Efficient Computation of Hessian Matrices in TensorFlow

Geir K. Nilsen[1,2], Antonella Z. Munthe-Kaas[1], Hans J. Skaug[1], and Morten Brun[1]

[1]*Department of Mathematics, University of Bergen*
[2]*geir.kjetil.nilsen@gmail.com*

Abstract

This paper deals with the practical aspects of efficiently computing Hessian matrices in the context of deep learning using the Python programming language and the TensorFlow library. We define a general feed-forward neural network model and show how to efficiently compute two quantities: the cost function's exact Hessian matrix, and the cost function's approximate Hessian matrix, known as the Outer Product of Gradients (OPG) matrix. Furthermore, as the number of parameters P in deep learning usually is very large, we show how to reduce the quadratic space complexity by an efficient implementation based on approximate eigendecompositions.

1 Introduction

The Hessian matrix has a number of important applications in a variety of different fields, such as optimzation, image processing and statistics. Geometrically, the Hessian matrix describes the local curvature of scalar functions $f : \mathbb{R}^P \to \mathbb{R}$, and is for this reason perhaps mostly known in the field of optimization [8]. Nevertheless, the Hessian matrix also has an important role in statistics, since its inverse is related to the powerful concept of uncertainty quantification [9].

In this technical note we mostly focus on the practical aspects of efficiently computing Hessian matrices in the context of deep learning [7] using the Python [10] programming language and the TensorFlow [1] library. We define a general feed-forward neural network model and show how to efficiently compute two quantities: the cost function's exact Hessian matrix, and the cost function's approximate Hessian matrix, known as the Outer Product of Gradients (OPG) matrix. Furthermore, as the number of parameters P in deep learning usually is very large, we show how to reduce the quadratic space complexity by efficient approximate eigendecompositions. Although we here use a feed-fordward neural network architecture to introduce terminology, the theory and implementation presented is still directly applicable on more general neural network architectures using convolutional layers, pooling and regularization.

The paper is organized as follows: In Section 2 we give definitions which will be used throughout the paper. In Section 3 we present the problem statement, and discuss three complications which need to dealt with in order to achieve a successful TensorFlow implementation: 1) `tf.hessians()` is fundamentally inadequate since it only

calculates a subset of all the partial derivatives (Section 3.3), 2) computing Hessian matrices essentially requires per-example gradients of the cost function with respect to model parameters, and unfortunately, the differentiation functionality provided by TensorFlow does not support computing gradients with respect to individual examples efficiently [2] (Section 3.1), and 3) when differentiating a function with respect to several variables represented by a list of tensors, the result is also a list of tensors (Section 3.2). In Section 5 we show how to overcome the aforementioned complications and introduce our Python module `pyhessian` [11] which is released as open source licensed under GNU GPL on GitHub. In Section 6 we summarize the paper and give some concluding remarks.

2 Deep Neural Networks

A feed-forward neural network is shown in Figure (1). There are L layers $l = 1, 2, ..., L$ with T_l neurons in each layer. The input layer $l = 1$, is represented by the input vector $x_n = \begin{bmatrix} x_{n,1} & x_{n,2} & \ldots & x_{n,T_1} \end{bmatrix}^T$ where $n = 1, 2, ..., N$ is the input index. Furthermore, there are $L - 2$ dense hidden layers, $l = 2, 3, ..., L-1$, and a dense output layer $l = L$, all represented by weight matrices $W^{(l-1)} \in \mathbb{R}^{T_l \times T_{l-1}}$, bias vectors $b^{(l)} \in \mathbb{R}^{T_l}$ and vectorized activation functions $\sigma^{(l)}$.
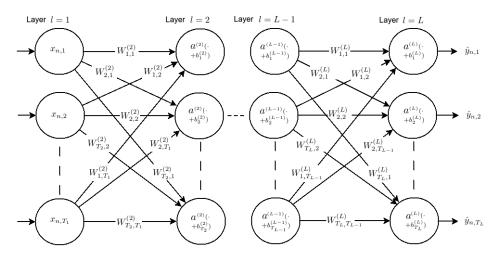

Figure 1: A Feed-Forward Neural Network with Dense Layers

Let the cost function C coincide with TensorFlow's built-in softmax cross-entropy function[1],

$$C = \frac{1}{N} \sum_{n=1}^{N} C_n(y_n, \hat{y}_n) \qquad (1)$$

$$= \frac{1}{N} \sum_{n=1}^{N} \left(- \sum_{m=1}^{T_L} y_{n,m} \log \hat{y}_{n,m} \right). \qquad (2)$$

It is defined as the average of N per-example cross-entropy cost functions $C_n(y_n, \hat{y}_n)$, where y_n represents the one-hot target vector for the nth

---

[1] TensorFlow API r1.13: tf.losses.softmax_cross_entropy()

$$\hat{y}_n = f(x_n, \omega) = \sigma^{(L)}(W^{(L-1)}\sigma^{(L-1)}(\cdots\sigma^{(2)}(W^{(1)}x_n + b^{(2)}) + \cdots) + b^{(L)}) \quad (3)$$

example, and where $\hat{y}_n$ represents the corresponding prediction vector. The prediction vector is obtained by evaluating the model function (3) using the input vector x_n and a flat vector of model parameters $\omega \in \mathbb{R}^P$ defined by

$$\omega = \begin{bmatrix} \omega_1 & \omega_2 & \dots & \omega_P \end{bmatrix}^T \quad (4)$$
$$= \underset{l=2,3,\dots,L}{\text{flatten}}(W^{(l-1)}, b^{(l)}). \quad (5)$$

