Prediction of the Electrical Power Output of a Power Plant

Jeff Wirojwatanakul
pw1103@nyu.edu

December 12 2018

***Abstract*—To predict the electrical output of a power plant using linear regression. The parameters that affect the electrical power output(PO), which will be the target variable, are ambient temperature(AT), atmospheric pressure(AP), relative humidity(RH), and exhaust steam pressure(EP). This report explores different non-linear optimization techniques to optimize the loss function, in which the different techniques are studied empirically.**

I. Introduction

In brief, in a supervised learning context, we are given a set of training points, which we will try to find an appropriate learning function f(x) which maps $\mathcal{X} \mapsto \mathcal{Y}$. $\mathcal{X}$ is the input space, and $\mathcal{Y}$ is the output space. When our target variable y is continuous, this is known as a regression problem. After we have trained this model, we will then, hopefully, be able to determine the corresponding power output of a new data point. The unseen data points that are not in the training set comprises the test set [2].

Let us define the curve fitting function, or sometimes known as the hypothesis function in Machine Learning literature as:

$$f(\boldsymbol{x}, \boldsymbol{w}) = w_0x_0 + w_1x_1 + ...w_nx_n = \boldsymbol{w}^T\boldsymbol{X}$$

The vector $\boldsymbol{w}$ is known as the weight or parameter. For notation purposes, we define $x_0 = 1$. In a single variate case, w_0x_0 determines the y intercept. To determine the optimal weight such that $f(\boldsymbol{x}, \boldsymbol{w})$ predicts y well, we consider the following Loss or Cost function to be:

$$L(\boldsymbol{w}) = \sum_{i=1}^{m}(f(x_i, w) - y_i)^2 + \lambda\sum_{i=1}^{n} w_i^2$$

λ is considered the regularization parameter, which controls the relation between fitting the data points well and keeping the weights small to make our hypothesis function simple, avoiding overfitting. For this particular data set, varying λ does not really affect the solution, thus I have chosen λ to be zero. Prior to optimizing the cost function, I have normalized the input data [3].

I have particularly chosen four optimization methods: gradient based methods(Gradient Descent with and without line-search), Quasi-Newton (BFGS), Newton's method, and Non-linear Conjugate Gradient method.

II. Dataset

Professor Pinar Tufekci of Namik Kemal University has published a paper on prediction of electrical power of a power plant using different machine learning methods [1]. I have used her dataset, but have studied something different. I did not experiment on different machine learning algorithms, but rather explored different optimization techniques for linear regression. This data set comprises 9568 rows, each row containing the values for the input variables, (AT, AP, RH, EP), and the target variable(PO). In addition, the dataset has already been randomly shuffled.

III. Gradient Descent

One simple and well known algorithm for minimizing the cost fuction is the gradient descent. The idea behind the algorithm is to find the direction of the fastest increase of the loss function. The gradient descent then takes a negative step in the direction of the fastest increase, thus aims to reduce the loss as quickly as possible. After finding this direction, choosing the appropriate step size α is vital, as α will decide how fast the loss function will decrease [4].

α is considered as one of the hyper-parameters. In machine learning, hyper-parameters are what you tweak to maximize the performance of your algorithm.

Given a function f(x) that is differentiable at x_0, the gradient descent will be computed as follows:

Algorithm 1 Gradient Descent

```
1: procedure GD($f(x)$)
2:     $x \leftarrow 0$
3:     while iterations $<$ maxit or $|g(x)| > ftol$ do
4:         Determine $\alpha$
5:         $x = x - \alpha\nabla f(x)$
6:     return $x$
```

Note: In line 4, I determine the step size by manually picking different α to see the trade-offs between big and small α as well as using in-exact backtracking line-search method. I have used two variations of the backtracking line-search which can be written as follows:

Note: For different algorithms, p is determined differently. Usually the exact line search can be expensive and impractical to find, thus in-exact backtracking is preffered, which aims

Algorithm 2 (In-exact)Backtracking Line-Search [5]

```
Choose $\beta$ between (0,1)
Let $\alpha = 1$ at the start of each iteration
$\sigma \in (0,1)$
while $f(x - \alpha p) > f(x) - \alpha\sigma\nabla f(x)^T p$ do
    $\alpha = \beta\alpha$
```

to pick a step size that is neither too long or short. For both algorithm 2 and 3, I set β to 0.5.

