CS 594: Advanced Topics in Machine Learning **Due: 17:00, Nov 4, 2016**

Project 1: Convex Optimization

Student: Yanzi Jin, Yaru Shi *Email: yjin25, yshi31@uic.edu*

1 Parallel Optimization: Tasks for Experiment

Question P1. Run your CRF implementation using the following command

```
mpirun -n 3 ./seq_train -data ./train.txt -tdata ./test.txt -lambda 1e-3 -loss CRF -tol 1e-3
```

Copy and paste to your report i) the first 11 lines (*i.e.* 10 iterations) of the output, and ii) the last line when it converges. For example (extracted from `sample_out_IID.txt`)

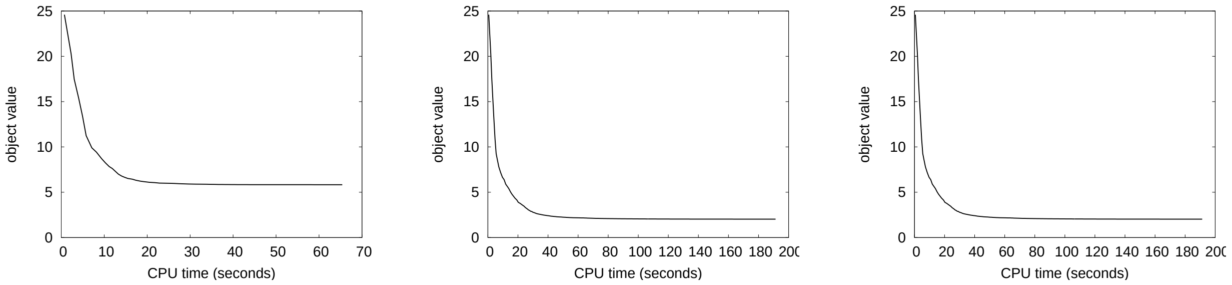
Solution P1. The output of the first 10 iterations and the last line when it converges is as follows:

```
iter fn.val gap time feval.num train_lett_err train_word_err test_lett_err test_word_err
0 24.5949 4.55704 0.487572 1 92.309174 100.000000 92.220780 100.000000
1 20.2619 5.09505 1.64694 3 71.379031 100.000000 71.421483 100.000000
2 17.428 3.65693 2.39035 4 66.666667 99.825480 66.382930 99.883687
3 15.3061 2.66886 2.9951 5 53.446615 97.643979 53.500267 97.702821
4 13.2133 2.13058 3.6022 6 45.382037 96.160558 45.644706 95.347485
5 10.8931 2.51846 4.19833 7 39.671714 91.826643 40.426750 92.294272
6 9.33426 1.1719 4.79845 8 33.803414 87.085515 34.647683 87.699913
7 8.75664 0.875296 5.41022 9 31.553192 84.293194 32.326895 85.199186
8 7.89758 0.851361 6.02768 10 28.551613 80.337405 29.361020 81.913347
9 7.2986 1.54779 6.65938 11 27.129812 77.370564 28.128101 78.045944
10 6.79134 0.731165 7.27257 12 24.552075 72.949389 25.608825 74.469322
:
:
114 3.07372 0.00436881 74.7762 120 10.545987 39.383362 14.268265 48.706019
Optimization converged with status CONVERGED_GTTOL.
```

Question P2. The test error obviously depends on the regularization parameter λ. For each λ value in `{1e-2, 1e-4, 1e-6}`, plot a figure with a single curve where the x-axis is the CPU time elapsed (in seconds), and the y-axis is the objective value of the current solution. One dot for each iteration's output. Again use three cores, and CRF loss. If it takes too long to converge, you may relax the tolerance to *e.g.* `-tol 1e-2`, or set the max number of iteration or function evaluation to some smaller value (line 77 and 79 of `seq_train.cpp`), provided that the trend has leveled off.

Does a larger value of λ allow the optimization to converge faster in CPU time, or a smaller value of λ? Why? Hint: the solver is quasi-Newton (like LBFGS).

Solution P2.



(a) $\lambda =$ 1e-2 (b) $\lambda =$ 1e-4 (c) $\lambda =$ 1e-6

Figure 1. Comparison of object value v.s. CPU time over $\lambda \in \{$1e-2, 1e-4, 1e-6$\}$.

Figure 1 gives the test error curve with different λ. From the plots we can tell a larger λ makes the optimization converge faster.

