The Report committee for Ge Gao certifies that this is the approved version of the following report:

hIPPYLearn: An inexact Newton-CG method for training neural networks with analysis of the Hessian

APPROVED BY

SUPERVISING COMMITTEE:

Omar Ghattas, Supervisor

Clint N. Dawson

hIPPYLearn: An inexact Newton-CG method for training neural networks with analysis of the Hessian

by

Ge Gao M.S Comp Sci, B.S

REPORT

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Computer Science

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2017

Dedicated to my father and mother, Kai and Yuehong.

Acknowledgments

Firstly, I would like to express my sincere gratitude to Prof. Omar Ghattas for his supervision of this report, for his motivation, patience, and support. His guidance helped me in through the whole process of preparing and writing this report. I could not have imagined having a better advisor for my master report.

Besides, I would like to thank Dr.Umberto Villa for his devoted time and continuous help. Dr.Villa is the author of hIPPYlib which is the most important library used for this report. He helped me get familiar with this topic and led me through those difficult problems.

I also want to thank Prof. Clint Dawson to be the reader of this report as well as his kind help through my graduate school study.

Thanks also go to Di(Larry) Liu as my teammate on this project for his commitment into this project. Working with him made me better understand this topic and work more efficient.

hIPPYLearn: An inexact Newton-CG method for training neural networks with analysis of the Hessian

Ge Gao M.S Comp Sci,
The University of Texas at Austin, 2017

Supervisor: Omar Ghattas

Neural networks, as part of deep learning, have become extremely popular due to their ability to extract information from data and to generalize it to new unseen inputs. Neural network has contributed to progress in many classic problems. For example, in natural language processing, utilization of neural network significantly improved the accuracy of parsing natural language sentences [11]. However, training complicated neural network is expensive and time-consuming. In this paper, we introduce more efficient methods to train neural network using Newton-type optimization algorithm.

Specifically, we use TensorFlow, the powerful machine learning package developed by Google [2] to define the structure of the neural network and the loss function that we want to optimize. TensorFlow's automatic differentiation capabilities allows us to efficiently compute gradient and Hessian of the loss function that are needed by the scalable numerical optimization algorithm implemented in hIPPYlib [12]. Numerical examples demonstrate the better performance of Newton method compared to Steepest Descent method, both in terms of number of iterations and computational time.

Another important contribution of this work is the study of the spectral properties of the Hessian of the loss function. The distribution of the eigenvalues of the Hessian, in fact, provides extremely valuable information regarding which directions in parameter space are well informed by the data.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Machine learning, according to the definition by Ton M. Mitchell [10], is

> A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E.

In common words, machine learning is a process that computer programs are able to extract information from same training data and apply this knowledge to new input data.

Neural Network, accurately referred as Artificial Neural Network, is the main type of model we use for this report. One definition of Neural Network is provided by inventor of one of the earliest neurocomputers, Maureen Caudill [7] as

> A computing system made up of a number of simple, highly interconnected processing elements, which process information by their dynamic state response to external inputs.

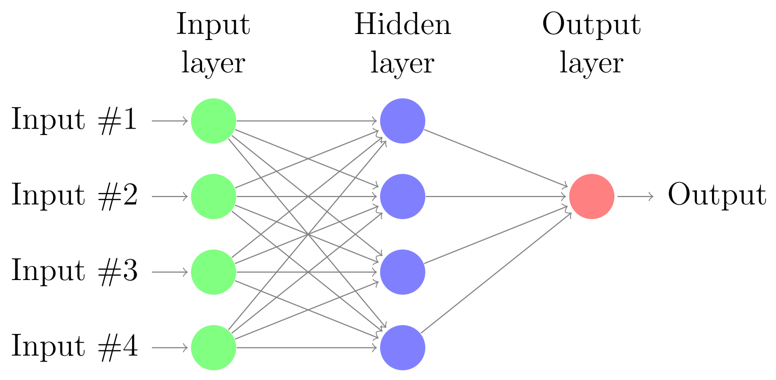

Figure 1.1: A simple neural network

So Neural Network is basically a network of computing elements accepting inputs and generating outputs based on the implementation of the network. Since usually these elements are not visible, they are usually called ”hidden units”. Also these elements are often grouped by layers and only units in adjacent layers are connected. Hence, these layers are often called “hidden layers”. Figure 1.1 shows a simple neural network with 3 layers. Input layer contains 4 nodes with color green. Hidden layer contains. Output layer contains 1 node with color red. 4 nodes with color blue.

Machine learning nowadays is changing people’s ordinary life and even the whole society, from Optical Character Recognition (OCR) to Natural Language Processing, from search engine to customized feed recommendation. Many of these problems are optimization problems in nature. For example, the main goal of the Optical Character Recognition is to minimize the difference between predicted characters and real characters. Given a testing dataset, training a OCR model is to minimize loss function over the training dataset. The main challenge here is that the model usually has very high complexity with thousands of hidden layers and millions of hidden units and the number of training data is also extremely huge. Therefore, the computing process, unsurprisingly, turns out to be very expensive and time-consuming.

TensorFlow is a open source machine learning software library developed by Google. It is widely used to develop and train neural networks. In this report, We used TensorFlow automatic differentiation capabilities. Thanks to automatic differentiation, when a user defines an operator in TensorFlow, the corresponding derivative is also defined. Furthermore, when a function is defined, TensorFlow automatically computes a dependency graph of all variables. Based on this graph and the chain rule, TensorFlow is capable of computing the derivative of the function by itself. Based on this functionality, we implemented problems with first order gradients and second order Hessian matrices that are needed to solve the optimization problems in this report.

For codes written for these report, TensorFlow and NumPy provide some basic math operations as well as automatic differentiation functionality. Based on these, two problem classes are implemented according to Problem API defined by algorithms picked from hIPPYlib. These problems also defines the structure of Neural Network including number of hidden layers and number of hidden units. Eventually, Neural Networks are trained by running solver program picked from hIPPYlib.

1.1 Motivation

Right now, the mainstream method used to train neural networks is steepest descent method. This method is relatively easy to implement and time consumed by each iteration is relatively cheap because it just needs to compute the gradient each time and apply it back. Also it turns out that for many problems, after a fairly large number of iterations, the accuracy of trained result is fairly good. On the contrary, int the numerical optimization literature there exist some more sophisticated methods that, instead of only using the first order derivative of the loss function, also use second order derivatives or even higher order derivatives. Using higher order derivative usually correspond to an increase of cost per iteration. However methods using higher order derivatives is likely to converge within a much smaller number of iterations, thus reducing the overall computing time ans efficiency.

Among those second order methods is Newton Method, which can significantly reduce number of iteration to meet same accuracy or converging requirement. However, the mainstream opinion of Newton Method in the machine learning community is that although it reduces the number of iteration, the time per iteration it so long that it makes this method not competitive compared with first order methods.

Here is where the scalable optimization algorithms implemented in hIPPYLib makes a difference. HIPPYlib implements state-of-the-art scalable algorithms for PDE-based deterministic and Bayesian inverse problems. In this report, we cherry-picked algorithms from hIPPYlib that have been proved to work extremely well in solving large scale optimization problems and apply them to problems of training neural networks.

1.2 Report organization

This report is composed of five chapters.

Chapter 2 introduces the optimization problem. Both steepest descent and higher order methods like Newton method are also covered in this chapter.

Chapter 3 describes the two test problems, namely MNIST problem and house price prediction problem, this report covers.

Chapter 4 presents the numerical results, which include a finite difference check for both the gradient and the Hessian matrix of the loss function. This chapter also includes comparison of convergence speed and computational time between Steepest Descent method and Inexact-NewtonCG method. Finally, this chapter talks about the computation of Hessian matrix and the eigenvalues of Hessian matrices.

Chapter 5 summarizes this report, highlights the key results and also covers potential work to do in the future.

1.3 Collaboration

I collaborated with Di(Larry) Liu on preparing for and writing this report. I implemented both gradient's and Hessian's finite difference check for some naive functions like y = x^2 and the first version of MNIST Neural

Network model with two layers. Larry refactored the model and did finite difference check based on this model. Larry implemented a three layer MNIST Neural Network model based on the two-layer one. I also finished the implementation of house price prediction model with both Steepest Descent method and Inexact NewtonCG methods as well as the finite difference check based on this model.

Afterwards, we did some independent research. Larry tried to apply stochastic methods to these models instead of using the whole dataset. I studied the special properties of the Hessian operator.

The eigenvalues of Hessian matrix are very important for the reasons below. First, it tells if the graph is locally convex. When all the eigenvalues are positive, then the function is “concave up” and a minimum is expected. If all eigenvalues are negative, then the function is “concave down”, and a maximum is expected. When eigenvalues are mixed with positive and negative, then there is a saddle point. The saddle point is neither a maximum nor a minimum, but training would “converge” because gradient on saddle point is zero. So with eigenvalues, we will be better aware of existence of saddle points. Also, larger eigenvalues mean the data strongly inform the parameter in the direction of the corresponding eigenvectors. Therefore, in practice, when a small subset of eigenvalues are significantly greater than the rest, one can reduce the dimension of the parameter space without losing much accuracy.

With respect to writing this report, I wrote most of chapter one and chapter two. Larry wrote most of chapter three and results of shared research in chapter four. Each of us wrote our own extended research results and analysis separately. We worked together on the rest of the report.

Chapter 2

Numerical Optimization Algorithms

This chapter introduces the optimization problem, Steepest Descent method and Newton method.

2.1 Optimization problem

> A computational problem in which the object is to find a solution in the feasible region which has the minimum (or maximum) value of the objective function.

Above is one definition of optimization problem [4]. In other words, solving an optimization problem is finding the best of all possible solutions. Optimization problems can be divided into continuous and discrete optimization problems. Classification problem, as one of the most focused category in machine learning problems, is an instance of discrete optimization problem. However, they can be relaxed and reformulated as continuous optimization problem.

2.2 Steepest Descent Method

The pseudo code of Steepest Descent method is shown below. Here w is a vector including all variables of the Neural Network model and w_0 is the initial guess of w. $J(w)$ is the cost function of the Neural Network depending on w, which is the target to be minimized in this optimization problem. And $g(w)$ is the gradient of J(w).

```
# Steepest Descent Method pseudo codes
num_iteration = 0
w = w_0
while J(w) > tolerance and num_iteration < max_iteration:
        d = - g(w)  #  compute negative gradient
        lambda = argmin(J(w + lambda * d)) # line search
        w = w + lambda * d
        num_iteration = num_iteration + 1
end of steepest descent method
```

From the codes above, we can see that we need to determine how far we want to walk, namely the value of λ, before recalculating the new direction. This value is also called step size or learning rate and it largely affects the convergence process. If step size is too large, then the optimizer might diverge, as shown on the left side. And if the value of step size is too small, then the convergence speed will suffer. For this reason, a proper value of λ generated by line search would make optimizer converge more efficiently. However, TensorFlow does not do line search. Instead, it requires the user to specify a fixed learning rate.

The above pseudo code uses exact line search to find the proper λ value. However in practice, this is usually very expensive and time-consuming. Instead, in the codes of this report, we used backtracking line search based on Armijo condition [8] to ensure a sufficient reduction of the loss function.

2.3 Newton Method

The w is a vector including all variables of the Neural Network model and w_0 as its initial guess. The $J(w)$ is the loss function depending on w, which is the target to be minimized. Newton Method is motivated by the quadratic approximation of J(w + s), which is denoted by Q(s). Here s is a small perturbation applied to w. Also g(w) is the gradient of J(w) depending on w and $\mathcal{H}(w)$ is the Hessian of J(w) depending on w.

$$Q(s) = J(w) + g(w)^T s + \frac{s^T \mathcal{H}(w)}{2} \tag{2.1}$$

If $\mathcal{H}(w)$ is positive definite, then $Q(s)$ has a unique minimum that satisfies

$$g(w) + \mathcal{H}(w)s = 0 \tag{2.2}$$

From this we can get the Newton iteration:

$$w = w - \mathcal{H}(w)^{-1} g(w) \tag{2.3}$$

So the pseudo code for Newton method is shown below.

```
# Newton Method pseudo codes
num_iteration = 0
```

```
w = w_0
while J(w) > tolerance and num_iteration < max_iteration:
        grad = g(w) # compute gradient
        hess = Hessian(J(w)) # compute Hessian
        d = -inverse(hess) * g
        w = w + d * lambda # lambda is chosen by line search
        num_iteration = num_iteration + 1
end of Newton method
```

Similar to the line search in Steepest Descent implementation, the line search in Newton method is an inexact line search for performance's sake. Line search is usually active at the beginning when we are very far from the optimum. As we get closer to the optimum, line search is satisfied with λ equal to one.