Algorithm 3 Backtracking Line-Search [12]

```
Choose $\beta$ between (0,1)
Let $\alpha = 1$ at the start of each iteration
while $f(x - \alpha p) > f(x)$ do
    $\alpha = \beta\alpha$
```

The function we are interested in optimizing is:

$$L(\boldsymbol{w}) = \frac{1}{2m}(\sum_{i=1}^{m}(f(x_i, w) - y_i)^2 + \lambda\sum_{i=1}^{n} w_i^2)$$

The gradient of L is:

$$\nabla L(\boldsymbol{w}) = \frac{1}{m}(f(x, w) - y)^T x + \frac{\lambda}{m} w \tag{1}$$

Note: I am including $\frac{1}{2}$ to make the derivative easier. Thus, performing gradient descent on $L(\boldsymbol{w})$ gives us:

Repeat {

$$\boldsymbol{w} = \boldsymbol{w}(1 - \frac{\alpha\lambda}{m}) - \frac{\alpha}{m}(f(\boldsymbol{x}, \boldsymbol{w}) - \boldsymbol{y})^T\boldsymbol{x}$$

}

Although gradient descent is easily computed, its run time is $O(\frac{1}{k})$, where k is the number of iterations. This implies sub-linear convergence. The other downside of gradient descent is that it cannot perform on non-differentiable functions.

IV. Newton's Method

Aim: To minimize a function f(x). Unlike gradient descent, Newton's method uses the information of the Hessian, therefore, assumes that the second derivatives of f exists. Newton's method is based on minimizing over the quadratic approximation. A pure Newton step looks like:

$$x_+ = x - (\nabla^2 f(x))^{-1}\nabla f(x)$$

Recall our gradient from (1), by taking the differentiation of (1), we can obtain the Hessian of $L(\boldsymbol{w})$:

$$\nabla^2(L(\boldsymbol{w})) = \frac{1}{m}(x^T x + \lambda) \tag{2}$$

Now, we can formulate the Newton's method as:

Algorithm 4 Newton's Method

```
procedure NEWTON($f(x), maxit$)
    while ($i < maxit$ or $|g(x)| > ftol$) do
        $p = -H^{-1}g$
        Determine $\alpha$
        $x_+ = x + \alpha p$
```

Similar to gradient descent, I determined α by both performing a grid search, one common machine learning method for hyper-parameter tuning, and backtracking line search (Algorithm 2). There are many grid-search libraries offered; for example Python's sklearn package, which finds the optimal parameter, in this case, α. Grid search exhaustively searches through a manually specified subset to determine the optimal value; as an example, $\alpha \in [0.001, 1]$ [11]

Since Newton's method uses both the first and second derivatives, it performs better than gradient descent as it converges quadratically to a stationary point. However, it doesn't necessarily converge to a minimzer. Since it uses the second derivative, if for some iteration, $\nabla^2 f(x_k)$ is not invertible, Newton's method will not work. In addition, if the starting point is too far from the solution, Newton's method may diverge.

V. Quasi-Newton Method

Computing the hessian of a function and solving for $\nabla^2 f(x)p = -\nabla f(x)$ could be expensive, which the Newton method relies on. To avoid this issue, quasi-newton methods create an approximation of the Hessian at the start of the algorithm, call this matrix B.

Quasi-newton methods update x by:

$$x^+ = x + \alpha p$$

Where p can be computed by solving:

$$Bp = -g$$

Ideally, B is easily computed and solving for $Bp = -g$, can be done so in a reasonable matter. Different quasi-newton methods update B differently after each iteration. The basic idea is that B_{k-1} contains some information of the Hessian and we are trying to suitably update B_k [6]. B is intended to approximate the curvature of f as much as possible. I am using the Broyden-Fletcher-Goldfarb-Shanno(BFGS) update, which can be summarized as follows:

I have determined α using both exact and in-exact line search. The exact line search can be computed by setting the step size as:

$$a = \frac{-g_k^T p_k}{p_k^T H p_k} \tag{3}$$

Note: This exact line search follows Professor Wright and Professor Gill's notes [8], which α is computed differently than algorithm 2 and 3.

Checking for $y^T s > 0$ is a vital step of BFGS as it ensures that if B is positive definite, then B_+ will also be positive

Algorithm 5 Quasi-Newton method with BFGS update

```
procedure BFGS($f(x)$)
    Let B = Identity Matrix
    while ($i < maxit$ or $|g(x)| > ftol$) do
        Compute p: $Bp = p$
        Determine $\alpha$
        $x_+ = x + \alpha p$
        $y = g(x_+) - g(x)$
        $s = x_+ - x$
        if $y^T s > 0$ then
            $B_+ = \frac{1}{s^T Bs} Bss^T B + \frac{1}{y^T s} yy^T$
```

definite. Therefore, it is natural to set B_1 as the identity matrix.

Under certain conditions, quasi-newton methods follows a local superlinear convergence. It is supposed to update B so that it approximates the Hessian well. The drawback of quasi-newton methods is storing the approximate Hessian, which for big programs, can be problematic. Fortunately, this data set only has 9568 rows, which will not be that problematic.