The function flatten$(\cdot)$ denotes a row-wise flattening operation to transform the collection of model parameters represented by the weight matrices $W^{(l-1)}$ and bias vectors $b^{(l)}, l = 2, 3, ..., L$ into a flat column vector of dimension $P = T_1T_2 + T_2 + \ldots + T_{L-1}T_L + T_L$. Further, the activation function in the output layer is the vectorized softmax function

$$\sigma^{(L)}(z) = \text{softmax}(z) \quad (6)$$
$$= \frac{\exp(z)}{\sum_{m=1}^{T_L}\exp(z_m)}, \quad (7)$$

where $z \in \mathbb{R}^{T_L}$, and where $\exp(\cdot)$ denotes the vectorized exponential function. Finally, training of the neural network can be defined as finding an 'optimal' parameter vector $\hat{\omega}$ by minimizing the cost function (1),

$$\hat{\omega} = \arg\min_{\omega\in\mathbb{R}^P} C(\omega). \quad (8)$$

3 Computing Hessian Matrices in Tensor-Flow

Given the cost function C defined in Section 2, the Hessian matrix $H \in \mathbb{R}^{P\times P}$ is defined[2]

$$H = \left.\frac{\partial^2 C}{\partial\omega\partial\omega^T}\right|_{\omega=\hat{\omega}} \quad (9)$$

$$= \frac{1}{N}\sum_{n=1}^{N}\left.\frac{\partial^2 C_n}{\partial\omega\partial\omega^T}\right|_{\omega=\hat{\omega}}. \quad (10)$$

The approximation to the Hessian matrix, known as the Outer Product of Gradients (OPG) matrix $G \in \mathbb{R}^{P\times P}$, is defined

$$G = \frac{1}{N}\sum_{n=1}^{N}\left.\frac{\partial C_n}{\partial\omega}\frac{\partial C_n}{\partial\omega}^T\right|_{\omega=\hat{\omega}} \quad (11)$$

$$\neq \left.\frac{\partial C}{\partial\omega}\frac{\partial C}{\partial\omega}^T\right|_{\omega=\hat{\omega}}. \quad (12)$$

Letting $J = \begin{bmatrix} \frac{\partial C_1}{\partial\omega} & \frac{\partial C_2}{\partial\omega} & \dots & \frac{\partial C_N}{\partial\omega} \end{bmatrix}$, yields

$$G = \left.\frac{1}{N}J^TJ\right|_{\omega=\hat{\omega}}. \quad (13)$$

We notice that H in Equation (10) is formed by summing over N per-example Hessian matrices, and that G in Equation (11) is formed by summing over N per-example OPG matrices. We also note that H can be obtained by differentating the cost function directly, whereas this property does not hold for G as seen by (12). Finally, we note that G can be written as a per-example cost Jacobian matrix product (13).

In order to proceed, we now need to consider three complications regarding gradients and Hessians in TensorFlow: the limitations of TensorFlow's built-in `tf.hessians()` function is discussed in Section 3.3, per-example gradients will be discussed in Section 3.1, and gradient representation will be discussed in Section 3.2.

---

[2]The notation used means that $H_{i,j} = \left.\frac{\partial^2 C}{\partial\omega_i\partial\omega_j}\right|_{\substack{\omega_i=\hat{\omega}_i \\ \omega_i=\hat{\omega}_i}}$

3.1 Per-Example Gradients

A per-example gradient of the cost function with respect to model parameters means to differentiate C_n in (10) and (11) with respect to model parameters for a single example n. However, when TensorFlow compute gradients (e.g. `tf.gradients()`) it performs back propagation, which never actually computes the per-example gradients, but instead directly obtains the sum of per-example gradients. To see what this means, consider the following dummy multiple linear regression model (for simplicity with no bias term):

```
In [1]: import tensorflow as tf
In [2]: import numpy as np
In [3]: W = tf.Variable([3., 4., 5., 2.])
In [4]: X = tf.placeholder('float32', shape=(None, 4))
In [5]: yhat = tf.tensordot(X, W, axes = 1)
In [6]: init = tf.global_variables_initializer()
In [7]: sess = tf.InteractiveSession()
In [8]: sess.run(init)
```

We have model parameters represented by the variable tensor `W` (`In [3]`), and we use the placeholder tensor `X` (`In [4]`) as the model input. For simplicity, we do not define a cost function here, but instead conduct several differentiation experiments directly on the scalar model function `yhat` (`In [5]`) with $N = 2$:

```
In [9]: sess.run(yhat, feed_dict={x:np.array([[1.,2.,3.,4.],
                                              [2.,3.,4.,5.]])})
Out [1]: array([34., 48.], dtype=float32)
```

We get back two values (`Out [1]`) corresponding to the two inner products as expected. We now take the gradient of the model function with respect to the model parameters for a single example:

```
In [10]: sess.run(tf.gradients(yhat, W),
                  feed_dict={X:np.array([[1.,2.,3.,4.]])})
Out [2]: [array([1., 2., 3., 4.], dtype=float32)]
```

We get back the per-example gradient as expected (`Out [2]`). We do the same for the second example:

```
In [11]: sess.run(tf.gradients(yhat, W),
                  feed_dict={X:np.array([[2.,3.,4.,5.]])})
Out [3]: [array([2., 3., 4., 5.], dtype=float32)]
```

But when we try to feed two examples:

```
In [12]: sess.run(tf.gradients(yhat, W),
                  feed_dict={X:np.array([[1.,2.,3.,4.],
                                         [2.,3.,4.,5.]])})
Out [4]: [array([3., 5., 7., 9.], dtype=float32)]
```

we notice that we do not get back two per-example gradients, but rather the sum of the two per-example gradients (`Out [4]`). The important observation is here that in order to obtain per-example gradients we seemingly need to run `tf.gradients()` once per example, which in turn is well known to be very

inefficient when N grows large. We will get back to this and discuss solutions in Sections (5.1) and 5.2).

