Recursive Least Squares for Online Learning

Sameen Islam

I. Introduction

Batch learning algorithms generate predictions by learning on the entire training dataset, whereas an online learning algorithm considers data arriving in a sequential order which is then used to update the prediction. This makes online learning adaptive to data distribution which changes with time. In this paper, we focus on comparing two online learning algorithms: stochastic gradient descent (SGD) and recursive least squares (RLS). Since RLS takes an online approach to the least squares regression problem, we build up to it by first considering the batch learning case where the function $f(x) = \mathbf{w}^T X$ is to be learned through adjusting weights $\mathbf{w} \in \mathbb{R}^D$ from input matrix $X \in \mathbb{R}^{N \times D}$ and target $\mathbf{y} \in \mathbb{R}^N$. Weights are learned by minimising error:

$$E = \sum_{i=1}^{N} (x_i^T w - y_i) = (X\mathbf{w} - \mathbf{y})^T (X\mathbf{w} - \mathbf{y}) = ||X\mathbf{w} - \mathbf{y}||_2^2 \tag{1}$$

Equation 6 in appendix shows that the derivative of this error function gives us the closed-form pseudo-inverse solution with non-singular covariance matrix $\Sigma = X^T X$:

$$\mathbf{w} = (X^T X)^{-1} X^T \mathbf{y} = \Sigma_N^{-1} \sum_{i=1}^{N} (x_i y_i) \tag{2}$$

Where the computational complexity of Σ is $O(ND^2)$ as $\Sigma_N = \sum_{i=1}^{N} (x_i x_i^T)$. In addition, inversing Σ incurrs $O(D^3)$ while the remaining matrix multiplication $X^T \mathbf{y}$ takes $O(D^2)$. Thus, the computational complexity for updating the weight is $O(ND^2 + D^2)$. If we consider performing this update on a dataset of size N, in a sequential nature after the arrival of each datapoint $n = 1, ..., N$ then the total complexity will be $O(N^2 D^2 + ND^3)$.

RLS does not incur the high computational complexity like the least squares solution as it does not compute an inverse after the arrival of a new datapoint. By initialising $w_0 = 0$ where $\mathbf{w} \in \mathbb{R}^D$ and $P_0 = \mathbf{I} \in \mathbb{R}^{D \times D}$ the weights can be learned through iteration using the matrix inversion lemma which is much more efficient as described in equation 7 in appendix:

$$\begin{aligned} P_n &= P_{n-1} - \frac{P_{n-1} \mathbf{x_n} \mathbf{x_n^T} P_{n-1}}{1 + \mathbf{x_n^T} P_{n-1} \mathbf{x_n}} \\ \mathbf{k_n} &= \frac{\lambda^{-1} \times P_n \cdot x_n}{1 + \lambda^{-1} \times x_n . T \cdot P_n \cdot x_n} \\ \mathbf{w_n} &= \mathbf{w_{n-1}} - P_n \mathbf{x_n} (\mathbf{x_n}^T \mathbf{w_{n-1}} - \mathbf{y_n}) \end{aligned} \tag{3}$$

Published on February 22, 2021. The author can be contacted via si1u19@soton.ac.uk

where P_n is the inverse of the covariance, k_n is the gain vector and $\mathbf{w_n}$ is the weight vector.

An alternative method of batch learning is gradient descent (GD) which scales well for large number of input data as it does not face the computational complexity of matrix inversion. The learning method minimises a function $\min_x f(x)$ by computing its gradient and taking a small step (defined by learning rate η) in the negative direction from a randomly initialised point:

$$\begin{aligned} \mathbf{w}^{(n+1)} &= \mathbf{w}^{(n)} - \eta \nabla_{\mathbf{w}} E \\ \nabla_{\mathbf{w}} E &= -2X^T (\mathbf{y} - X\mathbf{w}) \end{aligned} \tag{4}$$

This is really effective on smooth convex functions, however, for an input X with very large number of rows, $\nabla_{\mathbf{w}} E$ can take very long time to compute as the $X^T X$ operation essentially sums over all data points. Instead of computing the exact $\nabla_{\mathbf{w}} E$, if we instead approximated after the arrival of new datapoints, we could rapidly descend towards the minima, avoiding getting trapped in saddle points and local minima due to noise in each iterative update. With SGD, we may achieve this:

$$\mathbf{w}^{n+1} = \mathbf{w}^n - \eta (y_n - \mathbf{w}^T \mathbf{x}_n) \mathbf{x}_n \tag{5}$$

As it does not operate on the entire input X, SGD can benefit from lower computational complexity costs thereby presenting itself as an attractive learning option even for functions that tackle problems using batch learning.