Consider that the objective function is

$$-\frac{1}{n}\sum_{t=1}^{n}\log p(\mathbf{y}^t|X^t) + \frac{\lambda}{2}\left(\sum_{k\in\mathcal{Y}}\|\mathbf{w}_k\|^2 + \frac{1}{2}\sum_{ij}T_{ij}^2\right). \tag{1}$$

and the solver is quasi-Newton, which means the λ is kept in the second-order gradient. As the λ increases, more penalization is put on the l_2 norm of $\boldsymbol{w}$ and $\boldsymbol{T}$, and more restrictions regarding on the summation on the parameters. As the domain area for potential parameter estimation will be smaller, and the converging time will be shorter compared with the larger λ.

Question P3. Pick the λ from the last question that gives the lowest test word-wise error. Plot a figure with two curves: one for the test letter-wise error as time goes by (x-axis is CPU time in seconds), and one for the training letter-wise error. Then plot another figure for word-wise error.

What can be observed from these plots? In addition, compare these plots with those in Question P2. Does training/test error keep decreasing as the objective function is being reduced?

Solution P3.

Figure 2 and 3 gives how the training/testing error goes with time with $\lambda =$ 1e-4. We can see that training and testing errors drop with time, and testing error is always a little bit higher than training error, both letter-wise and word-wise. Compared with Figure 1, the optimization process converges around 60 seconds, and the letter-wise and word-wise error for both training and testing errors doesn't change too much after 60 seconds.

Question P4. Scalability test. In parallel computing, it is standard to study how the speedup depends on the number of cores. Suppose some job costs T_r time if run on r cores. Then the speedup of r cores is defined as T_1/T_r. To rigorously define the "time to converge", one approach is to first set the tolerance to very tight (*e.g.* `-tol 1e-5`) and get a highly accurate estimate of the true minimum objective value. Denote it as $f^* \geq 0$. Then for any given new parameter setting of the solver (the optimization problem itself is kept intact), define the "time to converge" as the time required to find an solution whose objective value falls below $1.01 \cdot f^*$ (or $1.001 \cdot f^*$, etc).

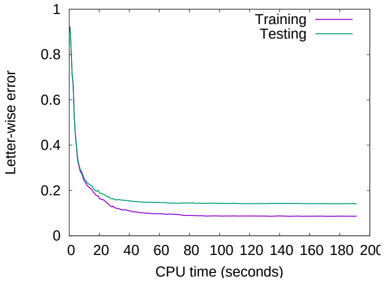


Figure 2. Letter-wise error v.s. CPU time (λ=1e-4)

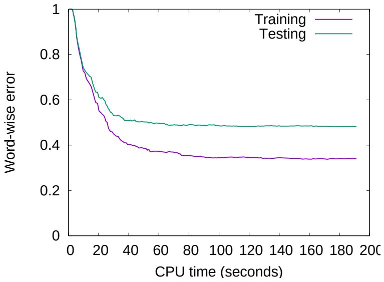


Figure 3. Word-wise error v.s. CPU time (λ=1e-4)

Fix `-lambda 1e-4 -loss CRF`. Vary the number of core $r \in \{1, 2, 4, 6, 8\}$ and plot a curve of T_1/T_r as a function of r. You may use 8 cores to first get a highly accurate estimate f^* with `-tol 1e-5`. When r is small, it will take longer, and you may relax the tolerance to "`-tol 1e-3`" provided that the solver does not terminate before finding a solution whose objective value is less than $1.01 \cdot f^*$ (or $1.001 \cdot f^*$ if you feel proper).

Based on the plot, what conclusion can be drawn? If the curve is not linear, explain why.

Solution P4. The solution is found when objective value is less than $1.01 \cdot f^*$. Figure 4 shows the relationship between the number of cores and speedup, with λ = 1e-4. We see that when the number of cores is smaller than 8, the speed increases with the number of cores, showing parallel computing helps with speedup. However, the speedup drops when the number of cores reaches 8. This indicates the communication cost is more than the computation cost, the speedup will keep dropping when the number of cores keeps growing.