VI. Nonlinear Conjugate Gradient Method

In many problems, gradient descent does not converge fast enough while Newton's method uses too much memory. Nonlinear Conjugate Gradient Method can be used for non-linear optimization, which does so without storing the information of a Hessian at each iteration and can obtain quadratic convergence. In CG, we assume that we are given a continuously differentiable function to optimize. For each iteration, CG stores only three vectors. In general, Nonlinear conjugate gradient follows the following recipe [10]:

Algorithm 6 Nonlinear Conjugate gradient method

```
 1: procedure CG($f(x), maxit$)
 2:     while ($i < maxit$ or $|g(x)| > ftol$) do
 3:         if i == 0 then
 4:             $p_i = -g_i$
 5:         else
 6:             Determine $\beta_{i-1}$
 7:             $p_i = -g_i + \beta_{i-1} p_{i-1}$
 8:         $x_i = x_{i-1} + \alpha * p_i$,
 9:         Where p is determined by a line-search
10:         $g_{i+1} = \nabla f(x_{i+1})$
```

Note that α can be determine using line search.

β is typically called the CG update parameter, where there have been many proposals on how to determine the update parameter. Fletcher and Reeves originally proposed the first nonlinear CG update back in 1964, which has been studied substantially, so I experimented with newer updates. In this paper, I experimented on the one proposed by Polak,Ribiere, and Polyak (PRP), the update proposed by Hager and Zhang in 2005, and the one by Dai and Yuan.

The PRP updates β as follows:

$$\beta^{PRP}_{k-1} = \frac{g^T_k (g_k - g_{k-1})}{g^T_{k-1} g_{k-1}}$$

For many computational problems, PRP is supposed to outperform CG method updated using the Fletcher-Reeves update since CG using the FR update takes too many tiny steps, a phenomena known as jamming. Due do this FR can make minimal progress [13]. The PRP uses a built-in restart feature, which occurs when $x_k - x_{k-1}$ becomes small. The small step results in $g_k - g_{k-1} \to 0$. Recall the update of p from line 7 of Algorithm 6. By using the PRP update, p is updated as:

$$p_k = -g_k + \frac{g^T_k (g_k - g_{k-1})}{g^T_{k-1} g_{k-1}} * p_{k-1}$$

Since $g_k - g_{k-1} \to 0$, p_k is basically restarted to $-g_k$, which avoids jamming.

Let y_k denote $g_k - g_{k-1}$

The Dai and Yuan updates β as:

$$\beta^{DY}_{k-1} = \frac{|g_k|^2}{p^T_{k-1} y_k}$$

The Hager and Zhang update:

$$\beta^{N}_{k-1} = \left(y_k - 2\frac{p_{k-1}|y_k|^2}{p^T_{k-1}(y_k)} \right)^T \frac{g_k}{p^T_{k-1}(y_k)}$$

The PRP can result in an uphill search direction. Unlike PRP, Dai and Yuan proved that the DY update ensures a descent direction under Wolfe line search while the Hager and Zhang update provides sufficient descent using any line search. The sufficient descent means:

$$g^T_k p_k \leq -c|g_k|^2, \ \forall k \geq 0$$

, for some positive constant c. It is advised that when a method fails to produce a descent direction, that one set p_k as $-g_k$.

Clarification: In some paper, the authors shift the iterations by one, thus β_{k-1} becomes β_k, $y_k = g_{k+1} - g_k$, and p_{k-1} becomes p_k, but the result will not change. For my notation, the iterate starts at zero.

All three of the update I have mentioned requires no evaluation of the hessian, which is preferred for larger problems. While I have created three different programs for the different parameter update, some of the "best" performing CG on the CUTEr test set uses a hybrid implementation. The top performing one, with respect to CPU time, is based on the Hager and Zhang update. Note that if the function is strongly convex, the choices for the update parameter with an exact line search should be the same [14].

VII. Accuracy

While there are many ways to evaluate performance like using MAE or RMSE, here I evaluate performance based on the MSE, namely the loss function. The goal is to minimize the loss function without over-fitting. In addition, for each algorithm I shuffled the data set three times, and averaged the MSE. The optimal weight was:

$$\boldsymbol{w} = \begin{bmatrix} 454.228 \\ -14.63 \\ -2.9884 \\ 0.3774 \\ -2.993 \end{bmatrix}$$

VIII. Experiments

I optimize the loss function by running multiple optimization methods. For each method, I determine α differently as mentioned above. In addition, I set the stopping tolerance to be 10^{-4}, a smaller tolerance will not work for gradient descent as it will take too long to converge. The max iterations is set at 4000 and the starting point is set as w = (0,0,0,0,0) for every algorithm.