Newton method is usually able to converge within a significantly smaller number of iteration steps than Steepest Descent method. The main reason why it is not widely used in machine learning is the computation of Hessian matrix. The Hessian matrix is formally a dense large operator and therefore it is prohibitive to explicitly construct such operator or to solve linear system involving such operator using direct solvers. In fact, computing Hessian naively is extremely expensive because the cost of factorizing a dense matrix with n rows and columns is $O(n^3)$, which is not feasible for complex Neural Networks.

The inexact-NewtonCG (conjugate gradient) algorithm was developed to exactly overcome those issues. It is an extremely powerful method to deal with such large scale problems, as demonstrated by the fact that it is the method of choice in PDE constrained optimization and variational inverse problem.

Since CG only requires the ability to compute the action of the Hessian in a given direction, we do not have to explicitly compute the Hessian matrix. Hessian action can be efficiently computed via a forward and backward sweep of a linearized neural network. Also by solving the system inexactly (i.e. with an accuracy that becomes more and more tight) [6] as we approach the optimum we can drastically reduce the over all computational cost.

Chapter 3

Test Problems

3.1 Preamble

In this section, we present two test problems that we'll use for our numerical results. In the next chapter, we will train these neural networks to solve these problems with different optimization. One is the MNIST handwritten digit classification problem [9] based on MNIST database with 60000 training images and 10000 testing images. The other one is house price prediction problem [1].

To simplify our notation:

$$\mathbf{w} : \text{vector, parameter of neural network}$$

$$\mathbf{w}_0 : \text{vector, initial guess of } \mathbf{w}$$

$$J(\mathbf{w}) : \text{loss function at } \mathbf{w}$$

$$g(J(\mathbf{w})) : \text{gradient of function J at } \mathbf{w}$$

$$\mathcal{H}(J(\mathbf{w})) : \text{Hessian of function J at } \mathbf{w}$$

3.2 Regularization

Training a neural network is fundamentally an inverse problem: given some observation (i.e. the training data) we seek to find the unknown neural network parameters that provide the smallest misfit between the output of the neural network and the training data. Inverse problems are most often ill-posed, that is, not all directions in parameter space are well determined by the data. Such ill-posedness may cause difficulties in the convergence of the optimizer, and, more importantly may undermine the ability of the neural network to generalize well to predict new (unseen) data.

This lack of generalization is often referred as the overfitting problem, i.e., the trained network "memorizes" all the input and output.

In conclusion, regularization should be added in order to both avoid overfitting and to guarantee the well-posedness of the optimization problem. In this work, we will use Tikhonov regularization, which corresponds to a penalization of the l-2 norm of the parameters, $\frac{\beta}{2}\mathbf{w}^T\mathbf{w}$.

3.3 MNIST

The MNIST database of handwritten digits has a training set of 60000 examples, and a test set of 10000 examples. It is a subset of a larger set available from NIST. The digits have been size-normalized and centered in a fixed-size image [9].

To solve this problem, a fully connected network with 3 layers is created, including output layer, as shown in Figure 3.1. There are $28 \cdot 28 = 784$ nodes in input layer, 128 nodes in hidden layer and 10 nodes in output layer. Each of output nodes indicates the probability of the corresponding digit.

This network is defined as:

$$y_i = softmax\Bigg(w_2 \cdot (w_1 \cdot x_i + b_1) + b_2\Bigg),$$

$$\text{cross entropy loss} = \frac{1}{n}\sum_{1}^{n} \overline{y_i} \log y_i$$

$$w_j, b_j : \text{weight and bias for j layer}$$

$$x_i, y_i, \overline{y}_i : i\text{th input, output and ground truth}$$

If we collect all hte weights and biases in the vector $\mathbf{w} = (w_{1,1}, w_{1,2}, ..., w_{1,N_1}, b_{1,1}, ..., w_{2,1}...)$ and add L2 regularization, we can define our loss function as:

$$J(\mathbf{w}) = \text{cross entropy loss}(\mathbf{w}) + \frac{\beta}{2}\mathbf{w}^T\mathbf{w}$$

$$\beta : \text{l-2 regularization parameter}$$

In the following chapters, we will assume $J(\mathbf{w})$ as our loss function to optimize, and $\mathbf{w}$ is a vector with length l. For this fully connected network with 3 layers, total number of parameters to train is $784 \cdot 128 + 128 + 128 \cdot 10 + 10 = 101770$, i.e., $\mathbf{w}$ length is 101770.

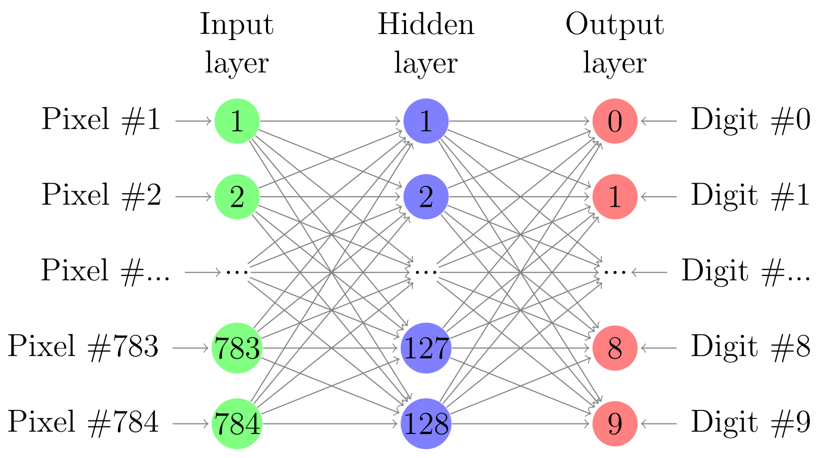

Figure 3.1: Fully connected network with 3 layers for MNIST problem

3.3.1 House price prediction

House price prediction is using 79 variables describing every aspect of residential homes to predict the house price. There are 1460 entries in this data set. The input data contains not only numbers but also labels (information strings). TWe need to binarize labels before solveing the optimization problem. For each label variable, we collect all unique labels and convert ith label to $e_i = (0, 0, 0, ..., 1, 0, ..., 0)^t$.

For example, we have data, [“red”, “blue”, “yellow”, “red”]. Binarize the labels and we get processed data $[(1, 0, 0)^T, (0, 1, 0)^T, (0, 0, 1)^T, (1, 0, 0)^T]$.

We binarize input data. Then, as a result of binarization process, number of input nodes increases to 835.

Similar to MNIST problem, we use a fully connected network with four layers. Input layer contains 835 nodes, first hidden layer contains 200 nodes, second hidden layer contains 200 nodes and output layer contains 1 node, the market value representing of the house.

This network is defined as:

$$y_i = w_3 \cdot \text{relu}(w_2 \cdot \text{relu}(w_1 \cdot x_i + b_1) + b_2) + b_3,$$

$$l_2 \text{ loss} = \sum_{i=1}^{n} \frac{\|y_i - \overline{y}_i\|^2}{n}$$

$$\text{relu} = \ln{(1 + \exp^x)}$$

$$w_j, b_j : \text{weight and bias for j layer}$$

$$x_i, y_i, \overline{y}_i : i\text{th x, y and ground truth}$$

The total number of parameters to train is $835 \cdot 200 + 200 + 200 \cdot 200 + 200 + 200 + 1 = 207601$. That is to say, the loss function $J(\mathbf{w})$ to optimize has input vector $\mathbf{w}$ with length 207601.

Chapter 4

Numerical results

4.1 Finite difference check

Before solving the problems, it's important to verify that our computation of gradient and Hessian is correct.

According to definition of directional derivative we have:

$$D_{\mathbf{w}^*} J(\mathbf{w}) = g(\mathbf{w})^T \mathbf{w}^* = \lim_{\epsilon \to 0} \frac{J(\mathbf{w} + \epsilon \mathbf{w}^*) - J(\mathbf{w})}{\epsilon}.$$

We then define the error between directional derivative with its approximates:

$$e = \left| \frac{\mathrm{J}(\boldsymbol{w} + \epsilon \boldsymbol{w}^*) - \mathrm{J}(\boldsymbol{w})}{\epsilon} - g(\boldsymbol{w})^T \boldsymbol{w}^* \right|.$$

If $\lim_{\epsilon \to 0} e = 0$, i.e., the finite difference approximation converges to the gradient, then our computation for gradient is correct. Similarly, for Hessian, we define the error as e:

$$e = \left\| \frac{g(\boldsymbol{w} + \epsilon \boldsymbol{w}^*) - g(\boldsymbol{w})}{\epsilon} - \mathcal{H}(\boldsymbol{w}) \boldsymbol{w}^* \right\|.$$

We did finite difference check on fully connected network with 3 layers for MNIST problem. Figure 4.1, and Figure 4.2 prove that our computation of gradient and Hessian is correct, in fact, the error e decays linearly with ϵ.

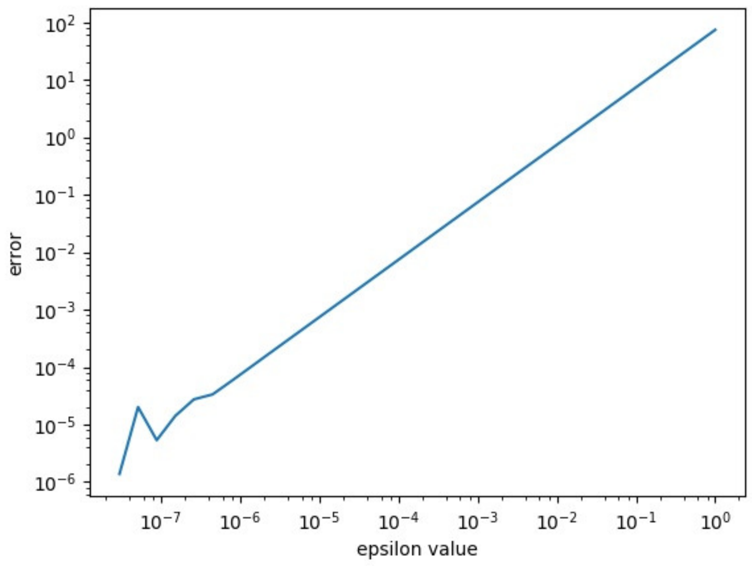

Figure 4.1: Finite difference check of gradient computation, x: ϵ, y: e

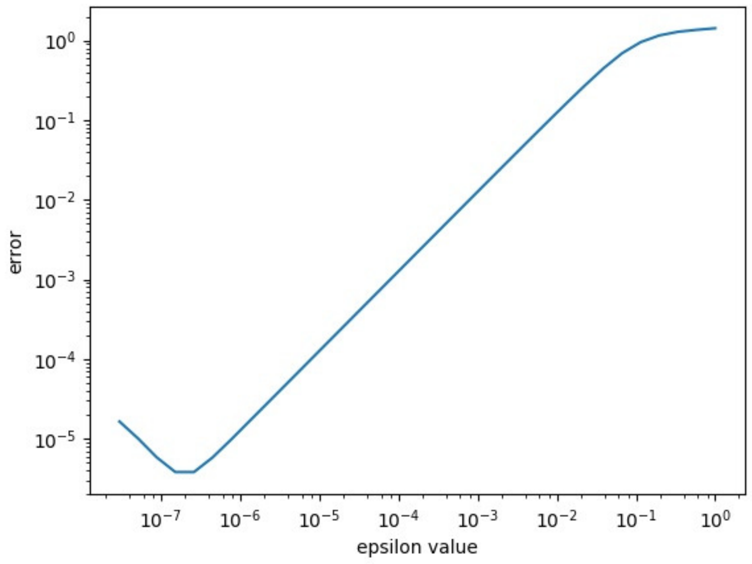

Figure 4.2: Finite difference check of Hessian computation, x: ϵ, y: e

4.2 Implementation abstraction

To abstract implementation, we created a "Problem" class, which is a simple interface for hIPPYlib to interact with TensorFlow. The method next_batch is useful when training network with stochastic Newton method.

```
class Problem:
    # compute the cost, which is J, at w
    cost(w)
    # compute the gradient of J at w
    gradient(w)
    # compute the Hessian of J at w * w_hat
    hessian_apply(w, w_hat)
    # change the input data of function J
    next_batch(num)
```

4.3 MNIST analysis

In this section, we used steepest descent algorithm and Newton CG algorithm to train the network. Parameters are listed as below:

```
input data size: 32768
beta: 1e-3
g_norm absolute tolerance: 1e-3
g_norm relative tolerance: 1e-3
max_iteration: 3000
```

The reason we chose 32768 for input data size is to facilitate the comparison with stochastic steepest descent/NewtonCG computation.