II. Dataset

To compare the learning algorithms, we formulate a dataset of varying sizes and dimensions as described in table I. There are two scenarios: in the first, there are a high number of rows and a small dimensionality. In the second, there are a small number of rows and a high dimensionality. Data is sampled from a normal distribution with Gaussian noise added to targets as we want to simulate the noise that arises in real-world data. In the case of linear least squares regression and gradient descent algorithms, we consider batch learning as the model is exposed to the entire training set of input $X \in \mathbb{R}^{N \times D}$ and target $\mathbf{y} \in \mathbb{R}^D$. On the other hand, since SGD and RLS are online learning algorithms, we consider m pairs of training samples $\{\mathbf{x}_1, y_1\}, ..., \{\mathbf{x}_m, y_m\}$ which simulates the sequential arrival of data over time:

$$..., \{\mathbf{x}_{t-1}, y_{t-1}\}, \{\mathbf{x}_t, y_t\}, \{\mathbf{x}_{t+1}, y_{t+1}\}, ...$$

where inputs $\mathbf{x_n} \in \mathbb{R}^D$ and y_n is a scalar. The three cases of input matrix X as defined in table I cover cases where number of datapoints far exceed dimensions, and vice-versa, lastly it covers the edge case where the input matrix is square. We

TABLE I: Synthetic data is generated from a normal Gaussian distribution with noise of $\mu = 0$, $\sigma = 0.8$ added to target $\mathbf{y}$. The table below shows the number of datapoints (N), dimension (D), mean (μ) and standard deviation (σ) of input matrix X.

N	D	μ	σ
500	30	0	1
30	500	0	1
300	300	0	1

TABLE II: Execution time T of learning algorithms on synthetic dataset. Times shown are averaged over 20 executions to minimise the effects of caching. RLS was applied with $\lambda = 0.9$. The closed-form solution is indicated by A^+.

Learner	N	D	I_{Max}	η	T
A^+	500	30	N/A	N/A	2.67 s
GD	500	30	1000	0.0001	498 ms
SGD	500	30	1000	0.0001	333 ms
RLS	500	30	1000	N/A	893 ms
A^+	300	800	N/A	N/A	2.45 s
GD	300	800	1000	0.0001	478 ms
SGD	300	800	1000	0.0001	388 ms
RLS	500	30	1000	0.0001	803 ms
A^+	300	800	N/A	N/A	3.35 s
GD	300	800	1000	0.0001	678 ms
SGD	300	800	1000	0.0001	488 ms
RLS	500	30	1000	0.0001	903 ms

also apply these learning algorithms on the diabetes dataset (Hastie, 2004) to demonstrate sample usage of a real-world application.

III. Results

We experimented with the three remaining learners: GD, SGD & RLS on synthetic datasets of varying dimensions as described in the previous section. We define a constant learning rate $\eta = 0.0001$ and maximum iteration $I_m = 1000$ such that training progress of each learner could be measured under the same conditions. Our main finding is that GD achieves best convergence in a relative manner within fewest iterations. This is in line with expectation as it has exposure to the entire input data and at each iteration, it takes a step towards lower gradient.

As we aimed to achieve an even comparison, we kept the η of SGD extremely small and the same as GD. As expected, the small learning rate causes SGD to undershoot approaching the minima by a very large margin as demonstrated best in figure 2. More interestingly, in an online learning scenario, we note that RLS is a much more attractive candidate as in every scenario, it manages to minimise error in a manner closely resembling GD.