3.2 Gradient Representation

In practice, the P model parameters are represented by a list of tensors (e.g. `[tf.Variable(),...]`) corresponding to the different layers of the model architecture. On the other hand, the Hessian matrix is only one `(P, P)`-shaped tensor (matrix) formed by every single variable element contained in the list of variable tensors.

When differentiating a function represented by a computational graph with respect to some variable(s) in that graph, the variable tensors we pass to the differentiation function (`tf.gradients()`) must be kept in their original form as upon defining the graph. One can still pass on the whole collection of variables as a list to get hold of the full gradient, but the result will not be a flat gradient vector – it will rather be a list of sub-gradients represented by multiple tensors. This means that in order to end up with the `(P, P)`-shaped Hessian matrix we want, we need to keep all the variables in a list during differentiation, and only afterwards reshape the result into the desired flat form.

3.2.1 Flattening of Gradients

To illustrate the concept of lists of sub-gradients vs. flat gradients, consider a dummy multinomial logistic regression model:

```
In [13]: import tensorflow as tf
In [14]: T1 = 64
In [15]: T2 = 32
In [16]: P = T1*T2 + T2 # Total number of model parameters
In [17]: W = tf.Variable(tf.ones((T1, T2)), 'float32')
In [18]: b = tf.Variable(tf.ones((T2,)), 'float32')
In [19]: params = [W, b]
In [20]: params
Out [5]: [<tf.Variable 'Variable...' shape=(64, 32) ...>,
          <tf.Variable 'Variable...' shape=(32,) ...]
In [21]: X = tf.placeholder(dtype='float32', shape=(None, T1))
In [22]: y = tf.placeholder(dtype='float32', shape=(None, T2))
In [23]: def model_fun(X, params):
             return tf.add(tf.matmul(X, params[0]), params[1])
In [24]: yhat_logits = model_fun(X, params)
In [25]: yhat = tf.nn.softmax(yhat_logits)
In [26]: def cost_fun(y, yhat_logits, params):
             return tf.losses.softmax_cross_entropy(y,
                                                    yhat_logits)
In [27]: cost = cost_fun(y, yhat_logits, params)
```

We thus have model parameters `W` (`In [17]`) and `b` (`In [18]`) with shapes `(T1, T2)` and `(T2,)`, respectively. We can differentiate the cost function represented by the tensor `cost` (`In [27]`) with respect to the individual variables, or the full list `params` (`In [19]`):

```
In [28]: tf.gradients(cost, W)
Out [6]: [<tf.Tensor 'gradients...' shape=(64, 32) ...>]
```

```
In  [29]:  tf.gradients(cost, b)
Out [7]:  [<tf.Tensor 'gradients...' shape=(32,) ...>]
In  [30]:  tf.gradients(cost, params)
Out [8]:  [<tf.Tensor 'gradients...' shape=(64, 32) ...>,
           <tf.Tensor 'gradients...' shape=(32,) ...>]
```

But if we try to reshape our parameters into a flat vector and then differentiate:

```
In  [31]:  params_flat = tf.concat([tf.reshape(W, [−1]), b],
                                     axis=0)
In  [32]:  params_flat
Out [9]:  <tf.Tensor 'concat...' shape=(2080,) ...>
In  [33]:  tf.gradients(cost, params_flat)
Out [10]:  [None]
```

We get `[None]` (`Out [10]`) because the new tensor `params_flat` (`In [31]`) is not part of the `cost` function graph (`In [27]`). We solve the issue by first differentiating with respect to the full list, and then flattening the resulting tensor:

```
In  [34]:  grads = tf.gradients(cost, params)
In  [35]:  grads
Out [12]:  [<tf.Tensor 'gradients_...' shape=(64, 32) ...>,
            <tf.Tensor 'gradients_...' shape=(32,) ...>]
In  [36]:  grads_flat = tf.concat([tf.reshape(grads[0],[−1]),
                                    grads[1]],
                                    axis=0)
In  [37]:  grads_flat
Out [13]:  <tf.Tensor 'concat...' shape=(2080,) dtype=float32>
```

3.3   The built-in TensorFlow function `tf.hessians()`

The fundamental question is, why can we not simply use the built-in TensorFlow function `tf.hessians()`? To see why, consider the following:

```
In  [38]:  tf.hessians(cost, params)
Out [14]:  [<tf.Tensor 'Reshape_...' shape=(64, 32, 64, 32) ...>,
            <tf.Tensor 'Reshape_...' shape=(32, 32) ...>]
```

We observe that we get back two tensors (`Out [14]`). Let us name the two H_U and H_L, respectively. Their respective shapes are (T1, T2, T1, T2) and (T2, T2). Firstly, if we reshape H_U into a (T1*T2, T1*T2)-shaped tensor, it will correspond to the full Hessian's upper block diagonal matrix $\in \mathbb{R}^{T_1 T_2 \times T_1 T_2}$. Secondly, the tensor H_L corresponds to the full Hessian's lower block diagonal matrix $\in$ $\mathbb{R}^{T_2 \times T_2}$. In other words, we get no information about the full Hessian's two off-diagonal block matrices $\in \mathbb{R}^{T_1 T_2 \times T_2}$ and $\mathbb{R}^{T_2 \times T_1 T_2}$. Equation (14) illustrates the concept.