4.3.1 Convergence rate with respect to the number of nonlinear iterations

In this section we compared convergence rate with respect to the number of nonlinear iterations between gradient descent and NewtonCG method.

From Figure 4.3, it's easy to see NewtonCG converged faster than gradient descent in terms of number of nonlinear iterations. NewtonCG took 146 nonlinear iterations to finish optimization while gradient descent did not converge within the maximum number of iterations(3000).

Because the computation of cost per iteration is different between Newton and gradient descent, we also compared the time in solution in following subsection.

4.3.2 Cross validation accuracy

For NewtonCG method, the cross validation accuracy is 0.9246. Some results are shown below in Figure 4.4. In result, only the 4th digit is predicted as 5 by mistake. On the contrary, the cross validation accuracy for steepest descent is 0.5077 and some results are listed below in Figure 4.5. In result, multiple images predicted in wrong way.

This is because gradient descent didn't converge within the maximum number of iterations and got terminated.

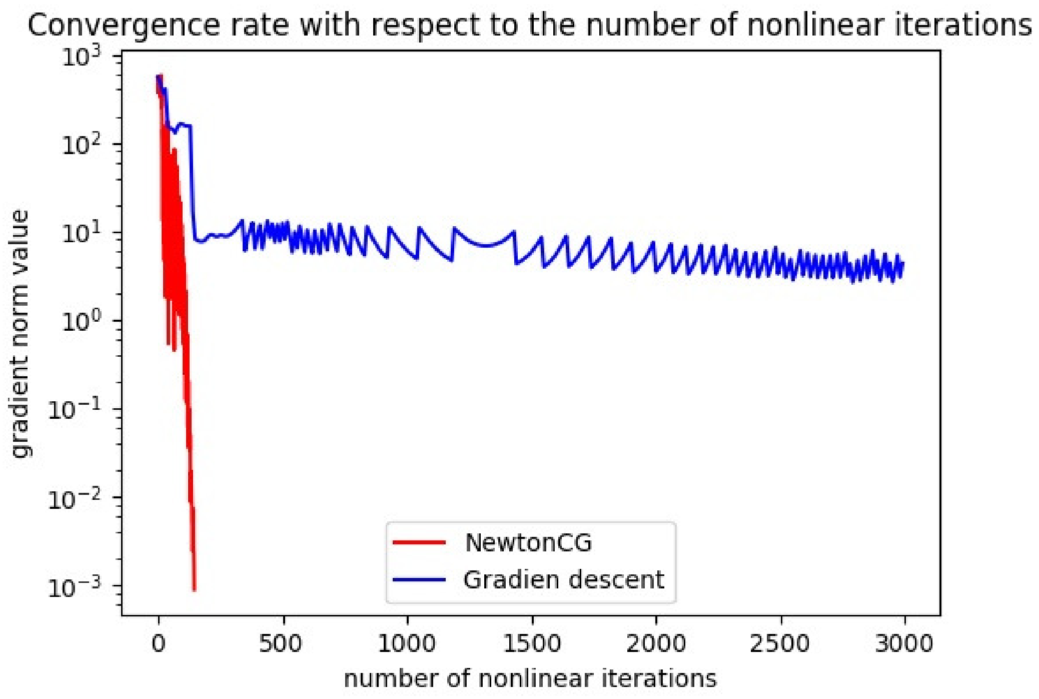

Figure 4.3: Convergence rate on MNIST problem network with respect to the number of nonlinear iterations

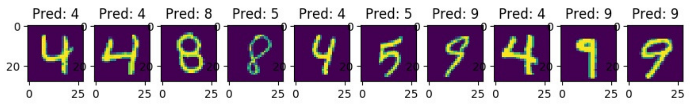

Figure 4.4: Cross validation accuracy demonstration for NewtonCG

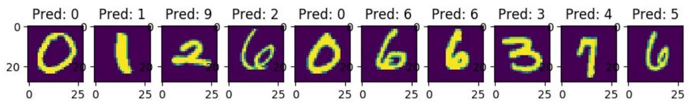

Figure 4.5: Cross validation accuracy demonstration for gradient descent

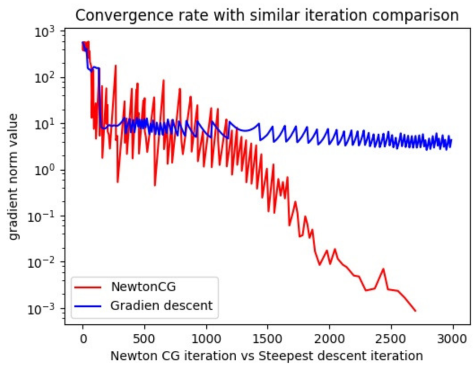

Figure 4.6: Convergence rate with respect to similar iteration comparison

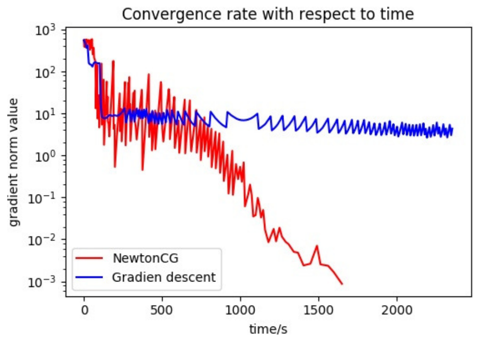

Figure 4.7: Convergence rate with respect to time comparison

4.3.3 Convergence time efficiency

In section 4.3.1, we demonstrated that Newton method converges in fewer steps than Gradient Descent method. But computation per iteration is different for each algorithm. Each nonlinear iteration Newton method solves a linear system involving the Hessian operator, which takes more time. So we investigated the convergence rate with respect to time. Specifically, we use CG to solve the linear system and each CG iteration requires a lineared forward and backward sweep of the network. So we compared each CG iteration of Newton with each iteration of gradient descent.

Figure 4.6 and Figure 4.7 demonstrate Newton methods converged faster in both metrics.

4.4 House price analysis

In this section, we used steepest descent algorithm and Newton CG algorithm to train the network. Parameters are listed as below:

```
input data size: 1460

beta: 1e-3

g_norm absolute tolerance: 1e-3

g_norm relative tolerance: 1e-3

max_iteration: 50000
```

4.4.1 Results

NewtonCG method took 396 iterations to train this network while gradient descent method reached maximum number of iterations before converging.

From Figure 4.8 and Figure 4.9, show that Newton method converges faster both in term of number of sweeps through the neural network and computational time.

4.5 Hessian Matrix and Eigenvalues

4.5.1 Computation of Hessian matrix

In our implementation of Inexact-NewtonCG method, considering the extreme large expenses of computing Hessian matrix, we computed the product of Hessian matrix and a vector. However, if we make the vector with only one

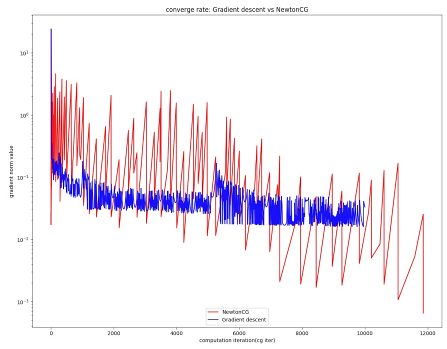

Figure 4.8: Convergence rate per iter

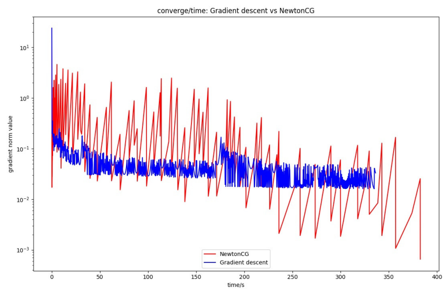

Figure 4.9: Convergence rate vs time

element as 1 and all others as 0, then the product of the Hessian matrix and this vector is actually a column of the Hessian matrix. And in this way, we can view the part of or the whole Hessian matrix.

In the pseudo codes below, w is a vector of all variables that the Hessian matrix depends on. And hessian_apply(w, w_hat) is a function computing the product of Hessian matrix and w_hat, which depends on w and w_hat.

```
hessian = initial empty matrix
total_number = size(w)
w_hat = zeros(total_number)
for i in range(total_number):
    w_hat[i] = 1
    hessian.add_column(hessian_apply(w, w_hat))
    w_hat[i] = 0
return hessian
```

4.5.2 Eigenvalues of Hessian matrix

Based on the previous subsection, we are capable of computing the whole Hessian matrix in theory. However in practice, for a non-trivial Neural Network model, the number of hidden units is at least hundreds of thousands, Therefore, to solve such a big matrix in memory and computing its eigenvalues is not feasible.

So we use eigenvalue solver from hIPPYlib. This solver is motivated by a randomized algorithm [3] for Hermitian eigenvalue problems and returns the k largest eigenvalues computed using the randomized algorithm, where k can be specified by the user.

The randomized solver requires inputs of a number of rows or columns in the matrix and a callable object that accepts a vector direction and returns the action of Hessian matrix in a given direction.

Figures 4.9, 4.10 and 4.11 show the eigenvalues of Hessian matrix MNIST two-layer model, MNIST three-layer model and house price prediction model. Each subplot uses different size of input data. Specifically, the whole dataset case is on the upper left corner and one half of the whole dataset case is on the upper right corner while a quarter dataset case is on the lower left corner and a 1/8 dataset case is on the lower right corner. A large eigenvalue means that parameter is strongly informed by the data in the direction of the corresponding eigenvector.