RLS features a λ term which is essentially a forgetting factor which places weights on the error terms stretching back to the past as a geometric series. In this way, error terms closer to present time retains its full values while error terms further back from the past is decayed by λ. In figure 1 we find that RLS reaches convergence rapidly,

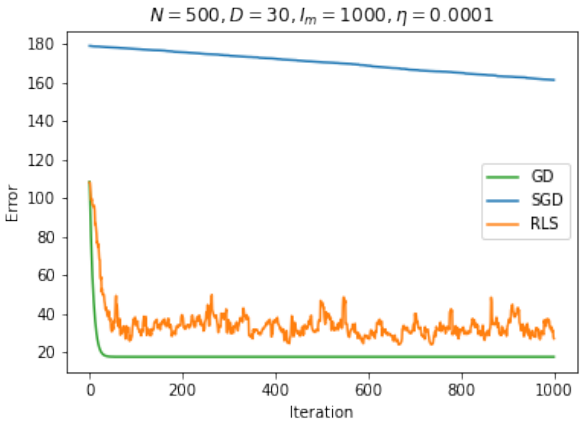

Fig. 1: RLS arrives close to convergence with $\lambda = 0.9$ but with significant variance. SGD monotonically decreases error but not quickly enough to converge into the minima.

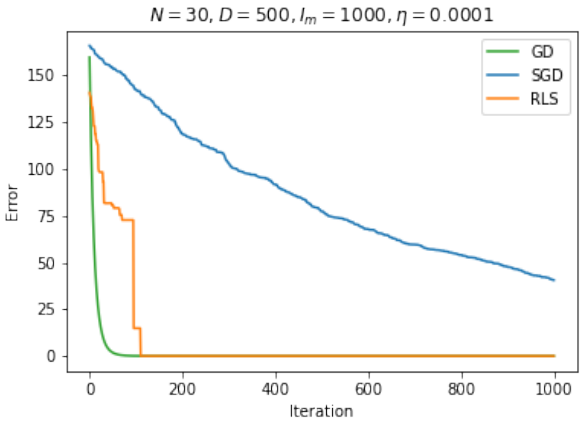

Fig. 2: SGD showing insufficient but more rapid progress. RLS with $\lambda = 0.9$ shows total convergence.

however it suffers from perturbations caused by noise when data arrives sequentially. Comparing against SGD, we find a vast difference in convergence speed with RLS as SGD monotonically decreases, albeit very slowly.

For an input matrix X that is square, we found that once again RLS demonstrated better performance compared to SGD as shown in figure 3.

When running these experiments on the diabetes data, we noticed that error could not be minimised by any of the algorithms. The reason for this could be that either the learning rate or the maximum iteration size is too small. However, increasing this did not result in error minimisation, which led us to believe that the gradient is already on a minimum. Hence, none of the algorithms exhibited convergence properties. In fact, figure 4 shows that error begins to increase as we 'jump' around the function and away from the optimum. This can be stopped with a lower η in the case of SGD or a lower λ for RLS, however, this still does not minimise the error lower than that achieved by GD. Figure 5 is interesting as we see an open source software package struggle to converge to a result when

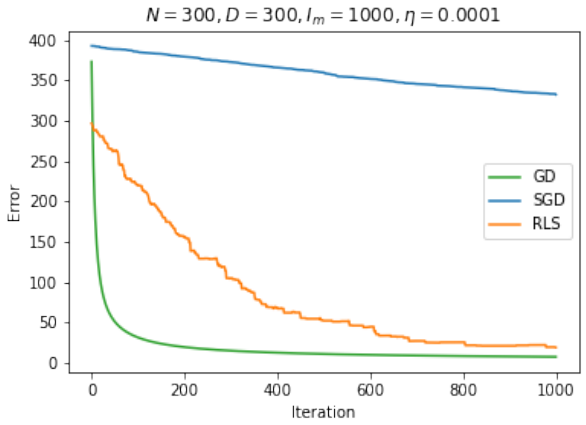

Fig. 3: RLS with $\lambda = 0.9$ monotonically decreases error however it is interesting to note that in previous cases, the learner exhibited $E \leq 50$ by 200 iterations. In this case, the learner needed to be exposed to 3 times as much data before showing the same performance.