$$H = \begin{bmatrix} H_U \in \mathbb{R}^{T_1 T_2 \times T_1 T_2} & ? \in \mathbb{R}^{T_1 T_2 \times T_2} \\ ? \in \mathbb{R}^{T_2 \times T_1 T_2} & H_L \in \mathbb{R}^{T_2 \times T_2} \end{bmatrix} \quad (14)$$

The two missing off-diagonal block matrices[3] represented by question marks

---

[3]The two matrices are equal up to transposition, since H is symmetric

in Equation (14) correspond to the partial derivatives involving variable entities from different tensors in the parameter list `params` (In [19]). The same principle applies for all `params` with `len(params) > 1`.

4 Approximate Hessian Eigendecompositions

In deep learning, the number of parameters P is usually so large that the full Hessian matrix will be prohibitively expensive to compute and store. In this section we present methodology addressing the issue in terms of approximate eigendecompositions based on K eigenpairs. Thus leading to a space complexity of $O(KP)$ rather than $O(P^2)$. As the time complexity is somewhat more involved, we leave this discussion for Sections 5.3 and 5.4.

4.1 Low-rank Approximation

A low-rank approximation of the Hessian matrix can be obtained by a eigendecomposition utilizing only K eigenpairs corresponding to the K largest eigenvalues of H (or G),

$$\widetilde{H} = Q\Lambda Q^T \in \mathbb{R}^{P \times P}, \qquad (15)$$

where $Q \in \mathbb{R}^{P \times K}$ is the matrix whose kth column is the eigenvector q_k of H (or G), and $\Lambda \in \mathbb{R}^{K \times K}$ is the diagonal matrix whose elements are the corresponding eigenvalues, $\Lambda_{kk} = \lambda_k$. We assume that the eigenvalues are algebraically sorted so that $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_K$.

4.2 Full-rank Approximation

A full-rank approximation of the Hessian matrix can be obtained by an extrapolation of its smallest eigenvalues. Assuming that $\lambda_{K+1} = \lambda_{K+2} = \ldots = \lambda_P = \widetilde{\lambda} > 0$, a full-rank approximation is given by

$$\widetilde{\widetilde{H}} = \widetilde{H} + \widetilde{\lambda}(I - QQ^T) \in \mathbb{R}^{P \times P}, \quad (16)$$

where $\widetilde{H}$ is the low-rank approximation (15) and where we have used that Q is an orthonormal basis. Details can be found in the Appendix 7.2. One particular choice for $\widetilde{\lambda}$ is to set it equal to the smallest eigenvalue in the low-rank approximation, e.g. $\widetilde{\lambda} = \lambda_K$.

5 Implementation

We will now address how to overcome the basic complications discussed in Sections 3.3, 3.1 and 3.2. The current section is divided into four parts: we first discuss how to compute the matrix H in Equation (10), and afterwards move on to the matrix G in Equation (11). Finally, in Sections 5.3 and 5.4 we address how to compute the aforementioned approximate eigendecompositions of both H and G.

5.1 Computing H

We compute the matrix H based on Hessian vector products [6]. A practial implementation of Equation (10) is essentially to form P Hessian vector products using the full set of basis vectors in $\mathbb{R}^P$. As a bonus, the resulting implementation can easily be paralellized because the columns of the Hessian matrix can be computed independently.

In the following we describe the essential parts of this paper's accompanying Python module `pyhessian` [11]. The Hessian vector product function `get_Hv_op(v)` can be described as follows:

1. Differentiates the cost function with respect to the model param-

eters contained in the list `params` and flattens the result

2. Performs elementwise multiplication of the flattened gradient and the vector v; `tf.stop_gradient()` ensures that v is treated as a constant during differentiation. This is important if the vector v is a function of the model parameters ω.

3. Differentiates the resulting elementwise vector product with respect to the model parameters (to get second order derivatives) and flattens the result. As this step can appear subtle, see the Appendix 7.1 for a rigorous derivation.

Note that the function `get_Hv_op(v)` uses the function `flatten()` which is based on the insights from Section 3.2.1 and the mathematical operation defined in Equation (5). Furthermore, we have defined a parallellized function `get_H_op()` to create the full Hessian matrix operation based on forming P Hessian vector products using `get_Hv_op(v)` for all v's in $\mathbb{R}^P$. The function `get_H_op()` sets up a parallel-lized operation using `tf.map_fn()` to get hold of all the P columns of the full Hessian matrix as defined in Equation (10). It works by applying `Hv_op` on all basis vectors in $\mathbb{R}^P$ represented by `tf.eye(self.P, self.P)`, where P is the total number of parameters in the model.