Figure 4.9 and Figure 4.10 use 8000, 4000, 2000 and 1000 input data

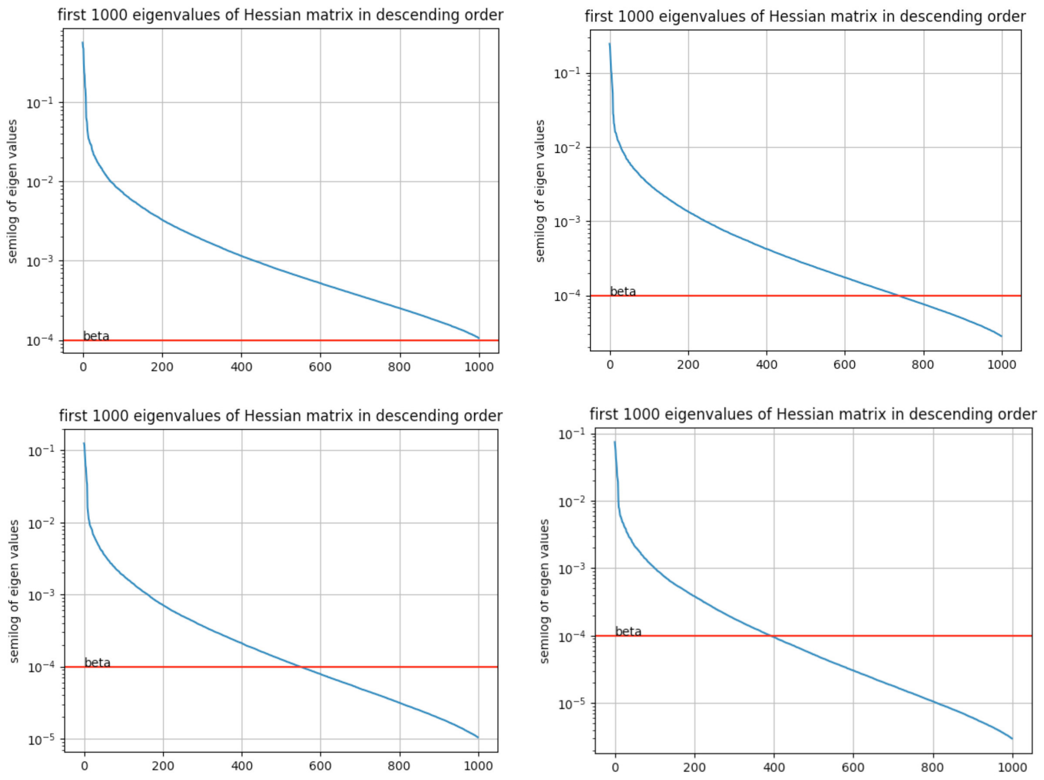

Figure 4.10: Eigenvalues of Hessian matrix of two layer Neural Network for MNIST

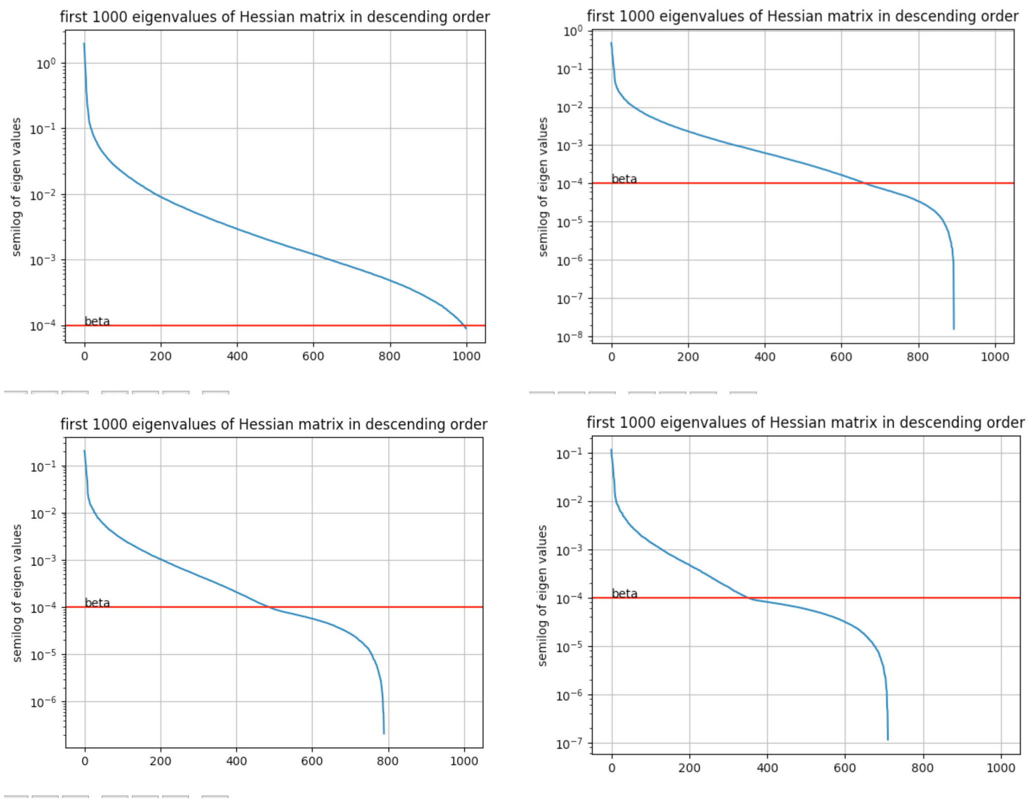

Figure 4.11: Eigenvalues of Hessian matrix of three layer Neural Network for MNIST

for each subplot. For the two-layer model, the total number of hidden units is 7850 and for the three-layer model, the total number is 101770. From the figures we can see that both lines go down sharply at the very beginning, which means only few directions in parameter space are well informed by the data. In fact, eigenvectors relative to eigenvalues that are above the red line in each subplot correspond to directions that are well informed by the data. This means,for example, the three-layer model with more than one hundred thousand hidden units, no more than 1000 eigenvectors are well informed by

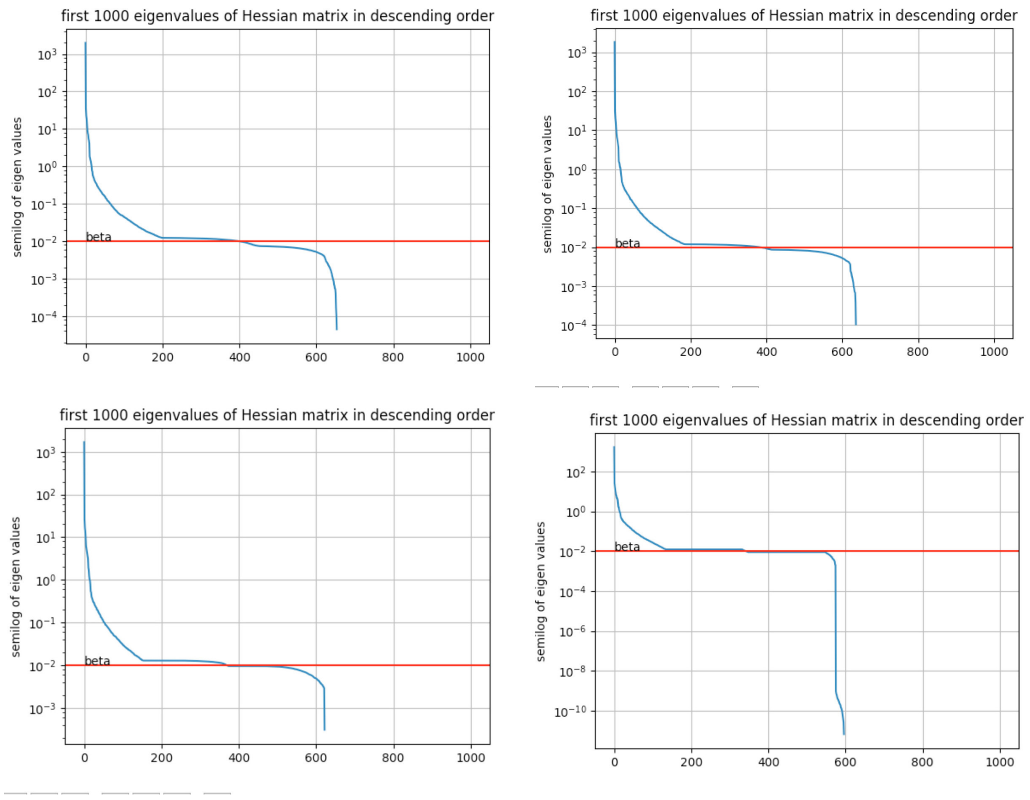

Figure 4.12: Eigenvalues of Hessian matrix of three layer Neural Network for house price prediction problem

the data. Figure 4.11 uses 1400, 700, 350 and 175 input data for each subplot. The number of hidden units in this Neural Network model is 207601. As we can see from the figure, the first 200 eigenvalues are significantly larger than the rest. Therefore, for the house prediction model, one can reduce the complexity of the Neural Network model without losing too much accuracy.

Chapter 5

Conclusion and future work

From the results of this report, we can see for all Neural Network models, Inexact-NewtonCG method from hIPPYlib has superior accuracy and performance compared with the Steepest Descent method, which is widely used as the default Neural Network training method. Since the Steepest Descent method is also the main Neural Network training method that TensorFlow uses, we believe that this is a good optimizer for TensorFlow to embed.

Furthermore, by analyzing the eigenvalues of Hessian matrix for a Neural Network model, one can reduce the complexity of Neural Network model and further improve training efficiency without losing too much accuracy.

Future work is how to choose the best β value without having to repeatedly run the training program.

Chapter 6

Source Code

Generic Problem class.

```
"""
File for generic GenericProblem class.
"""
class GenericProblem(object):
    """
    Generic problem model.
    """
    def __init__(self, input_size):
        """
          Init problem.
        """
        self.w_size = 0

        self.args = []
        self.input_size = input_size;

    def next_batch(self, batch_size):
        """
        Return the next batch of data with size = batch_size
        """
        pass

    def cost(self, feed_w):
        """
        Calculates the cost at position w, shape [n, 1].
        return: scalar value
        """
        pass

    def gradient(self, feed_w):
        """
        Calculates the gradients at position w, shape [n, 1].
        return: list of gradients according to w.
        """
        pass

    def hessian_apply(self, feed_w, feed_w_hat):
        """
        return: the action of the hessian (evaluated at w),
                in the direction of w_hat.
        """
        pass

    def cross_validation_accuracy(self, feed_w):
        """
        Get accuray from cross validation.
        """
        pass

    def peek(self, feed_w, num):
        """
        Peek the logits result.
        """
        pass

    def get_w_size(self):
        """
        Return the size of w.
        """
        return self.w_size

    def make_feed_dict_(self, feed_w):
        """
        make the feed input of network.
        """
        pos = 0
        res_dict = {}
        for arg in self.args:
            arg_shape = arg.get_shape().as_list()
            dim_0 = arg_shape[0]
            if len(arg_shape) > 1:
                dim_1 = arg_shape[1]
                res_dict[arg] = feed_w[pos : pos +\
                    int(dim_0*dim_1)].reshape(dim_0, dim_1)
                pos += int(dim_0*dim_1)
            else:
                res_dict[arg] = feed_w[pos : pos +\
                    int(dim_0)].reshape(dim_0,)
                pos += int(dim_0)
        return res_dict
```

