \frac{\frac{1}{m}\sum_{i=0}^{m} \nabla^2 L(z\_i, \hat{\theta})}{Scale} s'_{j-1} \text{ where } s'_j = Scale \cdot s_j$$

실제 구현된 코드의 input과 theoretical background의 hyperparameter들과 관련하여 설명하자면

1. $\gamma \leftarrow$ damping : logistic regression 같이 strictly convex한 loss를 사용할 경우 0.0으로 두면 된다. 논문에서는 $\lambda = 0.01$로 사용해서 CNN network를 재 조정했다.
2. $Scale \leftarrow$ scale : 앞서 설명했듯 H에 대해서 iteration을 반복할 때 H를 scale로 나눠서 진행하는 것. vector s의 변화량이 줄어들기 때문에 variance가 줄어들 수 있겠지만 더 많은 recursion step이 필요할 것이다. recursion step을 거쳐서 얻은 후 다시 scale로 나누는 것을 잊지 말 것.
3. $m \leftarrow$ batch_size : 여러 sample을 사용하여 평균취하는 것으로 unbiased estimator를 얻은 것. 클 수록 좋겠지만 rtp * batch_size만큼 계산량이 느는 것이라 주의해야함. 실제 구현 단계에서는 total loss 자체가 minibatch에 대해서 reduce sum한 것이기 때문에 feed_dict만 minibatch wise로 하면 된다.

$O(rtp)$

$r$: num_samples

$t$: recursion_depth

$p$: dimension of $\theta$

```
In [4]:     def get_inverse_hvp_lissa(self, v,
                                      batch_size=None,
                                      scale=10, damping=0.0, num_samples=1,
        recursion_depth=10000):
                """
                This uses mini-batching; uncomment code for the single sample case.
                """
                inverse_hvp = None
                print_iter = recursion_depth / 10

                for i in range(num_samples):
                    # samples = np.random.choice(self.num_train_examples, size=recursion_depth)

                    cur_estimate = v

                    for j in range(recursion_depth):

                        # feed_dict = fill_feed_dict_with_one_ex(
                        #   data_set,
                        #   images_placeholder,
                        #   labels_placeholder,
                        #   samples[j])
                        feed_dict = self.fill_feed_dict_with_batch(self.data_sets.train,
        batch_size=batch_size)

                        feed_dict = self.update_feed_dict_with_v_placeholder(feed_dict,
        cur_estimate)
                        hessian_vector_val = self.sess.run(self.hessian_vector,
        feed_dict=feed_dict)
                        # gradient is summed among minibatch (since def loss <- total loss, not
        loss vector)
                        # v, cur_estimate, hessian_vector_val are vectors i.e. R^|v| (no
        minibatch dimension)
                        # cf) for feed_dict list concatenation doesn't matter
                        # since we do batch sampling, each point of iteration is changed at
        every recursion
                        cur_estimate = [a + (1-damping) * b - c/scale for (a,b,c) in zip(v,
        cur_estimate, hessian_vector_val)]

                        # Update: v + (I - Hessian_at_x) * cur_estimate
                        if (j % print_iter == 0) or (j == recursion_depth - 1):
                            print("Recursion at depth %s: norm is %.8lf" % (j,
        np.linalg.norm(np.concatenate(cur_estimate))))

                    if inverse_hvp is None:
                        # element wise division (divide values by scale among batch_size)
                        inverse_hvp = [b/scale for b in cur_estimate]
                    else:
                        # summation among trials (num_samples)
                        # still, inverse_hvp has batch_size X |v| dimension
```

```
                inverse_hvp = [a + b/scale for (a, b) in zip(inverse_hvp, cur_estimate)]
        inverse_hvp = [a/num_samples for a in inverse_hvp]
        return inverse_hvp
```

## 2.3 Furtherworks

시간이 남는다면 위 3개 hyperparameter의 효과를 실험적으로 확인해볼 것.

```
In [ ]:
```