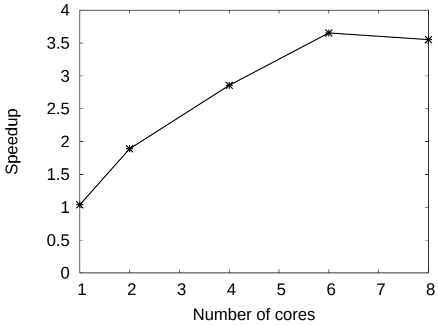


Figure 4. Speedup as a function of #core

Question P5. LBFGS (to be exact, the LMVM algorithm in TAO) keeps a number of previous gradients to approximate the inverse of the Hessian. Call this number p (default is 5 in TAO). Although a larger value of p leads to more accurate approximation, it also incurs higher computational cost. So there seems to be a delicate tradeoff. Set #core to 4, and fix λ to the optimal value in Question P2. Vary the value of p in $\{1, 5, 20\}$ ($p = 5$ has already been done in previous questions). To set p to 1, just append the following to the command line in Question P1:

```
-tao_lmm_vectors 1
```

Reproduce the plot in Question P2 for this optimal λ. Still use "`-n 3 -loss CRF -tol 1e-3`". But instead of plotting 3 figures (for the three values of p), plot 3 curves in a single figure. What conclusion can be drawn?

Solution P5.

The results is shown in 5, still with $\lambda =$ 1e-4. From the plot we can see that when $p = 1$, the object value drops slower than the other two curves, and it also takes longest time to converge. However, $p = 5$ and $p = 20$ does not have a big difference. Since larger p may cause heavier computation cost, the default $p = 5$ is a good balance between approximation accuracy and computation cost.

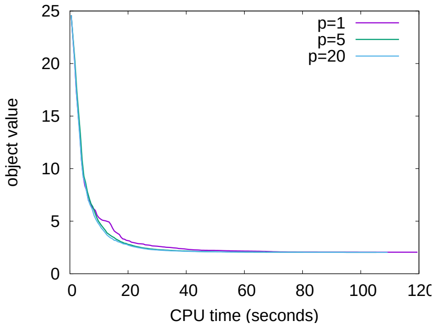


Figure 5. Comparison of object value v.s. CPU time over $p \in \{1, 5, 20\}$.

Question P6. Do we really need to store the whole C_{train} matrix in memory via storing $\mathbf{c}_{\text{local}}$ over all processes? If yes, explain why. If no, explain how this can be saved and how the computational performance will be impacted.

Solution P6.

We do not have to store $\mathbf{c}_{\text{local}}$ over all processes. Since the result of C_{train} computed by each process is not shared with other processes, it can also stored locally. Then the gradient G can also be computed locally. Finally we can just collect the gradient of all processes by scatter. In such case, we only need to copy the gradient once, which saves the communication cost. It will be affected more if the number of processes is large.

2 Torch: Tasks for Experiment

Question T1. Run your implementation with the setting in the first 19 lines of `train_CRF.lua`. That is, use the first 100 words for training. Copy and paste to your report i) the first 11 lines (*i.e.* 10 iterations) of the printout, and ii) the last line when it converges. See `sample_out_Torch.txt` for an example (only 4 lines are shown there, and the last line 49 is dummy).

For the questions below, you may create new files, *e.g.* if you feel it more convenient to collect SGD related code in a separate file.

Solution T1. The output of the first 10 iterations and the last line is:

```
iter fn.val gap time feval.num train_lett_err train_word_err test_lett_err test_word_err
0 24.0122 192.5973 3.234 1 92.40 100.00 92.22 100.00
1 24.0015 192.4616 30.401 2 66.62 100.00 69.44 100.00
2 22.1144 173.3414 57.821 3 58.62 100.00 63.11 100.00
3 19.8238 151.3710 88.527 4 56.85 99.00 61.57 99.85
4 17.7605 132.3901 121.813 5 53.05 99.00 59.30 99.39
5 16.0028 116.6411 151.097 6 46.40 96.00 56.25 98.95
6 14.4284 103.1173 178.721 7 38.94 92.00 52.21 98.05
7 12.9357 91.2750 206.400 8 34.60 90.00 49.59 97.06
8 11.2892 76.9717 234.086 9 29.04 83.00 45.07 95.64
9 10.1704 67.0261 262.332 10 26.19 79.00 43.09 94.82
10 9.2492 59.7605 289.435 11 24.02 75.00 41.95 94.13
:
:
49 0.7562 1.6087 1413.176 50 0.00 0.00 36.74 87.15
<optim.lbfgs>  reached max number of iterations
```

Question T2. Implement the vanilla SGD algorithm and do not use `optim.sgd`. Also implement the Stochastic Average Gradient method with Non-Uniform Sampling (SAG-NUS) introduced by