TwoLayerMNISTProblem class.

```
"""
File for TwoLayerMNISTProblem model.
"""
from Problem import GenericProblem
from ProblemUtil import flatten_matrices

import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
import numpy as np
import matplotlib.pyplot as plt

NUM_CLASSES = 10

# The MNIST images are always 28x28 pixels.
IMAGE_SIZE = 28
IMAGE_PIXELS = IMAGE_SIZE * IMAGE_SIZE

TOTAL_COUNT = IMAGE_PIXELS * NUM_CLASSES + NUM_CLASSES

class TwoLayerNetwork(GenericProblem):
    def __init__(self, input_size):
        GenericProblem.__init__(self, input_size)
        self.w_size = TOTAL_COUNT
        self.mnist = input_data.read_data_sets("test_data", one_hot=True)

        self.batch_x = self.mnist.train.images[0:self.input_size]
        self.batch_y = self.mnist.train.labels[0:self.input_size]

        self.input_x = tf.placeholder(tf.float64, [None, IMAGE_PIXELS], name="x")
        self.input_y = tf.placeholder(tf.float64, [None, NUM_CLASSES], name="y")

        weight_matrix = tf.placeholder(tf.float64, [IMAGE_PIXELS, NUM_CLASSES], name="weight")
        bias = tf.placeholder(tf.float64, [NUM_CLASSES], name="b")

        self.pred = tf.matmul(tf.cast(self.input_x, tf.float64), weight_matrix) + bias
        self.args = [weight_matrix, bias]

        self.loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=self.pred,
                                                                            labels=self.input_y))
        self.grads = tf.gradients(self.loss, self.args)
        reshaped_grads = []
        for one_grad in self.grads:
            reshaped_grads.append(tf.reshape(one_grad, [tf.size(one_grad), 1]))
        self.flattened_grads = tf.reshape(tf.concat(reshaped_grads, 0), [1, TOTAL_COUNT])

        self.w_hat_var = tf.placeholder(tf.float64, [TOTAL_COUNT, 1], name="w_hat")
        self.product = tf.matmul(self.flattened_grads, self.w_hat_var)[0][0]
        self.hess = tf.gradients(self.product, self.args)

        self.sess = tf.InteractiveSession()
        init = tf.global_variables_initializer()
        self.sess.run(init)

    def next_batch(self, batch_size):
        if batch_size < self.mnist.train.num_examples:
            perm = np.arange(self.mnist.train.num_examples)
            np.random.shuffle(perm)
            self.batch_x = self.mnist.train.images[perm[0:batch_size]]
            self.batch_y = self.mnist.train.labels[perm[0:batch_size]]
        else:
            self.batch_x = self.mnist.train.images[0:batch_size]
            self.batch_y = self.mnist.train.labels[0:batch_size]
        #self.batch_x, self.batch_y = self.mnist.train.next_batch(batch_size)

    def cost(self, feed_w):
        feed_dict = self.make_feed_dict_(feed_w)
        feed_dict[self.input_x] = self.batch_x
        feed_dict[self.input_y] = self.batch_y
        return self.sess.run(self.loss, feed_dict=feed_dict)

    def gradient(self, feed_w):
        feed_dict = self.make_feed_dict_(feed_w)
        feed_dict[self.input_x] = self.batch_x
        feed_dict[self.input_y] = self.batch_y
        return flatten_matrices(self.sess.run(self.grads, feed_dict=feed_dict))

    def cross_validation_accuracy(self, feed_w):
        correct_prediction = tf.equal(tf.argmax(self.pred, 1), tf.argmax(self.input_y, 1))
        self.batch_x, self.batch_y = self.mnist.test.next_batch(self.input_size)
        accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float64))
        feed_dict = self.make_feed_dict_(feed_w)
        feed_dict[self.input_x] = self.batch_x
        feed_dict[self.input_y] = self.batch_y
        return self.sess.run(accuracy, feed_dict=feed_dict)

    def hessian_apply(self, feed_w, feed_w_hat):
        feed_dict = self.make_feed_dict_(feed_w)
        feed_dict[self.w_hat_var] = np.transpose([feed_w_hat])
        feed_dict[self.input_x] = self.batch_x
        feed_dict[self.input_y] = self.batch_y
        hess_res = self.sess.run(self.hess, feed_dict=feed_dict)
        return flatten_matrices(hess_res)

    def peek(self, feed_w, num):
        self.batch_x, self.batch_y = self.mnist.test.next_batch(num)
        feed_dict = self.make_feed_dict_(feed_w)
        feed_dict[self.input_x] = self.batch_x
        feed_dict[self.input_y] = self.batch_y
        res = self.sess.run(self.pred, feed_dict=feed_dict)
        f, a = plt.subplots(1, 10, figsize=(10, 1))
        for i in range(10):
            a[i].imshow(np.reshape(self.batch_x[i], (28, 28)))
            a[i].set_title("Pred: " + str(np.argmax(res[i])))

        f.show()
        plt.draw()
        plt.waitforbuttonpress()
        print "label    vs  prediction"
        for i in range(num):
            pred = str(np.argmax(res[i]))
            lab = str(np.argmax(self.batch_y[i]))
            print "[ " + lab +" ]    vs     [ " + pred +" ]"


"""
File for ThreeLayerMNISTProblem model.
"""
from TwoLayerMNISTProblem import TwoLayerNetwork
from ProblemUtil import flatten_matrices

import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
import numpy as np
```

```
NUM_CLASSES = 10

# The MNIST images are always 28x28 pixels.
IMAGE_SIZE = 28
HIDDEN_SIZE = 128
IMAGE_PIXELS = IMAGE_SIZE * IMAGE_SIZE

TOTAL_COUNT = IMAGE_PIXELS * HIDDEN_SIZE + HIDDEN_SIZE + HIDDEN_SIZE * NUM_CLASSES + NUM_CLASSES

class ThreeLayerNetwork(TwoLayerNetwork):
    def __init__(self, input_size):
        TwoLayerNetwork.__init__(self, input_size)
        self.w_size = TOTAL_COUNT
        self.mnist = input_data.read_data_sets("test_data", one_hot=True)

        self.batch_x = self.mnist.train.images[0:self.input_size]
        self.batch_y = self.mnist.train.labels[0:self.input_size]

        self.input_x = tf.placeholder(tf.float64, [None, IMAGE_PIXELS], name="x")
        self.input_y = tf.placeholder(tf.float64, [None, NUM_CLASSES], name="y")

        weight_matrix = tf.placeholder(tf.float64, [IMAGE_PIXELS, HIDDEN_SIZE], name="weight1")
        bias_hidden = tf.placeholder(tf.float64, [HIDDEN_SIZE], name="b1")

        hidden_layer_matrix = tf.placeholder(tf.float64, [HIDDEN_SIZE, NUM_CLASSES], name="weight2")
        bias_output = tf.placeholder(tf.float64, [NUM_CLASSES], name="b2")

        hidden_layer = tf.matmul(tf.cast(self.input_x, tf.float64), weight_matrix) + bias_hidden
        self.pred = tf.matmul(hidden_layer, hidden_layer_matrix) + bias_output
        self.args = [weight_matrix, bias_hidden, hidden_layer_matrix, bias_output]

        self.loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=self.pred,
                                                                          labels=self.input_y))
        self.grads = tf.gradients(self.loss, self.args)
        reshaped_grads = []
        for one_grad in self.grads:
            reshaped_grads.append(tf.reshape(one_grad, [tf.size(one_grad), 1]))
        self.flattened_grads = tf.reshape(tf.concat(reshaped_grads, 0), [1, TOTAL_COUNT])

        self.w_hat_var = tf.placeholder(tf.float64, [TOTAL_COUNT, 1], name="w_hat")
        self.product = tf.matmul(self.flattened_grads, self.w_hat_var)[0][0]
        self.hess = tf.gradients(self.product, self.args)

        self.sess = tf.InteractiveSession()
        init = tf.global_variables_initializer()
        self.sess.run(init)


"""
File for util functions for Probel model.
"""
import numpy as np

def flatten_matrices(lst):
    """
    Flatten one list of matrics to an 1D array.
    """
    res = np.random.rand(0,)
    for one_matrix in lst:
        res = np.concatenate([res, one_matrix.flatten()])
    return res


def vector_to_matrices(vec, lst):
    """
    Reconstruct one list of matrics from an 1D array.
    """
    # check the total number match
    len_left = vec.shape[0]
    for one_shape in lst:
        len_left -= one_shape[0]*one_shape[1]
    if len_left != 0:
        raise Exception("the number of elements does not match!!")

    res = []
    curr = 0
    for one_shape in lst:
        offset = one_shape[0]*one_shape[1]
        res.append(vec[curr:curr+offset].reshape(one_shape[0], one_shape[1]))
        curr += offset
    return res


import numpy as np

class Regularization(object):
    def __init__(self, beta):
        """
        beta is the regularization parameter
        """
        self.beta = beta

    def cost(self, w):
        return .5*self.beta*np.inner(w,w)

    def gradient(self, w):
        return self.beta*w

    def hessian_apply(self, w, w_hat):
        return self.beta*w_hat


import numpy as np
import math

class Identity(object):
    def __init__(self):
        pass

    def __call__(self, x):
        return np.copy(x)

class CGSolver(object):
    """
    Solve the linear system A x = b using preconditioned conjugate gradient ( B preconditioner)
    and the Steihaug stopping criterion:
    - reason of termination 0: we reached the maximum number of iterations (no convergence)
    - reason of termination 1: we reduced the residual up to the given tolerance (convergence)
    - reason of termination 2: we reached a negative direction (premature termination due to not spd matrix)

    The stopping criterion is based on either
    - the absolute preconditioned residual norm check: || r^* ||_{B^{-1}} < atol
    - the relative preconditioned residual norm check: || r^* ||_{B^{-1}}/|| r^0 ||_{B^{-1}} < rtol
    where r^* = b - Ax^* is the residual at convergence and r^0 = b - Ax^0 is the initial residual.

    The operator A is set using the method setOperator(A).
    A must be callable, i.e. A(x) should return the action of A on the vector x

    The preconditioner B is set using the method setPreconditioner(B).
    B must be callable, i.e. B(r) should return the action of the precondtioner B on the vector r

    To solve the linear system A*x = b call self.solve(x,b).
    Here x and b are assumed to be numpy arrays.

    The parameter attributes allows to set:
    - rel_tolerance      : the relative tolerance for the stopping criterion
    - abs_tolerance      : the absolute tolerance for the stopping criterion
    - max_iter           : the maximum number of iterations
    - zero_initial_guess: if True we start with a 0 initial guess
                          if False we use the x as initial guess.
    - print_level        : verbosity level:
                          -1 --> no output on screen
                           0 --> only final residual at convergence
                                 or reason for not not convergence
    """

    reason = ["Maximum Number of Iterations Reached",
              "Relative/Absolute residual less than tol",
              "Reached a negative direction"
              ]
    def __init__(self):
        self.parameters = {}
        self.parameters["rel_tolerance"] = 1e-9
        self.parameters["abs_tolerance"] = 1e-12
        self.parameters["max_iter"]       = 1000
        self.parameters["print_level"] = 0

        self.A = None
        self.B = Identity()
        self.converged = False
        self.iter = 0
        self.reasonid = 0
        self.final_norm = 0

    def setOperator(self, A):
        """
        Set the operator A.
        """
        self.A = A


    def setPreconditioner(self, B):
        """
        Set the preconditioner B.
        """
        self.B = B

    def solve(self,b):
        """
        Solve the linear system Ax = b
        """
        self.iter = 0
        self.converged = False
        self.reasonid  = 0

        betanom = 0.0
        alpha = 0.0
        beta = 0.0

        r = np.copy(b)
        x = np.zeros_like(b)

        d = self.B(r)

        nom0 = np.inner(d,r)
        nom = nom0

        if self.parameters["print_level"] == 1:
            print " Iterartion : ", 0, " (B r, r) = ", nom

        rtol2 = nom * self.parameters["rel_tolerance"] * self.parameters["rel_tolerance"]
        atol2 = self.parameters["abs_tolerance"] * self.parameters["abs_tolerance"]
        r0 = max(rtol2, atol2)

        if nom <= r0:
            self.converged  = True
            self.reasonid   = 1
            self.final_norm = math.sqrt(nom)
            if(self.parameters["print_level"] >= 0):
                print self.reason[self.reasonid]
                print "Converged in ", self.iter, " iterations with final norm ", self.final_norm
            return
        z = self.A(d)
        den = np.inner(z,d)

        if den <= 0.0:
            self.converged = True
            self.reasonid = 2
            x += d
            r -= z
            z = self.B(r)
            nom = np.inner(r, z)
            self.final_norm = math.sqrt(nom)
            if(self.parameters["print_level"] >= 0):
                print self.reason[self.reasonid]
                print "Converged in ", self.iter, " iterations with final norm ", self.final_norm
            return x

        # start iteration
        self.iter = 1
        while True:
            alpha = nom/den
            x += alpha*d    # x = x + alpha d
            r -= alpha*z    # r = r - alpha A d

            z = self.B(r)    # z = B^-1 r
            betanom = np.inner(r,z)

            if self.parameters["print_level"] == 1:
                print " Iteration : ", self.iter, " (B r, r) = ", betanom

            if betanom < r0:
                self.converged = True
                self.reasonid = 1
                self.final_norm = math.sqrt(betanom)
                if(self.parameters["print_level"] >= 0):
                    print self.reason[self.reasonid]
                    print "Converged in ", self.iter, " iterations with final norm ", self.final_norm
                break

            self.iter += 1
            if self.iter > self.parameters["max_iter"]:
                self.converged = False
                self.reasonid = 0
                self.final_norm = math.sqrt(betanom)
                if(self.parameters["print_level"] >= 0):
                    print self.reason[self.reasonid]
                    print "Not Converged. Final residual norm ", self.final_norm
                break

            beta = betanom/nom
            d = z + beta*d

            z = self.A(d)

            den = np.inner(d,z)

            if den <= 0.0:
                self.converged = True
                self.reasonid = 2
                self.final_norm = math.sqrt(nom)
                if(self.parameters["print_level"] >= 0):
                    print self.reason[self.reasonid]
                    print "Converged in ", self.iter, " iterations with final norm ", self.final_norm
                break

            nom = betanom

        return x


import numpy as np
import numpy.linalg as LA
import math
from cgsolver import CGSolver
import time
class ParametersInexactNewtonCG(object):
    """
    Parameter list for SteepestDescent.
    - rel_tolerance:  SteepestDescent will terminate at iteration k \
                      if norm(g(w_k))/norm(g(w_0)) < rel_tolerance
    - abs_tolerance:  SteepestDescent will terminate at iteration k \
                      if norm(g(w_k)) < abs_tolerance
    - max_iterations: Maximum number of SteepestDescent iterations
    - cg_coarse_tol:  Coarse tolerance for Conjugate Gradient linear solver
    - max_cg_iter:    Maximum number of CG iterations
    - c_armijo:       Armijo constant used for the line search (should be very \
                      small between 1e-5 and 1e-4)
    - max_backtracking_iter:
                      Maximum number of backtracking iterations
    - print_level:    2 -> Print all info
                      1 -> Print only final info
                      0 -> Silent
    - print_every_n_it:  if print_level=2 show the value of the cost J and its \
                         gradient every n iterations
    """
    def __init__(self):
        self.rel_tolerance = 1e-3
        self.abs_tolerance = 1e-3
        self.max_iterations = 1000
        self.cg_coarse_tol  = 0.5
        self.cg_max_iter    = 1000
        self.c_armijo = 1e-5
        self.max_backtracking_iter = 100
        self.print_level = 2
        self.print_every_n_it = 1


class InexactNewtonCG(object):
    """
    Class that implements the Inexact-Newton-CG algorithm with backtracking (Armijo condition)
    """

    def __init__(self, problem, regularization, parameters):
        """
        The constructor of this class takes as input:

        - problem: an object that implements the methods:
          1. problem.cost(w)      that takes as input a numpy array w of shape [n,1],
                                  and returns the value of the cost J(w)
                                  (return type is scalar value)
          2. problem.gradient(w)  that takes as input a numpy array w of shape [n,1],
                                  and returns the gradient of the cost J evaluated at w.
                                  (return type is a numpy array of shape [n,1])

          3. problem.hessian_apply(w, w_hat) takes as input two numpy arrays of shape [n,1]
                                             and returns the action of the Hessian (evaluated at w),
                                             in the direction w_hat.
                                             (return type is a numpy array of shape [n,1])

         - regularization: an object that implements the methods:
           1. regularization.cost(w)
           2. regularization.gradient(w)
           3. regularization.hessian_apply(w, w_hat)

         - parameters: an object of type ParametersInexactNewtonCG
        """
        self.problem        = problem
        self.regularization = regularization
        self.parameters     = parameters


        self.final_iter   = 0
        self.final_cost   = 0.
        self.final_norm_g = 0.
        self.cum_cg_iter  = 0
        self.converged  = True
        self.termination_reason = 0
        self.reasons = ["Converged to tolerance within the maximum number of iterations", #0
                        "Did not converge within the maximum number of iterations",       #1
                        "Backtracking failed"                                             #2
                       ]

    def solve(self, w0):
        """
        Solve the optimization problem using w0 as initial guess.
        Input:
         - w0: numpy array of shape [n,1]
        Output:
         - w:  numpy array of shape [n,1]
        """

        self.converged = False
        self.termination_reason = 1
        self.cum_cg_iter  = 0
        start = time.time()
        n = w0.shape[0]
        w = np.copy(w0)

        cost = self.problem.cost(w0)     + self.regularization.cost(w0)
        g    = self.problem.gradient(w0) + self.regularization.gradient(w0)
        norm_g = LA.norm(g)
        cost0 = cost
        norm_g0 = norm_g

        tol = min(self.parameters.abs_tolerance, self.parameters.rel_tolerance*norm_g)
        alpha = 1

        if (self.parameters.print_level == 2):
            print "{0:3} {1:12} {2:12}".format("it", "cost", "g_norm", "alpha", "cum_CG_iter")

        for k in range(self.parameters.max_iterations):
            if (self.parameters.print_level == 2) and ( k % self.parameters.print_every_n_it == 0):
                print "{0:3} {1:1.6e} {2:1.6e} {3:1.6e} {4:3}".format(k, \
                      cost, norm_g, time.time() - start, self.cum_cg_iter)

            #convergence check
            if(norm_g < tol):
                self.converged = True
                self.termination_reason = 0
                break

            #Newton step
            cgsolver = CGSolver()
            H = lambda w_hat: self.regularization.hessian_apply(w, w_hat) + \
                              self.problem.hessian_apply(w, w_hat)
            cgsolver.setOperator(H)
            cgsolver.parameters["rel_tolerance"] = min(self.parameters.cg_coarse_tol, \
                                                       math.sqrt(norm_g/norm_g0))
            cgsolver.parameters["max_iter"] = self.parameters.cg_max_iter
            cgsolver.parameters["print_level"] = -1
            d = cgsolver.solve(-g)
            self.cum_cg_iter  += cgsolver.iter


            d_inner_g = np.inner(d,g)
            alpha = 1.
            search = True
            i = 0
            while search and (i < self.parameters.max_backtracking_iter):
                i += 1
                cost_new = self.problem.cost(w+alpha*d) + self.regularization.cost(w+alpha*d)
                if cost_new < cost + alpha*self.parameters.c_armijo*d_inner_g:
                    cost = cost_new
                    w += alpha*d
                    search = False
                else:
                    alpha *= 0.5

            if search:
                self.termination_reason = 2
                break

            #prepare for next iteration
            g = self.problem.gradient(w) + self.regularization.gradient(w)
            norm_g = LA.norm(g)



        self.final_iter = k
        self.final_cost = cost
        self.final_norm_g = norm_g

        if self.parameters.print_level > 0:
            print "\n"
            print self.reasons[self.termination_reason]
            print "Initial cost: {0:1.6e}, Initial gradient norm: {1:1.6e}".format(cost0, norm_g0)
            print "Final   cost: {0:1.6e}, Final   gradient norm: \
                  {1:1.6e}".format(self.final_cost, self.final_norm_g)
            print "Newton Iterations: ", self.final_iter
            print "Comulate CG Iterations: ", self.cum_cg_iter
            print "\n"

        return w


import numpy as np
import numpy.linalg as LA
import time
```

```
class ParametersSteepestDescent(object):
    """
    Parameter list for SteepestDescent.
    - rel_tolerance:  SteepestDescent will terminate at iteration k \
                      if norm(g(w_k))/norm(g(w_0)) < rel_tolerance
    - abs_tolerance:  SteepestDescent will terminate at iteration k \
                      if norm(g(w_k)) < abs_tolerance
    - max_iterations: Maximum number of SteepestDescent iterations
    - c_armijo:       Armijo constant used for the line search (should be very \
                      small between 1e-5 and 1e-4)
    - max_backtracking_iter:
                      Maximum number of backtracking iterations
    - initial_alpha:  Initial value of alpha for line search
    - print_level:    2 -> Print all info
                      1 -> Print only final info
                      0 -> Silent
    - print_every_n_it:  if print_level=2 show the value of the cost J and its \
                         gradient every n iterations
    """
    def __init__(self):
        self.rel_tolerance = 1e-3
        self.abs_tolerance = 1e-3
        self.max_iterations = 3000
        self.c_armijo = 1e-5
        self.max_backtracking_iter = 100
        self.initial_alpha = 1.
        self.print_level = 2
        self.print_every_n_it = 1


class SteepestDescent(object):
    """
    Class that implements the Steepest Descent algorithm with backtracking (Armijo condition)
    """

    def __init__(self, problem, regularization, parameters):
        """
        The constructor of this class takes as input:

        - problem: an object that implements the methods:
          1. problem.cost(w)      that takes as input a numpy array w of shape [n,1],
                                  and returns the value of the cost J(w)
                                  (return type is scalar value)
          2. problem.gradient(w) that takes as input a numpy array w of shape [n,1],
                                  and returns the gradient of the cost J evaluated at w.
                                  (return type is a numpu array of shape [n,1])

         - parameters: an object of type ParametersStepestDescent
        """
        self.problem        = problem
        self.regularization = regularization
        self.parameters     = parameters


        self.final_iter    = 0
        self.final_cost    = 0.
        self.final_norm_g = 0.
        self.converged  = True
        self.termination_reason = 0
        self.reasons = ["Converged to tolerance within the maximum number of iterations", #0
                        "Did not converge within the maximum number of iterations",       #1
                        "Backtracking failed"                                              #2
                       ]

    def solve(self, w0):
        """
        Solve the optimization problem using w0 as initial guess.
        Input:
         - w0: numpy array of shape [n,1]
        Output:
         - w:  numpy array of shape [n,1]
        """
        start = time.time()
        self.converged = False
        self.termination_reason = 1
        n = w0.shape[0]
        w = np.copy(w0)

        cost = self.problem.cost(w0)     + self.regularization.cost(w0)
        g    = self.problem.gradient(w0) + self.regularization.gradient(w0)
        norm_g = LA.norm(g)

        cost0 = cost
        norm_g0 = norm_g

        tol = min(self.parameters.abs_tolerance, self.parameters.rel_tolerance*norm_g)
        alpha = self.parameters.initial_alpha

        if (self.parameters.print_level == 2):
            print "{0:3} {1:12} {2:12} {3:12}".format("it", "cost", "g_norm", "alpha")

        for k in range(self.parameters.max_iterations):
            if (self.parameters.print_level == 2) and ( k % self.parameters.print_every_n_it == 0):
                print "{0:3} {1:1.6e} {2:1.6e} {3:1.6e}".format(k, cost, norm_g, time.time() - start)

            #convergence check
            if(norm_g < tol):
                self.converged = True
                self.termination_reason = 0
                break

            #steepest descent direction
            d = -g
            d_inner_g = np.inner(d,g)
            search = True
            i = 0
            while search and (i < self.parameters.max_backtracking_iter):
                i += 1
                cost_new = self.problem.cost(w+alpha*d) + self.regularization.cost(w+alpha*d)
                if cost_new < cost + alpha*self.parameters.c_armijo*d_inner_g:
                    cost = cost_new
                    w += alpha*d
                    search = False
                else:
                    alpha *= 0.5

            if search:
                self.termination_reason = 2
                break

            #prepare for next iteration
            g = self.problem.gradient(w) + self.regularization.gradient(w)
            norm_g = LA.norm(g)
            alpha *= 2.



        self.final_iter = k
        self.final_cost = cost
        self.final_norm_g = norm_g

        if self.parameters.print_level > 0:
            print "\n"
            print self.reasons[self.termination_reason]
            print "Initial cost: {0:1.6e}, Initial gradient norm: \
                                 1:1.6e}".format(cost0, norm_g0)
            print "Final   cost: {0:1.6e}, Final   gradient norm: \
                                 {1:1.6e}".format(self.final_cost, self.final_norm_g)
            print "Iterations: ", self.final_iter
            print "\n"

        return w


import numpy as np
import sys

sys.path.append("../")
from optimizer import InexactNewtonCG, ParametersInexactNewtonCG, randomizedEigensolver
from Problem.ThreeLayerMNISTProblem import ThreeLayerNetwork
from Problem.regularization import Regularization
```

```
if __name__ == "__main__":
    problem        = ThreeLayerNetwork(32768)
    # beta is the regularization parameter.
    # The smaller is beta, the better the fit to the training data.
    # However, smaller beta will make harder to solve the optimization problem.
    beta = 1e-3
    regularization = Regularization(beta)
    w0 = np.random.rand(problem.get_w_size())
    parameters = ParametersInexactNewtonCG()
    parameters.print_every_n_it = 1
    solver = InexactNewtonCG(problem, regularization, parameters)
    w = solver.solve(w0)
    print "this is accuracy: "
    print problem.cross_validation_accuracy(w)
    problem.peek(w, 20)

    n = w.shape[0]
    k = 100
    p = 10
    Aop = lambda w_hat: problem.hessian_apply(w, w_hat)
    lmbda, U = randomizedEigensolver(Aop, n, k, p)

    print lmbda


import sys
sys.path.append("../")
from optimizer.inexactNewtonCG import InexactNewtonCG, ParametersInexactNewtonCG
import math
import numpy as np
import pandas as pd
import tensorflow as tf
from sklearn.preprocessing import LabelBinarizer
from Problem.regularization import Regularization
from Problem.ProblemUtil import flatten_matrices

TRAINING_EXAMPLES = 10
NUM_EPOCHS = 4000
HIDDEN_SIZE = 200
num_features = 835
TOTAL_COUNT = num_features*HIDDEN_SIZE + HIDDEN_SIZE + HIDDEN_SIZE*HIDDEN_SIZE + \
              HIDDEN_SIZE + HIDDEN_SIZE + 1

def encode(data_frame):
    data_encoded = []
    encoders = []
    for feature in data_frame:
        data_i = data_frame[feature]
        encoder = None
        if data_frame[feature].dtype == 'O':
            encoder = LabelBinarizer()
            encoder.fit(list(set(data_frame[feature])))
            data_i = encoder.transform(data_i)
        data_i = np.array(data_i, dtype=np.float32)
        data_encoded.append(data_i)
        encoders.append(encoder)
    return data_encoded


def normalize(data_frame_encoded):
    data = data_frame_encoded
    data = [np.log(tt + 1) for tt in data]
    return data


def batch_generator(data_frame_encoded):
    labels = data_frame_encoded[-1]
    data = data_frame_encoded[:-1]

    num_features = len(data)
    num_batches = len(data[0])
    for i in range(num_batches):
        batch_compiled = []
        for j in range(num_features):
            if type(data[j][i]) is np.ndarray:
                batch_compiled.extend(data[j][i])
            else:
                batch_compiled.extend([data[j][i]])
        yield batch_compiled, labels[i]

class Problem(object):
    def __init__(self):
        df_train = pd.read_csv('./train.csv', keep_default_na=False)
        df_train = df_train.drop(['Id'], 1)
        column_names = df_train.columns.values
        df_train_encoded = encode(df_train)
        df_train_encoded_normalized = normalize(df_train_encoded)
        batch_gen = batch_generator(df_train_encoded_normalized)
        all_examples = np.array([[np.array(b), l] for b, l in batch_gen])

        input_batches = np.array(all_examples[:, 0])
        len_batches = len(input_batches)
        input_batches = np.concatenate(input_batches)
        input_batches = np.reshape(input_batches, [len_batches, -1])

        output_labels = np.array(all_examples[:, 1]).astype(np.float32)
        print output_labels
        output_labels = np.reshape(output_labels, [1460, 1])

        self.input_layer = tf.Variable(input_batches, name='input')
        self.W1 = tf.placeholder(tf.float32, [num_features, HIDDEN_SIZE], name='W1')
        self.b1 = tf.placeholder(tf.float32, [HIDDEN_SIZE], name='b1')
        h1_layer = tf.add(tf.matmul(self.input_layer, self.W1), self.b1)

        self.W2 = tf.placeholder(tf.float32, [HIDDEN_SIZE, HIDDEN_SIZE], name='W2')
        self.b2 = tf.placeholder(tf.float32, [HIDDEN_SIZE], name='b2')
        h2_layer = tf.add(tf.matmul(h1_layer, self.W2), self.b2)

        self.W3 = tf.placeholder(tf.float32, [HIDDEN_SIZE, 1], name='W3')
        self.b3 = tf.placeholder(tf.float32, [1], name='b3')
        output_layer = tf.add(tf.matmul(h2_layer, self.W3), self.b3)
        self.labels = tf.Variable(output_labels, name='labels')


        self.args = [
            self.W1, self.b1,
            self.W2, self.b2,
            self.W3, self.b3
        ]

        self.loss = tf.squared_difference(output_layer, self.labels)
        reg_losses = tf.get_collection(tf.GraphKeys.REGULARIZATION_LOSSES)
        self.loss = self.loss + 0.01 * sum(reg_losses)
        self.loss = tf.reduce_mean(self.loss)

        self.grads = tf.gradients(self.loss, self.args)

        reshaped_grads = []
        for one_grad in self.grads:
            reshaped_grads.append(tf.reshape(one_grad, [tf.size(one_grad), 1]))
        self.flattened_grads = tf.reshape(tf.concat(reshaped_grads, 0), [1, TOTAL_COUNT])

        self.w_hat_var = tf.placeholder(tf.float32, [TOTAL_COUNT, 1], name="w_hat")
        self.product = tf.matmul(self.flattened_grads, self.w_hat_var)[0][0]
        self.hess = tf.gradients(self.product, self.args)

        self.sess = tf.Session()
        init = tf.global_variables_initializer()
        self.sess.run(init)

    def make_feed_dict(self, w):
        pos = 0
        res_dict = {}
        for arg in self.args:
            arg_shape = arg.get_shape().as_list()
            dim_0 = arg_shape[0]
            if len(arg_shape) > 1:
                dim_1 = arg_shape[1]
                res_dict[arg] = w[pos : pos + int(dim_0*dim_1)].reshape(dim_0, dim_1)
                pos += int(dim_0*dim_1)
            else:
                res_dict[arg] = w[pos : pos + int(dim_0)].reshape(dim_0,)
                pos += int(dim_0)
        return res_dict

    def cost(self, w):
        return self.sess.run(self.loss, feed_dict=self.make_feed_dict(w))

    def gradient(self, w):
        return flatten_matrices(self.sess.run(self.grads, feed_dict=self.make_feed_dict(w)))

    def hessian_apply(self, feed_w, feed_w_hat):
        feed_dict = self.make_feed_dict(feed_w)
        feed_dict[self.w_hat_var] = np.transpose([feed_w_hat])
        hess_res = self.sess.run(self.hess, feed_dict=feed_dict)
        return flatten_matrices(hess_res)


if __name__ == "__main__":
    problem = Problem()
    w0 = np.random.normal(0.0, 0.001, TOTAL_COUNT)
    # w0 = 0.001 * np.random.randn(TOTAL_COUNT)
    beta = 1e-3
    regularization = Regularization(beta)
    parameters = ParametersInexactNewtonCG()
    parameters.abs_tolerance = 1e-3
    parameters.print_every_n_it = 1
    solver = InexactNewtonCG(problem, regularization, parameters)
    w = solver.solve(w0)
    print w


import sys
sys.path.append("../")
from optimizer.steepestdescent import SteepestDescent, ParametersSteepestDescent
import math
import numpy as np
import pandas as pd
import tensorflow as tf
from sklearn.preprocessing import LabelBinarizer
from Problem.regularization import Regularization
from Problem.ProblemUtil import flatten_matrices

TRAINING_EXAMPLES = 10
NUM_EPOCHS = 4000
HIDDEN_SIZE = 200
num_features = 835
TOTAL_COUNT = num_features*HIDDEN_SIZE + HIDDEN_SIZE + HIDDEN_SIZE*HIDDEN_SIZE + \
              HIDDEN_SIZE + HIDDEN_SIZE + 1

def encode(data_frame):
    data_encoded = []
    encoders = []
    for feature in data_frame:
        data_i = data_frame[feature]
        encoder = None
        if data_frame[feature].dtype == 'O':
            encoder = LabelBinarizer()
            encoder.fit(list(set(data_frame[feature])))
            data_i = encoder.transform(data_i)
        data_i = np.array(data_i, dtype=np.float32)
        data_encoded.append(data_i)
        encoders.append(encoder)
    return data_encoded


def normalize(data_frame_encoded):
    data = data_frame_encoded
    data = [np.log(tt + 1) for tt in data]
    return data


def batch_generator(data_frame_encoded):
    labels = data_frame_encoded[-1]
    data = data_frame_encoded[:-1]

    num_features = len(data)
    num_batches = len(data[0])
    for i in range(num_batches):
        batch_compiled = []
        for j in range(num_features):
            if type(data[j][i]) is np.ndarray:
                batch_compiled.extend(data[j][i])
            else:
                batch_compiled.extend([data[j][i]])
        yield batch_compiled, labels[i]

class Problem(object):
    def __init__(self):
        df_train = pd.read_csv('./train.csv', keep_default_na=False)
        df_train = df_train.drop(['Id'], 1)
        column_names = df_train.columns.values
        df_train_encoded = encode(df_train)
        df_train_encoded_normalized = normalize(df_train_encoded)
        batch_gen = batch_generator(df_train_encoded_normalized)
        all_examples = np.array([[np.array(b), l] for b, l in batch_gen])

        input_batches = np.array(all_examples[:, 0])
        len_batches = len(input_batches)
        input_batches = np.concatenate(input_batches)
        input_batches = np.reshape(input_batches, [len_batches, -1])

        output_labels = np.array(all_examples[:, 1]).astype(np.float32)
        output_labels = np.reshape(output_labels, [1460, 1])

        self.input_layer = tf.Variable(input_batches, name='input')
        self.W1 = tf.placeholder(tf.float32, [num_features, HIDDEN_SIZE], name='W1')
        self.b1 = tf.placeholder(tf.float32, [HIDDEN_SIZE], name='b1')
        h1_layer = tf.add(tf.matmul(self.input_layer, self.W1), self.b1)

        self.W2 = tf.placeholder(tf.float32, [HIDDEN_SIZE, HIDDEN_SIZE], name='W2')
        self.b2 = tf.placeholder(tf.float32, [HIDDEN_SIZE], name='b2')
        h2_layer = tf.add(tf.matmul(h1_layer, self.W2), self.b2)

        self.W3 = tf.placeholder(tf.float32, [HIDDEN_SIZE, 1], name='W3')
        self.b3 = tf.placeholder(tf.float32, [1], name='b3')
        output_layer = tf.add(tf.matmul(h2_layer, self.W3), self.b3)
        self.labels = tf.Variable(output_labels, name='labels')


        self.args = [
            self.W1, self.b1,
            self.W2, self.b2,
            self.W3, self.b3
        ]

        self.loss = tf.squared_difference(output_layer, self.labels)
        reg_losses = tf.get_collection(tf.GraphKeys.REGULARIZATION_LOSSES)
        self.loss = self.loss + 0.01 * sum(reg_losses)
        self.loss = tf.reduce_mean(self.loss)

        self.grads = tf.gradients(self.loss, self.args)

        self.sess = tf.Session()
        init = tf.global_variables_initializer()
        self.sess.run(init)

    def make_feed_dict(self, w):
        pos = 0
        res_dict = {}
        for arg in self.args:
            arg_shape = arg.get_shape().as_list()
            dim_0 = arg_shape[0]
            if len(arg_shape) > 1:
                dim_1 = arg_shape[1]
                res_dict[arg] = w[pos : pos + int(dim_0*dim_1)].reshape(dim_0, dim_1)
                pos += int(dim_0*dim_1)
            else:
                res_dict[arg] = w[pos : pos + int(dim_0)].reshape(dim_0,)
                pos += int(dim_0)
        return res_dict

    def cost(self, w):
        return self.sess.run(self.loss, feed_dict=self.make_feed_dict(w))

    def gradient(self, w):
        return flatten_matrices(self.sess.run(self.grads, feed_dict=self.make_feed_dict(w)))


if __name__ == "__main__":
    problem = Problem()
    w0 = np.random.normal(0.0, 0.001, TOTAL_COUNT)
    # w0 = 0.001 * np.random.randn(TOTAL_COUNT)
    beta = 1e-3
    regularization = Regularization(beta)
    parameters = ParametersSteepestDescent()
    parameters.abs_tolerance = 1e-3
    parameters.print_every_n_it = 10
    parameters.max_iterations = 50000
    solver = SteepestDescent(problem, regularization, parameters)
    w = solver.solve(w0)
    print w
```