The important remark is now to realize that, by definition, the matrix H in Equation (10) is the sum of per-example Hessian matrices. It means that we can directly leverage from the fact that `tf.gradients()` returns the sum of per-example gradients discussed in Section 3.1. In other words, when we run the resulting `H_op` in a graph session, we get per-example Hessians (below `In [43]`) if we feed single examples, and the average of per-example Hessians if we feed more than one example. Thus, we can get a mini-batch (below using size `batch_size_H`) Hessian matrix if we feed a mini-batch (below `In [45]`), or we can obtain the full Hessian matrix directly by feeding the complete training set. However, to avoid excessive memory consumption for large N, we can sum over mini-batch Hessians and divide by the number of mini-batches (`In [46]` - `In [56]`):

```
In [39]: from pyhessian import HessianEstimator
In [40]: hest = HessianEstimator(...)
In [41]: H_op = hest.get_H_op()
In [42]: # Per-example
In [43]: H = sess.run(H_op, feed_dict={X:[X_train[0]],
                                       y:[y_train[0]]})
In [44]: # Mini-batch
In [45]: H = sess.run(H_op, feed_dict={X:X_train[:batch_size_H],
                                       y:y_train[:batch_size_H]})
In [46]: # Full
In [47]: B = int(N/batch_size_H)
In [48]: H = np.zeros((hest.P, hest.P), dtype='float32')
In [49]: for b in range(B):
In [50]:     H = H + sess.run(H_op,
In [51]:                      feed_dict={ \
In [52]:                      X: X_train[b*batch_size_H: \
In [53]:                              (b+1)*batch_size_H],
In [54]:                      y: y_train[b*batch_size_H: \
In [55]:                              (b+1)*batch_size_H]})
In [56]: H = H/B
```

Listing 1: Computing H

5.2 Computing G

Due to the inequality sign in Equation (12), the computation of G (unlike H) cannot exploit the implicit sum of gradients as discussed in Section 3.1. Instead, we will pursue another efficient technique based on parallized per-example gradients. Although the technique we present here has been reformulated and adapted to our needs, the original implementation idea is to our knowledge originating from the author of [2]. The OPG matrix operation function `get_G_op()` can be described as follows:

1. Creates `batch_size_G` copies of the model parameters

2. Splits the model input variable `X`, and the model output variable `y` into respective lists of `batch_size_G` elements

3. Creates a list of `batch_size_G` elements holding model output tensors resulting from evaluating the model function using respective inputs and parameter copies

4. Creates a list of `batch_size_G` elements holding cost output tensors resulting from evaluating the cost using respective labels, model outputs and parameter copies

5. Stacks up a flat per-example gradient tensor by paralell differentiation of per-example costs with respect to the corresponding model parameter copy

6. Forms the OPG matrix operation by matrix multiplication of per-example cost Jacobians as in Equation (13)

Note that the function `get_G_op()` utilizes the function `flatten()` which is based on the insights from Section 3.2.1 and the mathematical operation defined in Equation (5). Also note that the function `get_G_op()` requires itself to maintain redundant model parameter copies which size scale with `batch_size_G`. To avoid excessive memory consumption, we can sum over mini-batch OPGs and divide by the number of mini-batches (`In [64]` - `In [68]`):

```
In [57]: hest = HessianEstimator(..., batch_size_G)
In [58]: G_op = hest.get_G_op()
In [59]: # Per-example
In [60]: sess.run(G_op, feed_dict={X:[X_train[0]],
                                   y:[y_train[0]]})
In [61]: # Mini-batch
In [62]: sess.run(G_op, feed_dict={X:X_train[:batch_size_G],
                                   y:y_train[:batch_size_G]})
In [63]: # Full
In [64]: B = int(N/batch_size_G)
In [65]: G = np.zeros((hest.P, hest.P), dtype='float32')
In [66]: for b in range(B):
In [67]:     G = G + sess.run(G_op,
                   feed_dict={ \
                   X: X_train[b*batch_size_G:\
                            (b+1)*batch_size_G],
                   y: y_train[b*batch_size_G:\
                            (b+1)*batch_size_G]})
In [68]: G = G/B
```

Listing 2: Computing G

5.3 Computing Eigenpairs of H

The Lanczos iteration [5] can be applied to find $K < P$ eigenvalues (and corresponding eigenvectors) in $O(SNP)$ time and $O(KP)$ space when Pearlmutter's technique [6] is applied inside the iteration. Pearlmutter's technique can simply be described as a procedure based on two-pass backpropagations of complexity $O(NP)$ time and $O(P)$ space to obtain exact Hessian vector products without requiring to keep the full Hessian matrix in memory. The number S denotes the number of Lanczos iterations to reach convergence. Typically the convergence of the Lanczos algorithm will be fast enough so that S is orders of magnitude less than P.

Essentially, we select the number of eigenapirs K and use `LinearOperator` from the `scipy` distribution in combination with the Lanczos implementation `eigsh`, and setup the former to compute Hessian vector products using `get_Hv_op()` from `pyhessian` (In [69] - In [74]). The `LinearOperator` (In [83]) is initialized with a callback function `Hv()` (In [75] - In [82]) where the actual graph session is executed. The `eigsh` argument `which='LA'` (In [84]) ensures that the eigenpairs returned corresponds to the algebraically largest eigenvalues of H, and the lines `In [85] - In [87]` sorts the eigenpairs in descending eigenvalue order.

```
In [69]: from scipy.sparse.linalg import LinearOperator
In [70]: from scipy.sparse.linalg import eigsh
In [71]: K = 10
In [72]: hest = HessianEstimator(...)
In [73]: _v = tf.placeholder(shape=(hest.P,), dtype='float32')
In [74]: Hv_op = hest.get_Hv_op(_v)
In [75]: def Hv(v):
In [76]:     B = int(N/batch_size_H)
In [77]:     Hv = np.zeros((hest.P))
In [78]:     Bs = batch_size_H
In [79]:     for b in range(B):
In [80]:         Hv = Hv + sess.run(Hv_op,
                            feed_dict={X:X_train[b*Bs:\
                                        (b+1)*Bs],
                                       y:y_train[b*Bs:\
                                        (b+1)*Bs],
                                       _v:np.squeeze(v)})
In [81]:     Hv = Hv / B
In [82]:     return Hv
In [83]: H = LinearOperator((hest.P, hest.P), matvec=Hv,
                            dtype='float32')
In [84]: L, Q = eigsh(H, k=K, which='LA')
In [85]: sinds = np.flip(np.argsort(L))
In [86]: L = L[sinds]
In [87]: Q = Q[:,sinds]
```