IX. Result for Gradient Descent

I've exhaustively tested different $\alpha \in [0.0001, 1]$. Any α below 0.0001 will take over 4000 iterations to converge, and any α over 0.5 will diverge, $|g|$ and value of loss function goes up. As can be seen, for gradient descent, choosing a "bad" α can be very costly. The optimal α from manual tuning, which occurs at 0.1, converges at iteration 110. Even at the optimal α from manual tuning, in-exact line search outperforms manual tuning by a considerable margin.

When comparing two line-search methods on GD, Algorithm 2 and Algorithm 3, one can notice that both does a very decent job initially, see figure 3. However, algorithm 2 actually converges in 88 iterations, while algorithm 3 converges in in 89. Note that I have excluded the first iteration, since at iteration one, the loss is over 10^4, see figure 1. Although there is a big decrease from iterations 1-5, it takes around 80 more iterations for the $|g|$ of both algorithm to be below 10^{-4}. After the 5th iteration, GD using line-search makes minimal progress. Regardless, line search is a useful tool that performed better that manual searching on GD in this experiment.

Even by exhaustively searching for the optimal α and employing line-search, from figure 3, gradient descent seems to converge sub-linearly.

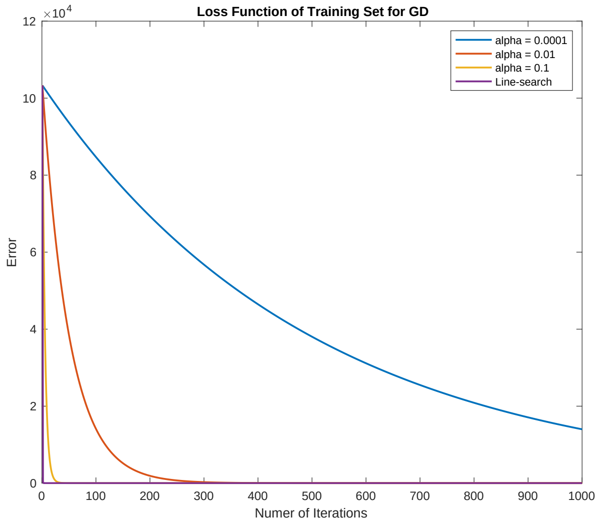


Figure 1. Comparison of Gradient Descent With multiple α

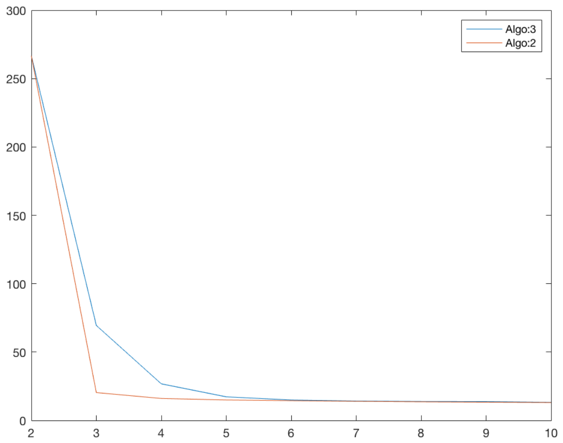
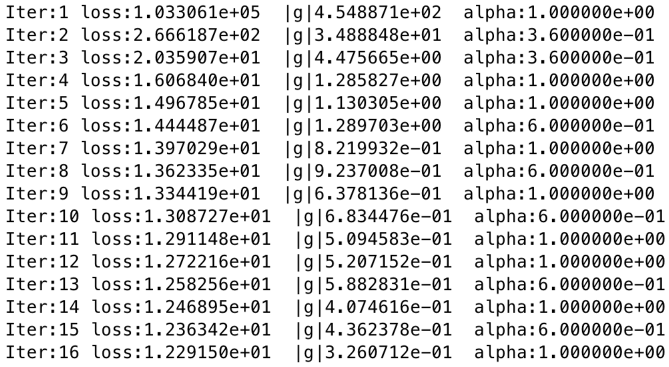