Codes for eigenvbalues computation and plot of house price prediction model

```
import sys
sys.path.append("../")
sys.path.append("../Problem")
from optimizer_house import InexactNewtonCG, ParametersInexactNewtonCG, randomizedEigensolver
import math
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import tensorflow as tf
import optimizer
from sklearn.preprocessing import LabelBinarizer
from Problem.regularization import Regularization
from ProblemUtil import flatten_matrices

TRAINING_EXAMPLES = 10
NUM_EPOCHS = 4000
HIDDEN_SIZE = 200
num_features = 835
TOTAL_COUNT = num_features*HIDDEN_SIZE + HIDDEN_SIZE + HIDDEN_SIZE*HIDDEN_SIZE + \
              HIDDEN_SIZE + HIDDEN_SIZE + 1

def encode(data_frame):
    data_encoded = []
    encoders = []
    for feature in data_frame:
        data_i = data_frame[feature]
        encoder = None
        if data_frame[feature].dtype == 'O':
            encoder = LabelBinarizer()
            encoder.fit(list(set(data_frame[feature])))
            data_i = encoder.transform(data_i)
        data_i = np.array(data_i, dtype=np.float32)
        data_encoded.append(data_i)
        encoders.append(encoder)
    return data_encoded


def normalize(data_frame_encoded):
    data = data_frame_encoded
    data = [np.log(tt + 1) for tt in data]
    return data


def batch_generator(data_frame_encoded):
    labels = data_frame_encoded[-1]
    data = data_frame_encoded[:-1]

    num_features = len(data)
    num_batches = len(data[0])
    for i in range(num_batches):
        batch_compiled = []
        for j in range(num_features):
            if type(data[j][i]) is np.ndarray:
                batch_compiled.extend(data[j][i])
            else:
                batch_compiled.extend([data[j][i]])
        yield batch_compiled, labels[i]

class Problem(object):
    def __init__(self):
        df_train = pd.read_csv('./train.csv', keep_default_na=False)
        df_train = df_train.drop(['Id'], 1)
        column_names = df_train.columns.values
        df_train_encoded = encode(df_train)
        df_train_encoded_normalized = normalize(df_train_encoded)
        batch_gen = batch_generator(df_train_encoded_normalized)
        all_examples = np.array([[np.array(b), l] for b, l in batch_gen])
        all_examples = all_examples[:175]

        input_batches = np.array(all_examples[:, 0])
        len_batches = len(input_batches)
        input_batches = np.concatenate(input_batches)
        input_batches = np.reshape(input_batches, [len_batches, -1])

        output_labels = np.array(all_examples[:, 1]).astype(np.float32)
        output_labels = np.reshape(output_labels, [175, 1])

        self.input_layer = tf.Variable(input_batches, name='input')
        self.W1 = tf.placeholder(tf.float32, [num_features, HIDDEN_SIZE], name='W1')
        self.b1 = tf.placeholder(tf.float32, [HIDDEN_SIZE], name='b1')
        h1_layer = tf.add(tf.matmul(self.input_layer, self.W1), self.b1)

        self.W2 = tf.placeholder(tf.float32, [HIDDEN_SIZE, HIDDEN_SIZE], name='W2')
        self.b2 = tf.placeholder(tf.float32, [HIDDEN_SIZE], name='b2')
        h2_layer = tf.add(tf.matmul(h1_layer, self.W2), self.b2)

        self.W3 = tf.placeholder(tf.float32, [HIDDEN_SIZE, 1], name='W3')
        self.b3 = tf.placeholder(tf.float32, [1], name='b3')
        output_layer = tf.add(tf.matmul(h2_layer, self.W3), self.b3)
        self.labels = tf.Variable(output_labels, name='labels')


        self.args = [
            self.W1, self.b1,
            self.W2, self.b2,
            self.W3, self.b3
        ]

        self.loss = tf.squared_difference(output_layer, self.labels)
        reg_losses = tf.get_collection(tf.GraphKeys.REGULARIZATION_LOSSES)
        self.loss = self.loss + 0.01 * sum(reg_losses)
        self.loss = tf.reduce_mean(self.loss)

        self.grads = tf.gradients(self.loss, self.args)

        reshaped_grads = []
        for one_grad in self.grads:
            reshaped_grads.append(tf.reshape(one_grad, [tf.size(one_grad), 1]))
        self.flattened_grads = tf.reshape(tf.concat(reshaped_grads, 0), [1, TOTAL_COUNT])

        self.w_hat_var = tf.placeholder(tf.float32, [TOTAL_COUNT, 1], name="w_hat")
        self.product = tf.matmul(self.flattened_grads, self.w_hat_var)[0][0]
        self.hess = tf.gradients(self.product, self.args)

        self.sess = tf.Session()
        init = tf.global_variables_initializer()
        self.sess.run(init)

    def make_feed_dict(self, w):
        pos = 0
        res_dict = {}
        for arg in self.args:
            arg_shape = arg.get_shape().as_list()
            dim_0 = arg_shape[0]
            if len(arg_shape) > 1:
                dim_1 = arg_shape[1]
                res_dict[arg] = w[pos : pos + int(dim_0*dim_1)].reshape(dim_0, dim_1)
                pos += int(dim_0*dim_1)
            else:
                res_dict[arg] = w[pos : pos + int(dim_0)].reshape(dim_0,)
                pos += int(dim_0)
        return res_dict
    def cost(self, w):
        return self.sess.run(self.loss, feed_dict=self.make_feed_dict(w))

    def gradient(self, w):
        return flatten_matrices(self.sess.run(self.grads, feed_dict=self.make_feed_dict(w)))

    def hessian_apply(self, feed_w, feed_w_hat):
        feed_dict = self.make_feed_dict(feed_w)
        feed_dict[self.w_hat_var] = np.transpose([feed_w_hat])
        hess_res = self.sess.run(self.hess, feed_dict=feed_dict)
        return flatten_matrices(hess_res)

class CallableMatrix(object):
    def __init__(self, p, w_val):
        self.problem = p
        self.w = w_val
    def __call__(self, w_hat):
        return self.problem.hessian_apply(self.w, w_hat)


if __name__ == "__main__":
    problem = Problem()
    w0 = np.random.normal(0.0, 0.001, TOTAL_COUNT)
    beta = 1e-2
    regularization = Regularization(beta)
    parameters = ParametersInexactNewtonCG()
    parameters.print_every_n_it = 1
    solver = InexactNewtonCG(problem, regularization, parameters)
    w = solver.solve(w0)

    matrix = CallableMatrix(problem, w)
    values, vectors = randomizedEigensolver(matrix, TOTAL_COUNT, 1000)

    plt.semilogy(values)
    plt.ylabel('semilog of eigen values')
    plt.title("first 1000 eigenvalues of Hessian matrix in descending order")
    plt.grid(True)
    plt.axhline(y=beta, color='r', label='beta')
    plt.annotate('beta', xy=(0, beta),xycoords='data', textcoords="offset points")
    plt.show()
```