Listing 3: Computing the Eigendecomposition of H

5.4 Computing Eigenpairs of G

For the OPG approximation (12), a slightly different approach can be applied. Since the OPG matrix can be written as a Jacobian matrix product (13), we get by the singular value decomposition that its eigenvectors will be the right singular vectors of the Jacobian, and its eigenvalues the squared singular values

$$NG = J^T J = (U\Sigma V^T)^T U\Sigma V^T$$
$$= V\Sigma U^T U\Sigma V^T$$
$$= V\Sigma^2 V^T \quad (17)$$

However, even the $N \times P$-dimensional Jacobian matrix J is prohibitively expensive to store. Luckily, mini-batches of J can easily be obtained, and so an incremental singular value decomposition [3, 4] can be applied to each mini-batch. The computational cost is thus $O(KNP)$ time and $O(KP)$ space. We select the number of eigenapirs K and use `IncrementalPCA` from the `sklearn` distribution (`In [88]` - `In [94]`). We then make use of J in (17) which is available via the func-

tion `get_J_op()`. The `get_J_op()` implementation is similar to `get_G_op()` except from that it excludes the final matrix product $J^T J$ and just returns J (`In 95`). Essentially, the rest of the details are tied to filling up the buffer `J` in a mini-batch fashion and also ensuring that the number of examples per mini-batch is large enough to support the selected K (`In [96]` - `In [103]`). Finally, the eigenpairs are computed based on (17) (`In [104]`).

```
In  [88]:  from sklearn.decomposition import IncrementalPCA
In  [89]:  K = 10
In  [90]:  Bs = batch_size_G
In  [91]:  hest = HessianEstimator(...)
In  [92]:  _N = int(np.ceil(K / Bs))
In  [93]:  assert N % _N != 0, 'N must be divisible by \
                              K/batch_size_G!'
In  [94]:  ipca = IncrementalPCA(n_components=K, batch_size=Bs*N,
                              copy=False)
In  [95]:  J_op = hest.get_J_op()
In  [96]:  J = np.zeros((Bs*_N, hest.P), dtype='float32')
In  [97]:  B = int(N/Bs)
In  [98]:  for b in range(B):
In  [99]:      s1 = Bs*(b%_N)
In  [100]:     s2 = Bs*(b%_N+1)
In  [101]:     J[s1:s2] = sess.run(J_op,
                              feed_dict={X: X_train[b*Bs:\
                                         (b+1)*Bs],
                              y: y_train[b*Bs:\
                                         (b+1)*Bs]})
In  [102]:         if (b+1) % _N == 0:
In  [103]:             ipca.partial_fit(J)
In  [104]: L, Q = np.float32(ipca.singular_values_**2 / N),\
                  np.float32(ipca.components_.T)
```

Listing 4: Computing the Eigendecomposition of G

5.5 Low-Rank Approximations

Given the implementations of the eigendecompositions of H and G in Sections 5.3 and 5.4, low-rank approximations can be computed by

```
Q@np.diag(L)@Q.T
```

Listing 5: Computing the Low-Rank Approximation

However, the primary motivation of this approximation is to avoid storing the full Hessian in memory. For example, if the intent is to evaluate

$y = x^T H x$ for $x \in \mathbb{R}^P$ then we can use

```
y = (x.T@Q)@np.diag(L)@(Q.T@x)
```

Listing 6: Implicit Application of the Low-Rank Approximation

where we have intentionally introduced superfluous parenthesis to illustrate that this expression avoids to form a full $P \times P$ matrix as an intermediate step.

11

5.6 Full-Rank Approximations

Given the implementations of H and G in Sections 5.3 and 5.4, full-rank approximations (using $\widetilde{\lambda} = \lambda_K$) can be computed by

```
Q@np.diag(L)@Q.T \
+ L[-1]*(np.eye(hest.P) \
    - Q@Q.T)
```

Listing 7: Computing the Full-Rank Approximation

Analogously to the low-rank example, if we wish to evaluate $y = x^T H x$ using the full-rank approximation with no intermediate formation of the full Hessian (nor I), we can use

```
y = (x.T@Q)@np.diag(L)@(Q.T@x)\
    + L[-1]*x.T@x \
    - L[-1]*(x.T@Q)@(Q.T@x)
```

Listing 8: Implicit Application of the Full-Rank Approximation

6 Summary and Concluding Remarks

We have presented a practical and efficient TensorFlow implementation for computing Hessian matrices in a deep learning context. The naive methods have a complexity of $O(NP^2)$ time and $O(P^2)$ space where N is the number of examples in the training set and P is the number of parameters in the model. Furthermore, we have introduced means for efficient computation of approximate Hessian eigendecompositions based on K eigenpairs, and shown how these can be applied as both low-rank and full-rank operators. The complexity of the approximate eigendecompositon of the Hessian is $O(SNP)$ and $O(KP)$ space where S represents the number of required Lanczos steps, whereas for the OPG approximation $O(KPN)$ time and $O(KP)$ space. The novelty of the naive methodology presented prominently lies in the implementation technique rather than in the asymptotic bound analysis point of view. As noted by [2], a naive method running back propagation N times with a mini-batch of size 1 is very inefficient because TensorFlow's back propagation implementation will not be able to exploit the parallelism of mini-batch operations by efficient matrix operation implementations. An usage example of the `pyhessian` module [11] applied on a feed-forward neural network TensorFlow model can be found in the included file `pyhessian_example.py`.