Figure 2. Different Line-Searches on GD

In many online tutorials and undergraduate machine learning courses, like Professor Andrew Ng's Machine Learning course [9], there is more emphasis on manual tuning than on line-search. While manual tuning may work well for many other linear regression data set, one should also implement a backtracking line-search technique to determine α as it can



```
Iter:1 loss:1.033061e+05  |g|4.548871e+02  alpha:1.000000e+00
Iter:2 loss:2.666187e+02  |g|3.488848e+01  alpha:3.600000e-01
Iter:3 loss:2.035907e+01  |g|4.475665e+00  alpha:3.600000e-01
Iter:4 loss:1.606840e+01  |g|1.285827e+00  alpha:1.000000e+00
Iter:5 loss:1.496785e+01  |g|1.130305e+00  alpha:1.000000e+00
Iter:6 loss:1.444487e+01  |g|1.289703e+00  alpha:6.000000e-01
Iter:7 loss:1.397029e+01  |g|8.219932e-01  alpha:1.000000e+00
Iter:8 loss:1.362335e+01  |g|9.237008e-01  alpha:6.000000e-01
Iter:9 loss:1.334419e+01  |g|6.378136e-01  alpha:1.000000e+00
Iter:10 loss:1.308727e+01  |g|6.834476e-01  alpha:6.000000e-01
Iter:11 loss:1.291148e+01  |g|5.094583e-01  alpha:1.000000e+00
Iter:12 loss:1.272216e+01  |g|5.207152e-01  alpha:1.000000e+00
Iter:13 loss:1.258256e+01  |g|5.882831e-01  alpha:6.000000e-01
Iter:14 loss:1.246895e+01  |g|4.074616e-01  alpha:1.000000e+00
Iter:15 loss:1.236342e+01  |g|4.362378e-01  alpha:6.000000e-01
Iter:16 loss:1.229150e+01  |g|3.260712e-01  alpha:1.000000e+00
```

Figure 3. Output of Gradient Descent with linesearch

result in faster convergence, see Table 1.

Table I
GRADIENT DESCENT CONCLUSION

Methods:	Iterations till converged
$\alpha = 0.0001$	Doesn't converge in 4000 iterations
$\alpha = 0.001$	660
$\alpha = 0.01$	110
GD using Algo 3	89
GD using Algo 2	88

X. RESULT FOR NEWTON'S METHOD

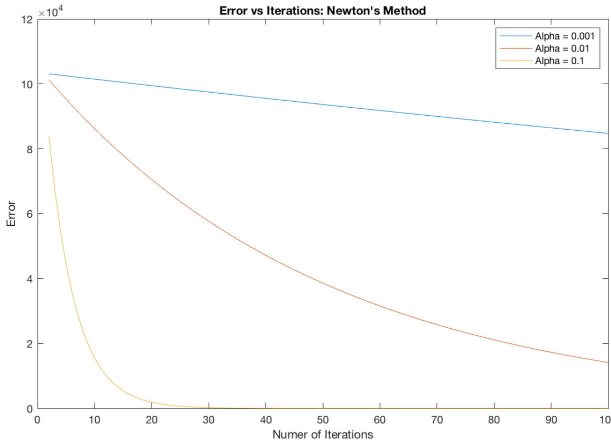


Figure 4. Newton with manual tuning

In Newton's algorithm, I manually tuned for $\alpha \in (0.001, 1)$. The difference between each α from 0.001 to 0.01 to 1 is substantial, which is hard to graph, so I've included the plot of $\alpha = 1$ along with Newton using line-search in the figure 5. Initially, when looking at $\alpha \in \{0.001, 0.01, 0.1\}$ one might be tempted to think that manually tuning is not good. However, even at a non-optimal α, like $\alpha = 0.1$, it still converges within 50 iterations. Figure 4 shows that manual tuning can perform better when the you pick a better α.

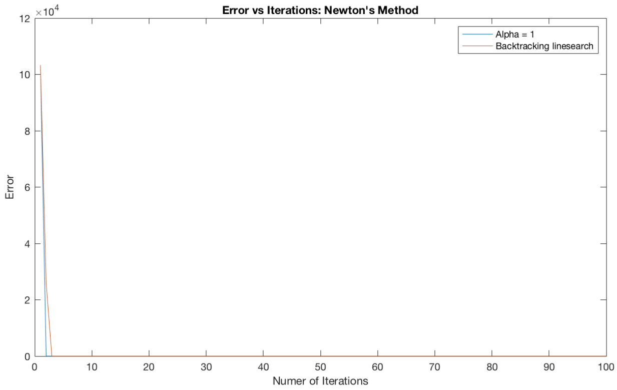


Figure 5. Newton (In-Exact) line-search vs manual tuning

The optimal α from manual tuning occurs at α = 1, which converges at iteration 2. Note that $\alpha = 1$, is considered as a pure Newton method. On the other hand, with in-exact backtracking line search (Algorithm 2), Newton's algorithm converges at iteration three. This is sometimes known as a damped Newton's method. In Newton's case, a pure Newton method beats Newton with backtracking line-search. Newton's method, pure and damped, performs as expected, since our hessian is positive definite and easy to compute.