Codes for computation of eigenvalues computations and plot of MNIST two layer model

```
import numpy as np
import matplotlib.pyplot as plt
import sys

sys.path.append("../")
from optimizer import InexactNewtonCG, ParametersInexactNewtonCG, randomizedEigensolver
from Problem.TwoLayerMNISTProblem import TwoLayerNetwork
from Problem.regularization import Regularization


class CallableMatrix(object):
    """docstring for CallableMatrix"""
    def __init__(self, p, w_val):
        self.problem = p
        self.w = w_val
    def __call__(self, w_hat):
        return self.problem.hessian_apply(self.w, w_hat)


if __name__ == "__main__":
    problem = TwoLayerNetwork(500)
    beta = 1e-4
    regularization = Regularization(beta)
    w0 = np.random.rand(problem.get_w_size())
    print problem.get_w_size()
    parameters = ParametersInexactNewtonCG()
    parameters.print_every_n_it = 1
    solver = InexactNewtonCG(problem, regularization, parameters)
    w = solver.solve(w0)

    total_count = problem.get_w_size()
    matrix = CallableMatrix(problem, w)
    values, vectors = randomizedEigensolver(matrix, total_count, 1000)
    print "the values are " + str(values)

    plt.semilogy(values)
    plt.ylabel('semilog of eigen values')
    plt.title("first 1000 eigenvalues of Hessian matrix in descending order")
    plt.grid(True)
    plt.axhline(y=beta, color='r', label='beta')
    plt.annotate('beta', xy=(0, beta),xycoords='data', textcoords="offset points")
    plt.show()
```