7 Appendix

7.1 Derivation of the Hessian Vector Product Implementation

Let y be the product between the Hessian matrix and an arbitrary vector v,

$$y = v^T H(\omega)|_{\omega=\hat{\omega}} \in \mathbb{R}^P, \tag{18}$$

$$H(\omega)|_{\omega=\hat{\omega}} = \begin{bmatrix} \frac{\partial^2 C(\omega)}{\partial^2 \omega_1} & \frac{\partial^2 C(\omega)}{\partial \omega_1 \partial \omega_2} & \cdots & \frac{\partial^2 C(\omega)}{\partial \omega_1 \partial \omega_p} \\ \frac{\partial^2 C(\omega)}{\partial \omega_2 \partial \omega_1} & \frac{\partial^2 C(\omega)}{\partial^2 \omega_2} & \cdots & \frac{\partial^2 C(\omega)}{\partial \omega_2 \partial \omega_p} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 C(\omega)}{\partial \omega_p \partial \omega_1} & \frac{\partial^2 C(\omega)}{\partial \omega_p \partial \omega_2} & \cdots & \frac{\partial^2 C(\omega)}{\partial^2 \omega_p} \end{bmatrix}_{\omega=\hat{\omega}} \in \mathbb{R}^{P \times P}, \quad v = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_P \end{bmatrix} \in \mathbb{R}^P, \tag{19}$$

where $C(\omega)$ is the scalar cost function (1), $\omega \in \mathbb{R}^P$ denotes the model parameter vector, and where $\hat{\omega}$ is the point in parameter space where we would like to evaluate the Hessian. The implementation of `get_Hv_op()` in `pyhessian` is as follows

```
y = flatten(tf.gradients(tf.math.multiply(flatten(tf.gradients(C, ŵ)),
                                          tf.stop_gradient(v)),
            params))
```

Listing 9: get_Hv_op() implementation

The inner-most differentiation (e.g. `tf.gradients()`) will return the gradient of the scalar function $C(\omega)$ evaluated at $\omega = \hat{\omega}$, which we will denote by $\nabla_\omega C(\omega)|_{\omega=\hat{\omega}} \in \mathbb{R}^P$. Furthermore, this gradient is multiplied element-wise by the vector v, and we get

$$\nabla_\omega C(\omega) \circ v|_{\omega=\hat{\omega}} = \begin{bmatrix} \frac{\partial C(\omega)}{\partial \omega_1} v_1 \\ \frac{\partial C(\omega)}{\partial \omega_2} v_2 \\ \vdots \\ \frac{\partial C(\omega)}{\partial \omega_P} v_P \end{bmatrix}_{\omega=\hat{\omega}}. \tag{20}$$

Therefore the first argument of the outer-most differentiation (e.g. `tf.gradients()`), will be a vector function rather than a scalar function as was not the case in the inner-most differentiation. Since differentiation of tensors in TensorFlow will evaluate to the <u>sum</u> of the gradients of the individual elements (of the tensor which is differentiated), we get

$$\nabla_\omega \nabla_\omega C(\omega) \circ v|_{\omega=\hat{\omega}} = \begin{bmatrix} \frac{\partial}{\partial \omega_1}\frac{\partial C(\omega)}{\partial \omega_1} v_1 + \frac{\partial}{\partial \omega_1}\frac{\partial C(\omega)}{\partial \omega_2} v_2 + \ldots + \frac{\partial}{\partial \omega_1}\frac{\partial C(\omega)}{\partial \omega_P} v_P \\ \frac{\partial}{\partial \omega_2}\frac{\partial C(\omega)}{\partial \omega_1} v_1 + \frac{\partial}{\partial \omega_2}\frac{\partial C(\omega)}{\partial \omega_2} v_2 + \ldots + \frac{\partial}{\partial \omega_2}\frac{\partial C(\omega)}{\partial \omega_P} v_P \\ \vdots \\ \frac{\partial}{\partial \omega_P}\frac{\partial C(\omega)}{\partial \omega_1} v_1 + \frac{\partial}{\partial \omega_P}\frac{\partial C(\omega)}{\partial \omega_2} v_2 + \ldots + \frac{\partial}{\partial \omega_P}\frac{\partial C(\omega)}{\partial \omega_P} v_P \end{bmatrix}_{\omega=\hat{\omega}} \tag{21}$$

$$= \begin{bmatrix} \frac{\partial^2 C(\omega)}{\partial^2 \omega_1} v_1 + \frac{\partial^2 C(\omega)}{\partial \omega_1 \partial \omega_2} v_2 + \ldots + \frac{\partial^2 C(\omega)}{\partial \omega_1 \partial \omega_P} v_P \\ \frac{\partial^2 C(\omega)}{\partial \omega_2 \partial \omega_1} v_1 + \frac{\partial^2 C(\omega)}{\partial^2 \omega_2} v_2 + \ldots + \frac{\partial^2 C(\omega)}{\partial \omega_2 \partial \omega_P} v_P \\ \vdots \\ \frac{\partial^2 C(\omega)}{\partial \omega_P \partial \omega_1} v_1 + \frac{\partial^2 C(\omega)}{\partial \omega_P \partial \omega_2} v_2 + \ldots + \frac{\partial^2 C(\omega)}{\partial^2 \omega_P} v_P \end{bmatrix}_{\omega=\hat{\omega}} \tag{22}$$