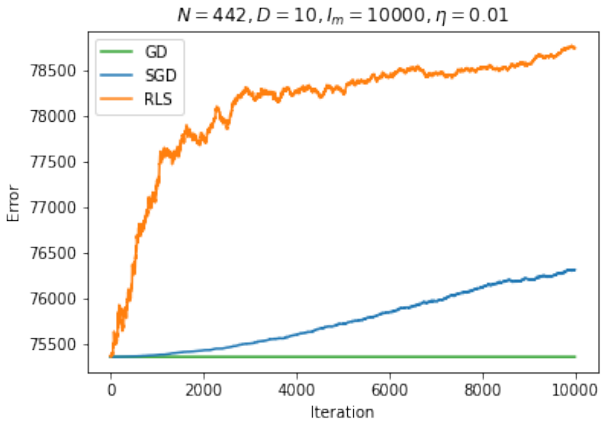

Fig. 4: Performance on the diabetes dataset. Online learning methods fail to minimise error and begin to travel in the opposite direction. The reason for non-convergence remains unknown.

Algorithm 1: RLS

Result: w_n weight at iteration i
X input matrix ;
y target vector ;
$I_m = 1000$ maximum iteration;
$\lambda = 0.98$ forgetting factor ;
$P_n = \mathbf{I}$;
for $i \leftarrow 0$ **to** I_m **do**
  $k_n = \frac{\lambda^{-1} \times P_n \cdot x_n}{1 + \lambda^{-1} \times x_n . T \cdot P_n \cdot x_n}$;
  $P_n = (\lambda^{-1} \times P_n) - (\lambda^{-1} \times k_n \cdot x_n^T \cdot P_n)$;
  $w_n = w_{n-1} - k_n \times (x_n^T @ w_{n-1} - y_n)$;
end

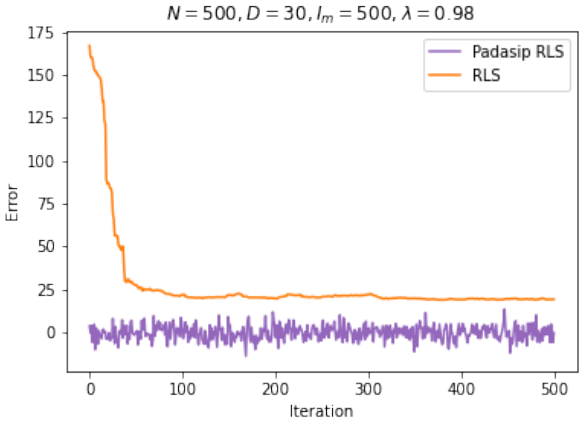

Fig. 5: Padasip RLS fails to converge whilst our implementation performs as expected. The fault in the external package may have arisen due to implementation bug.

compared against our own. We believe this may be caused by implementation error.

IV. Conclusion

In this paper we have provided the algorithm and implemented the Recursive Least Squares (RLS) filter in an online learning setting. We performed experiments to verify that RLS indeed is faster than existing gradient based methods, in addition to converging to minima faster. This claim is supported by benchmarks showing execution time of each algorithm and the relative error value after convergence. Furthermore, we have applied these algorithm to a real-world diabetes dataset to test performance on non-synthetic data. Finally, we discovered that an open source implementation of RLS did not show convergence when compared against our algorithm on synthetic dataset. We believe this may be caused by an implementation bug, but further investigation is needed to verify this.

Appendix

A. Linear Least Squares Solution

$$\begin{aligned} \nabla_{\mathbf{w}} E &= 2X^T ||X\mathbf{w} - \mathbf{y}||_2 \\ 0 &= 2X^T ||X\mathbf{w} - \mathbf{y}||_2 \\ 0 &= X^T X\mathbf{w} - X^T\mathbf{y} \\ X^T X\mathbf{w} &= X^T\mathbf{y} \\ (X^T X)^{-1}(X^T X)\mathbf{w} &= (X^T X)^{-1} X^T\mathbf{y} \\ \mathbf{w} &= (X^T X)^{-1} X^T\mathbf{y} \end{aligned} \tag{6}$$

B. Matrix Inversion Lemma

We first compute the inverse A^{-1} and repeatedly apply the following lemma for rank one update of the inverse with new datapoints in $\mathbf{x}$:

$$(A + \mathbf{x}\mathbf{x}^T)^{-1} = A^{-1} - \frac{A^{-1}\mathbf{x_n}\mathbf{x_n^T}A^{-1}}{1 + \mathbf{x_n^T}A^{-1}\mathbf{x_n}} \tag{7}$$