Codes for computation of eigenvalues computations and plot of MNIST two layer model

```
import numpy as np
import matplotlib.pyplot as plt
import sys


sys.path.append("../")
from optimizer import InexactNewtonCG, ParametersInexactNewtonCG, randomizedEigensolver
from Problem.ThreeLayerMNISTProblem import ThreeLayerNetwork
from Problem.regularization import Regularization


class CallableMatrix(object):
    """docstring for CallableMatrix"""
    def __init__(self, p, w_val):
        self.problem = p
        self.w = w_val
    def __call__(self, w_hat):
        return self.problem.hessian_apply(self.w, w_hat)


if __name__ == "__main__":
    problem = ThreeLayerNetwork(4000)
    beta = 1e-4
    regularization = Regularization(beta)
    w0 = np.random.rand(problem.get_w_size())
    parameters = ParametersInexactNewtonCG()
    parameters.print_every_n_it = 1
    solver = InexactNewtonCG(problem, regularization, parameters)
    w = solver.solve(w0)

    total_count = problem.get_w_size()
    matrix = CallableMatrix(problem, w)
    values, vectors = randomizedEigensolver(matrix, total_count, 1000)
    print "the values are " + str(values)

    plt.semilogy(values)
    plt.ylabel('semilog of eigen values')
    plt.title("first 1000 eigenvalues of Hessian matrix in descending order")
    plt.grid(True)
    plt.axhline(y=beta, color='r', label='beta')
    plt.annotate('beta', xy=(0, beta),xycoords='data', textcoords="offset points")
    plt.show()
```

Bibliography

[1] House prices: Advanced regression techniques. `https://www.kaggle.com/c/house-prices-advanced-regression-techniques`. Accessed: 2017-04-27.

[2] 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, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, 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 tensorflow.org.

[3] alko Nathan Per-Gunnar Martinsson and Joel A. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM review*, (53.2):217–288, 2011.

[4] Mikhail J. Atallah and Susan Fox, editors. *Algorithms and Theory of*

Computation Handbook. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1998.

[5] Christopher M. Bishop. *Pattern Recognition and Machine Learning.* Springer, 2006.

[6] Eisenstat Stanley C. and Homer F. Walker. Globally convergent inexact Newton methods. *SIAM Journal on Optimization 4.2*, pages 393–422, 1994.

[7] Maureen Caudill. Neural Network Primer: Part I. 1989.

[8] Haskell B. Curry. The method of steepest descent for nonlinear minimization problems. *Quart. Appl. Math*, 2(3):250–261, 1944.

[9] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[10] Thomas M. Mitchell. *Machine Learning.* McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.

[11] Collobert Ronan and Jason Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. *Proceedings of the 25th international conference on Machine ACM*, 2008.

[12] U. Villa, N. Petra, and O. Ghattas. hIPPYlib: an Extensible Software Framework for Large-scale Deterministic and Linearized Bayesian Inversion. 2016.

[13] Curtis R. Vogel. *Computational Methods for Inverse Problems.* Society for Industrial & Applied Mathematics (SIAM), January 2002.

Index

Vita

Ge Gao was born in Jiangsu province, China in 1993. Ge accomplished National First Prize in Chinese Mathematics Olympiad (Top 0.02 %) and Regional First Prize in Chinese Physics Olympiad (Top 0.05%).

Ge graduated from Huaiyin Middle School in 2011 and received Bachelor of Mathematics from the University of Texas at Austin in 2016.

Ge worked as intern for Fujitsu Communication Network, Epic System and Tableau Software from 2014 to 1016.

Ge is now a full-time core developer at Hudson River Trading LLC, developing high frequency trading system, internal tools and libraries.

Permanent address: gegao1017@gmail.com

This report was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.