$$= \begin{bmatrix} \frac{\partial^2 C(\omega)}{\partial^2 \omega_1} & \frac{\partial^2 C(\omega)}{\partial \omega_1 \partial \omega_2} & \cdots & \frac{\partial^2 C(\omega)}{\partial \omega_1 \partial \omega_p} \\ \frac{\partial^2 C(\omega)}{\partial \omega_2 \partial \omega_1} & \frac{\partial^2 C(\omega)}{\partial^2 \omega_2} & \cdots & \frac{\partial^2 C(\omega)}{\partial \omega_2 \partial \omega_p} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 C(\omega)}{\partial \omega_p \partial \omega_1} & \frac{\partial^2 C(\omega)}{\partial \omega_p \partial \omega_2} & \cdots & \frac{\partial^2 C(\omega)}{\partial^2 \omega_p} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_P \end{bmatrix}_{\omega=\hat{\omega}} \tag{23}$$

$$= v^T H(\omega)|_{\omega=\hat{\omega}} \quad \square \tag{24}$$

7.2 Derivation of the Full-rank Approximation

The full eigendecomposition of the Hessian matrix can be written

$$H = Q_L \Lambda_L Q_L^T + Q_R \Lambda_R Q_R^T, \tag{25}$$

where $Q_L \in \mathbb{R}^{P \times K}$ is the matrix whose kth column is the eigenvector q_k of H, and $\Lambda_L \in \mathbb{R}^{K \times K}$ is the diagonal matrix whose elements are the corresponding eigenvalues, $\Lambda_{Lkk} = \lambda_k$. Further, $Q_R \in \mathbb{R}^{P \times (P-K)}$ is the matrix whose kth column is the eigenvector q_{K+k} of H, and $\Lambda_R \in \mathbb{R}^{(P-K) \times (P-K)}$ is the diagonal matrix whose elements are the corresponding eigenvalues, $\Lambda_{Rkk} = \lambda_{K+k}$. We assume that the eigenvalues are algebraically sorted so that $\lambda_1 \geq \lambda_2 \geq \lambda_K \geq \ldots \geq \lambda_P$. Assuming that the eigenvalues $\lambda_{K+1} = \lambda_{K+2} = \ldots = \lambda_P = \widetilde{\lambda} > 0$, we get

$$\widetilde{\widetilde{H}} = Q_L \Lambda_L Q_L^T + Q_R \widetilde{\lambda} I Q_R^T \tag{26}$$

$$= Q_L \Lambda_L Q_L^T + \widetilde{\lambda} Q_R Q_R^T. \tag{27}$$

Since the columns of Q_L and Q_R forms an orthonormal basis, it follows that $I = Q_L Q_L^T + Q_R Q_R^T$, and thus

$$\widetilde{\widetilde{H}} = Q_L \Lambda_L Q_L^T + \widetilde{\lambda}(I - Q_L Q_L^T). \tag{28}$$

Consequently, $\widetilde{\widetilde{H}}$ will be full-rank since all its eigenvalues are greater than zero. $\square$

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Rafal Jozefowicz, Yangqing Jia, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Mike Schuster, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. *TensorFlow: Large-scale machine learning on heterogeneous systems*, 2015. Software available from `https://tensorflow.org`

[2] Ian Goodfellow *Efficient Per-Example Gradient Computations* Technical report, Google, Inc, 2015, `http://arxiv.org/abs/1510.01799v2`

[3] A. Levy and M. Lindenbaum *Sequential Karhunen?Loeve Basis Extraction and its Application to Images* IEEE TRANSACTIONS ON IMAGE PROCESSING, VOL. 9, NO. 8, 2000, `http://www.cs.technion.ac.il/~mic/doc/skl-ip.pdf`

[4] H. Cardot and D. Degras *Online Principal Component Analysis in High Dimension: Which Algorithm to Choose?*, arXiv:1511.03688 [stat.ML], 2015 `https://arxiv.org/abs/1511.03688`

[5] , L. N. Trefethen and D. Bau III *Numerical Linear Algebra, pp. 243-284*, Siam, 1997

[6] Barak A. Pearlmutter *Fast Exact Multiplication by the Hessian* Neural Computation, 1993, `http://www.bcl.hamilton.ie/~barak/papers/nc-hessian.pdf`

[7] Ian Goodfellow and Yoshua Bengio and Aaron Courville *Deep Learn-*

ing. MIT Press, 2016, `http://www.deeplearningbook.org`

[8] Jorge Nocedal and Stephen Wright *Numerical Optimization* Springer Verlag, 2000, `http://www.bioinfo.org.cn/~wangchao/maa/Numerical_Optimization.pdf`

[9] Whitney K. Newey and Daniel Mc-Fadden *Handbook of Econometrics, Volume 4, Chapter 36 Large sample estimation and hypothesis testing, Pages 2111-2245* Elsevier, 1994, `https://www.sciencedirect.com/science/article/pii/S1573441205800054`

[10] *Python* `https://www.python.org`

[11] *pyhessian Python Module* `https://github.com/gknilsen/pyhessian.git`