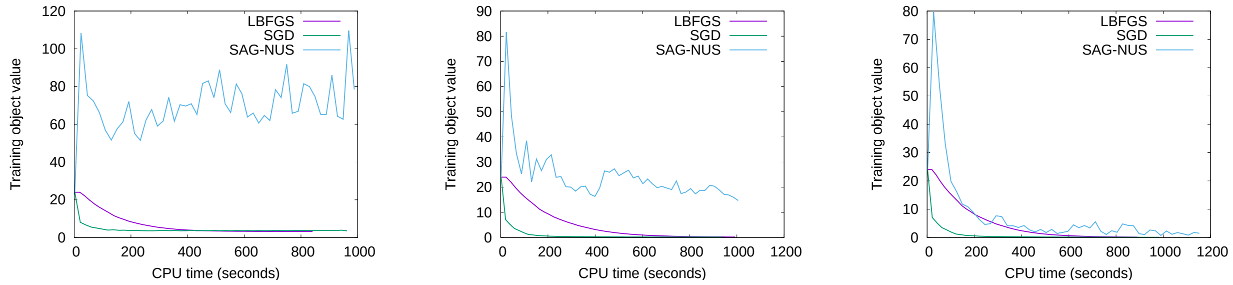
Non-Uniform Stochastic Average Gradient Method for Training Conditional Random Fields. M. Schmidt, R. Babanezhad, M.O. Ahmed, A. Defazio, A. Clifton, A. Sarkar. AISTATS, 2015. Their talk and Matlab code (for reference) can be found at `http://www.cs.ubc.ca/~schmidtm/Documents`.

Now for each $\lambda \in$ `{1e-2, 1e-4, 1e-6}`, plot a figure that that compares the three methods (LBFGS, SGD, SAG-NUS) over the training objective value. See Figure 6 for made-up plots. Still use the first 19 lines of `train_CRF.lua` for the problem setting. However, **you may now tune the parameters of the solvers.** This includes, but not limited to, the `optimState` structure in `train_CRF.lua` for LBFGS, as well as the step size rules for SGD and SAG-NUS. Use whatever parameter that you feel is good for this problem, and mention them in the report. What conclusion can be drawn from these figures?

Note it can be expensive if we compute the objective value after *each single* update of SGD/SAG-NUS. So you may compute the value every 1000 updates (or pick a frequency that you feel reasonable). The training set is again the first 100 words, and SGD/SAG-NUS randomly picks one out of them at each step.

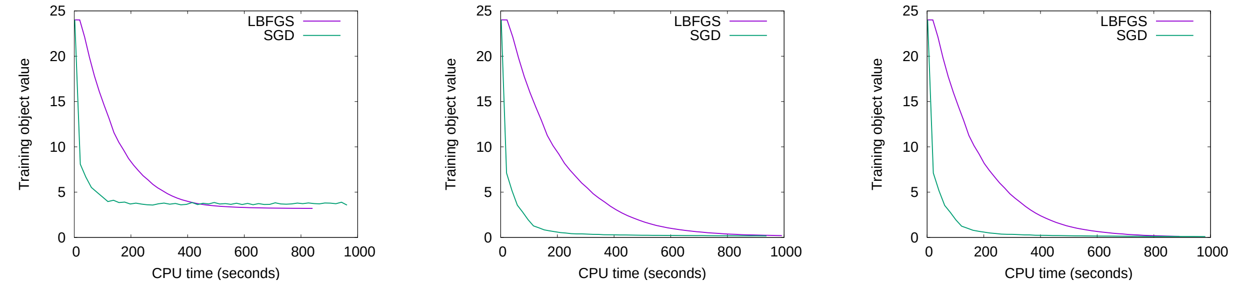
Solution T2.

For small training set (100 words), we compute the objective value after every 100 updates. Figure 6 is the overall performance of three methods and Figure 7 gives a closer look of LBFGS and SGD since the scale in Figure 6 is not able to show the difference of SGD and LBFGS. From the plots we can see that generally SGD converges faster than LBFGS and a lager λ results in faster convergence. When the $\lambda = 1\mathtt{e} - 2$, the solution still has some fluctuation after convergence. For the SAG-NUS method, it can been seen that it performs the worst in general among these three methods. And there is always a spike after the first 100 updates. Furthermore, the convergence



(a) $\lambda =$ 1e-2 (b) $\lambda =$ 1e-4 (c) $\lambda =$ 1e-6

Figure 6. Comparison of LBFGS, SGD, and SAG-NUS over $\lambda \in \{$1e-2, 1e-4, 1e-6$\}$.