XI. BFGS RESULT

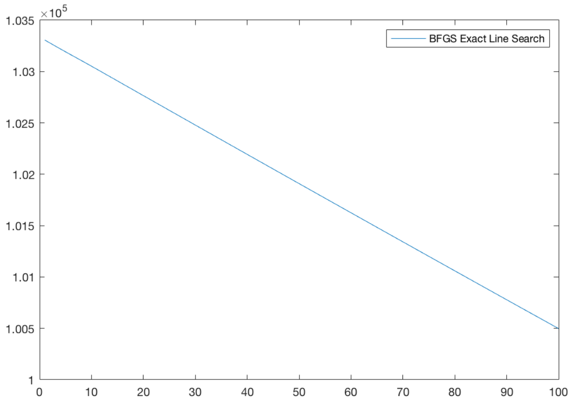


Figure 6. BFGS using Exact Line Search

While BFGS is a common technique that can converge super-linearly, using exact line search on BFGS is not an ideal choice. As can be seen, by setting α as:

$$a = \frac{-g_k^T p_k}{p_k^T H p_k}$$

BFGS looks like it will converge linearly, which fails the expectation of how BFGS should perform. Figure 6 shows that it is not advisable to use exact line search on BFGS.

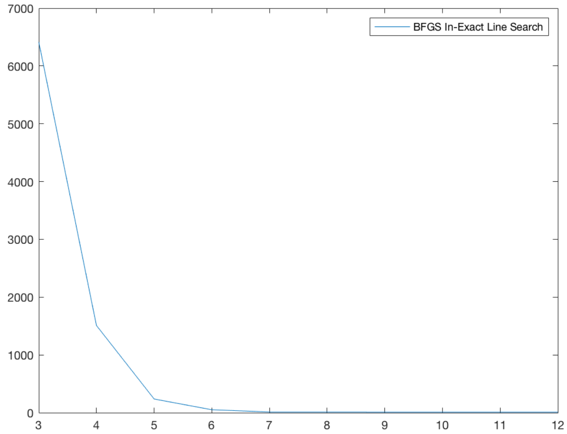
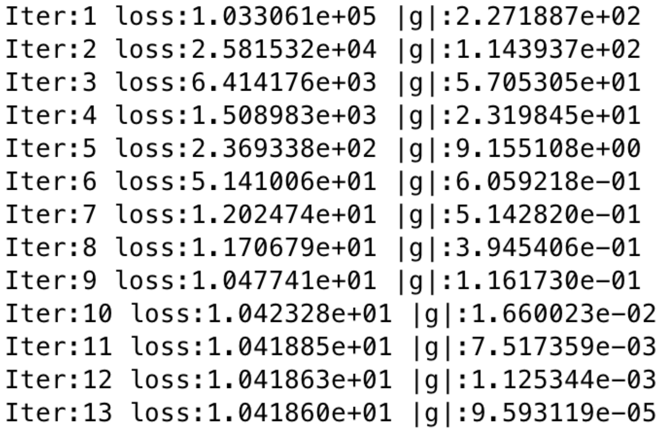


Figure 7. BFGS using (In-exact) Line Search

When using (In-exact) line search, BFGS performs significantly better than using exact line search, as it converges in 13 iterations. Although, figure 7 looks similar to figure 2, notice that at iteration 4, the value of the loss function for gradient descent is down below 50; however, the value of loss function for BFGS is over 1508. In addition, gradient descent makes minimal progress after the first few iterations while BFGS continues to effectively reduce the value of the loss function, see figure 8. $|g|$ from iteration 12 to 13 suggests super-linear convergence.



```
Iter:1 loss:1.033061e+05 |g|:2.271887e+02
Iter:2 loss:2.581532e+04 |g|:1.143937e+02
Iter:3 loss:6.414176e+03 |g|:5.705305e+01
Iter:4 loss:1.508983e+03 |g|:2.319845e+01
Iter:5 loss:2.369338e+02 |g|:9.155108e+00
Iter:6 loss:5.141006e+01 |g|:6.059218e-01
Iter:7 loss:1.202474e+01 |g|:5.142820e-01
Iter:8 loss:1.170679e+01 |g|:3.945406e-01
Iter:9 loss:1.047741e+01 |g|:1.161730e-01
Iter:10 loss:1.042328e+01 |g|:1.660023e-02
Iter:11 loss:1.041885e+01 |g|:7.517359e-03
Iter:12 loss:1.041863e+01 |g|:1.125344e-03
Iter:13 loss:1.041860e+01 |g|:9.593119e-05
```

Figure 8. BFGS using (In-exact) Line Search

Notice from iteration 10 to 13 that while the norm of the gradient decreases by a considerable amount, the value of the loss function does not change that much. Depending on your goal, if you do not want the optimal value of the weight, but just an appropriate weight that predicts y sufficiently, you may add a stopping criteria when the loss function is “small” enough. For example, stop when $loss < 11$. This can save a few iteration for bfgs, and many for gradient descent.