(a) $\lambda =$ 1e-2 (b) $\lambda =$ 1e-4 (c) $\lambda =$ 1e-6

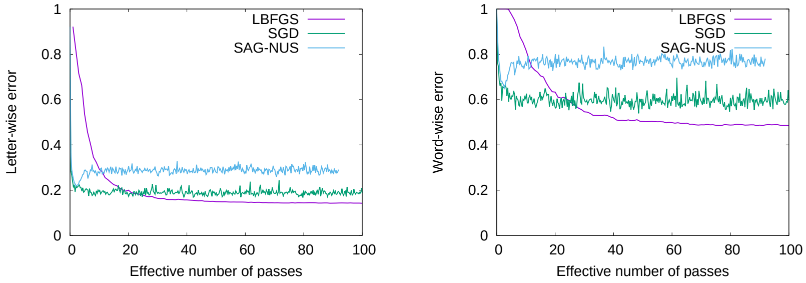
Figure 7. Closer look of comparison of LBFGS, SGD, and SAG-NUS over $\lambda \in \{$1e-2, 1e-4, 1e-6$\}$.

status of it is greatly associated with the value of λ. More specifically, as λ goes larger, it becomes harder for SAG-NUS to reach convergence. After checking the step size calculated based on the L_g in SAG-NUS, we noticed that the step size is first around 0.9 then drop to 0.5 during the updates. This step size is much larger compared with the 0.1 we choose for the SGD method. Therefore, the first several iteration might drive the w to far away from the truth due to the large step size, and the large λ can also make the things get worse according to the update function.

Question T3. What is your general experience of tuning the parameters of the solvers in **Question T2**? You do not need to be exhaustive, and just briefly describe what you tried and what you found.

Solution T3. We have tried different step size, and as we expected, the solution converges slower with a smaller step size. We also tried to evaluate the objective function and error every 5, 30 and 100 updates of SGD/SAG-NUS, more frequent computation shows more fluctuation of object value, this also makes sense because of the randomness by sampling. We also see that different λ takes different time to converge, generally large λ should have a larger maximal iteration number.

Question T4. The advantage of stochastic optimization lies in the low cost per step. Batch solvers like LBFGS require a lot of computation to process the whole data set, which made it computationally challenging to train on the entire 3438 words. So in the previous questions, we only used the first 100 words for training. Now let us try SGD and SAG-NUS on the whole training set by randomly picking a word out of the **3438 words**.



(a) Letter-wise error

(b) Word-wise error

Figure 8. Comparison of testing error on the test data for $\lambda = 1e - 4$.

Fix the value of λ to what you used in **Question P3**, whose results were given in Figures 2 and 3. Re-produce Figure 3 (word-wise error) but do *not* include the training error any more. Instead, include the test error of LBFGS, SGD, and SAG-NUS. Now it should look like Figure 8.

The result of LBFGS is just copied from the PETSc results. However, its timing was based on running on the cluster with multiple cores, and is therefore not comparable with that of SGD/SAG-NUS which use Torch on a single core. So we changed the x-axis in Figure 8 to the *effective number of passes*, which is essentially how many passes the solvers have swept over the entire training set. For LBFGS, it is the 5-th number (integer) in the printout shown in **Question P1**. For SGD/SAG-NUS, it is the number of words sampled so far divided by 3438 (#words in the training set). Again for stochastic methods, the error can be computed every, say, 1000 steps.

What conclusions can be drawn from the plot?

Extra credit. You may also plot a similar figure for the test letter-wise error, and that will earn you extra 10 points out of 100.

Solution T4. Figure 8 gives the comparison of letter-wise and word-wise error of three methods, trained on the entire training set with 3348 words. In LBFGS, the solver has 184 effective passes, however, since the torch version is quite slow, and both SGD and SAG-NUS tend to converge after 20 effective passes, we constrain the x axis to be $0 \sim 100$. Consistent with our conclusion above, SAG-NUS performs the worse in terms of test error and LBFGS achieves the lowest letter-wise and word-wise error. As stochastic methods, SGD and SAG-NUS has very fast convergence rate at the first several iterations, and keep fluctuating after convergence, while LBFGS has a relative slow convergence rate. For the performance of SAG-NUS, we believe that its strategy is too risky and sometimes makes too big step during update and stuck in local minimal. Overall, SGD is a good balance between convergence rate and accuracy.