XII. Nonlinear CG result

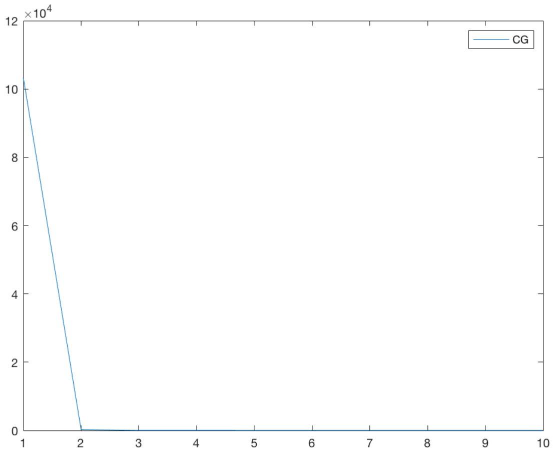
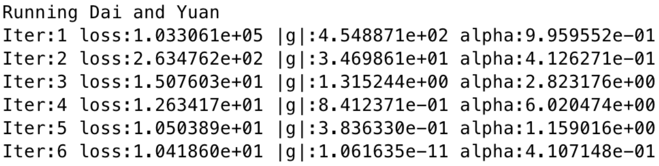


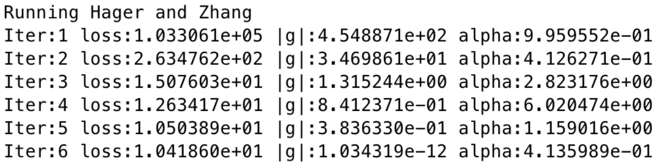
Figure 9.

This plot shows the CG plot of update using PRP, DY, and HZ. They converge exactly the same way(see figure 10 and 11), implying the function given is convex and the hessian is positive definite. When a quadratic function is strongly convex and the hessian is positive definite, PRP DY and HZ are theoretically the same [14]. This is expected given the way the loss function is defined. The same cannot be said if the loss function were to be defined as the MAE, since it is not a quadratic function.



```
Running Dai and Yuan
Iter:1 loss:1.033061e+05 |g|:4.548871e+02 alpha:9.959552e-01
Iter:2 loss:2.634762e+02 |g|:3.469861e+01 alpha:4.126271e-01
Iter:3 loss:1.507603e+01 |g|:1.315244e+00 alpha:2.823176e+00
Iter:4 loss:1.263417e+01 |g|:8.412371e-01 alpha:6.020474e+00
Iter:5 loss:1.050389e+01 |g|:3.836330e-01 alpha:1.159016e+00
Iter:6 loss:1.041860e+01 |g|:1.061635e-11 alpha:4.107148e-01
```

Figure 10.



```
Running Hager and Zhang
Iter:1 loss:1.033061e+05 |g|:4.548871e+02 alpha:9.959552e-01
Iter:2 loss:2.634762e+02 |g|:3.469861e+01 alpha:4.126271e-01
Iter:3 loss:1.507603e+01 |g|:1.315244e+00 alpha:2.823176e+00
Iter:4 loss:1.263417e+01 |g|:8.412371e-01 alpha:6.020474e+00
Iter:5 loss:1.050389e+01 |g|:3.836330e-01 alpha:1.159016e+00
Iter:6 loss:1.041860e+01 |g|:1.034319e-12 alpha:4.135989e-01
```

Figure 11.

One can see that in the first two iterations, CG doesn’t look better than GD. However CG picks up the pace rapidly, as it converges in just 6 iterations, which is faster than BFGS but uses less memory.

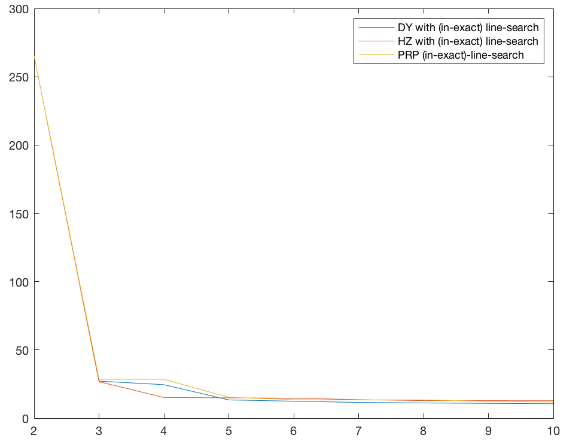


Figure 12.

In general one has to be careful when applying inexact linesearch (algorithm 2) on CG using a PRP update since PRP may find a p_k that is uphill. Figure 12 shows the plot of the loss function using CG with backtracking line-search. While it does make progress, DY takes 23 iterations before $|g|$ is below ftol. In addition, HZ takes 36 iterations, and PRP takes 50 iterations. Nonetheless, with (in-exact) line search, CG still outperforms gradient descent by a considerable margin.

XIII. Discussion

Table II

Methods:	Iterations till converged
exact line-search CG	6
(in-exact) linesearch BFGS	14
Pure Newton	2
GD using Algo 2	88

Table 2 summarizes the best setting of each method. I have chosen gradient descent to be the lower bound, in which I expected all the other methods to be faster. For this data set, since the Hessian was not hard to compute, relatively small, and positive definite, one can expect Newton's method to converge quadratically, in which it did. Newton outperformed gradient descent and BFGS by a considerable margin; however one can not rely just on one method. Amazingly, even when the data was conducive to using Newton, CG also proved to be effective, in which it uses far less memory than Newton. If there were more data and variables, CG would probably be the better one to use.

The loss function at the optimal weight is approximately 10.418. Fitting a straight line, clearly, cannot pass through all the points meaning the data points are not exactly linear. One can use other methods like polynomial regression, meta-learning algorithms, tree-based learning algorithms, and the like. However, by using those, one has to pay more attention to the regularization parameter, as polynomial regression can easily over fit without the regularization term.

The goal of the paper was to:
1) Explore different optimization methods
2) Solve a real world problem

While I can see some trade offs between the parameter choice and optimization method choice as well as as see numerical optimization techniques solving a real world data set, there are extensions that could been made. A possible extension to this paper would be to use the same methods on a bigger data set. Ideally a data set where one would see the usefulness of CG. It is difficult to say precisely how big a data set should be when one would use CG over Newton's method. In addition, I was fortunate that the hessian is positive definite, which does not hold true for many other dataset. Furthermore, the update choice for β using exact line-search in general, will not always be the same.

XIV. Conclusion

A typical machine learning algorithm comprises a model, a loss function, and an optimization method. Many machine learning papers put emphasis on choosing a good model and/or loss function. As an example, in Professor Tufekci's paper, she wrote about different machine learning models to reduce the minimize the RMSE and the MAE. While she also used linear regression, she didn't mention which optimization methods she used to optimize the weights. I would guess she used gradient descent since she used gradient descent to update the network parameters(the weight) when using a feed-forward neural network method. From this experiment it is not clear which is the optimal optimization technique to use as it differs from each data set and function type. Since many machine learning algorithms require optimizing the objective function, one should pay attention to optimization and not rely on one optimization method.

Hopefully, my result shows that choosing a good parameter, whether it is the α or β, can affect performance. While hyper-parameter tuning is a common method in machine learning, manually choosing the optimal α can be time consuming. Moreover, it may not yield the best result. One draw back of grid-search is while it does find the optimal parameter from a specified subset, the parameter is fixed throughout the iterations. Unlike grid-search, backtracking does adjust α. In conclusion, even in a relatively easy simple data set, one needs to make choices on which optimization method to use and how to best select the parameters regarding each method. In a more complicated data set, it is possible that you will need to employ hybrid methods.

References

[1] Pınar Tüfekci, Prediction of full load electrical power output of a base load operated combined cycle power plant using machine learning methods. *International Journal of Electrical Power & Energy Systems,* , Volume 60, September 2014, Pages 126-140,

[2] Chris Bishop, *Pattern Recognition and Machine Learning*. Springer ,2006.

[3] Andrew Ng. CS229 Lecture Notes: Supervised Learning. http://cs229.stanford.edu/notes/cs229-notes1.pdf.

[4] David. Rosenberg. Gradient and Stochastic Gradient Descent https://davidrosenberg.github.io/mlcourse/Archive/2017Fall/Lectures/02b.SGD.pdf.

[5] Geoff Gordon. Gradient Descent Revisted https://www.cs.cmu.edu/~ggordon/10725-F12/slides/05-gd-revisited.pdf.

[6] Zico Kolter. Quasi-newton methods http://www.stat.cmu.edu/~ryantibs/convexopt/lectures/quasi-newton.pdf.

[7] Aarti Singh. Newton Method http://www.cs.cmu.edu/~pradeepr/convexopt/Lecture_Slides/Newton_methods.pdf.

[8] M.H Wright and P.E Gill, *Introduction to Numerical Optimization*. ,November 26, 2018.

[9] Andrew NG. Machine Learning https://www.coursera.org/learn/machine-learning.

[10] Yu-Hong Dai. Nonlinear Conjugate Gradient Methods ftp://lsec.cc.ac.cn/pub/dyh/papers/CGoverview.pdf

[11] https://scikit-learn.org/stable/modules/grid_search.html

[12] Raphael Hauser. Line Search Methods for Unconstrained Minimization https://people.maths.ox.ac.uk/hauser/hauser_lecture2.pdf

[13] Neculai Andrei. A modified Polak-Ribière-Polyak conjugate gradient algorithm for unconstrained optimization https://www.tandfonline.com/doi/abs/10.1080/02331931003653187

[14] William W. Hager and Hongchao Zhang. http://people.cs.vt.edu/~asandu/Public/Qual2011/Optim/Hager_2006_CG-survey.pdf

XV. Relevant Code

All the relevant code can be found at: https://github.com/jeffcipher/nonlinear_op