Invited Paper

aSNAQ: An adaptive stochastic Nesterov's accelerated quasi-Newton method for training RNNs

Indrapriyadarsini Sendilkkumaar^{1a)}, Shahrzad Mahboubi², Hiroshi Ninomiya², and Hideki Asai³

¹ Graduate School of Science and Technology, Shizuoka University
3-5-1 Johoku, Naka-ku, Hamamatsu, Shizuoka 432-8561, Japan

² Graduate School of Electrical and Information Engineering, Shonan Institute of Technology
1-1-25 Tsujido-nishikaigan, Fujisawa, Kanagawa 251-8511, Japan

³ Research Institute of Electronics, Shizuoka University
3-5-1 Johoku, Naka-ku, Hamamatsu, Shizuoka 432-8561, Japan

^{a)} *s.indrapriyadarsini.17@shizuoka.ac.jp*

Received January 31, 2020; Revised May 26, 2020; Published October 1, 2020

Abstract: Recurrent Neural Networks (RNNs) are powerful sequence models that are particularly difficult to train. This paper proposes an adaptive stochastic Nesterov's accelerated quasi-Newton (aSNAQ) method for training RNNs. Several algorithms have been proposed earlier for training RNNs. However, due to high computational complexity, very few methods use second-order curvature information despite its ability to improve convergence. The proposed method is an accelerated second-order method that attempts to incorporate curvature information while maintaining a low per iteration cost. Furthermore, direction normalization has been introduced to solve the vanishing and/or exploding gradient problem that is prominent in training RNNs. The performance of the proposed method is evaluated in Tensorflow on benchmark sequence modeling problems. The results show that the proposed aSNAQ method is effective in training RNNs with a low per-iteration cost and improved performance compared to the second-order adaQN and first-order Adagrad and Adam methods.

Key Words: Recurrent neural network, training algorithm, Nesterov's accelerated quasi-Newton, stochastic method, Tensorflow

1. Introduction

Neural networks have shown to be effective in several applications. However, neural network training poses several challenges. A common problem in training neural networks is the vanishing and/or exploding gradient problem which is more prominently seen in the training of Recurrent Neural Networks. Recurrent Neural Networks (RNNs) are powerful sequence models, popularly used in solving

pattern recognition and sequence modeling problems such as text generation [1], image captioning [2, 3], machine translation [4, 5], speech recognition [6], etc. The structure of RNNs are similar to feed-forward neural networks except that they also allow self-loops and backward connections between its nodes [7]. These connections make RNNs capable of learning, retaining and expressing long sequential relations.

Despite the capabilities of RNNs in modeling sequences, RNNs are particularly very difficult to train mainly due to the vanishing and/or exploding gradient problem [8]. Several algorithms and architectures have been proposed to address the issues involved in training RNNs [9]. Architectures such as Long Short-Term Memory (LSTM) [10] and Gated Recurrent Units (GRU) [11] have shown to be more resilient to the gradient issues compared to vanilla RNNs. Several other studies revolve around proposing algorithms that can be effectively used in training RNNs, some of which propose the use of second-order curvature information [12]. Though first-order methods are popular for their simplicity and low computational complexity, second-order methods have shown to speed up convergence despite its high computational cost. However, very few attempts have been made to train RNNs using second-order methods. In [13] the drawbacks of first-order algorithms for RNNs were discussed and second-order approaches to speed up the convergence and to tackle the issue of vanishing gradient were proposed. [14] proposed a variant of the Broyden-Fletcher-Goldfarb-Shanon (BFGS) method that incorporates adaptive mechanisms to train RNNs. However, the high computational cost incurred in second-order methods still poses a major challenge, which further adds up in very long sequence modeling problems. Recent studies [9, 12, 15, 16] propose algorithms that judiciously incorporate curvature information while taking the computation cost into consideration.

In this paper, we propose a novel adaptive stochastic Nesterov’s accelerated quasi-Newton (aSNAQ) method for training RNNs. The proposed method is an accelerated method that incorporates the momentum and Nesterov’s accelerated gradient term. Momentum based methods and its conjunction with second-order methods have shown to significantly improve the performance and convergence speed [17–19]. The proposed method is similar to the framework of SQN [20] and adaQN [16] with some changes which are described in later sections. It combines the Nesterov’s accelerated quasi-Newton (NAQ) method [19] and adaQN method [16], thus accelerating convergence and maintaining a low per-iteration cost. This paper attempts to study the performance of the proposed aSNAQ algorithm in extension to our previous work in [21] and verify its effectiveness on larger problems and networks. The results show an improved performance compared to adaQN and popular first-order algorithms such as Adam and Adagrad, with a low computational cost that is comparable with adaQN.

2. Background

Given an input sequence $\mathbf{x} = \{x_1, \dots, x_\tau\}$, RNNs use their internal states to process the sequence of inputs. The internal states $\mathbf{h}_t$ of the RNN and the output vector sequence $\hat{\mathbf{z}}_t$ can be formalized as

$$\mathbf{h}_t = \psi_h(\mathbf{x}_t \mathbf{w}_{xh} + \mathbf{h}_{t-1} \mathbf{w}_{hh} + \mathbf{b}_h), \quad (1)$$

$$\hat{\mathbf{z}}_t = \psi_o(\mathbf{h}_t \mathbf{w}_{ho} + \mathbf{b}_o), \quad (2)$$

where $\mathbf{w}_{xh}$ is the input to hidden weight matrix, $\mathbf{w}_{hh}$ is the hidden to hidden recurrent weight matrix, and $\mathbf{w}_{ho}$ is the hidden to output weight matrix. $\mathbf{b}_h$ and $\mathbf{b}_o$ are the bias vectors of the hidden and output nodes respectively. An activation function ψ such as tanh, ReLU, sigmoid or softmax is used to introduce non-linearity.

Training in neural networks is an iterative process in which the parameters (the weights and biases) are updated in order to minimize an objective function. Gradient based algorithms are popularly used for training and the gradients of the RNN are computed using backpropagation through time [22, 23]. Given a mini-batch $X \subseteq T_r$ with samples $(x_p, z_p)_{p \in X}$ drawn at random from the training set T_r and error function $E_p(\mathbf{w}; x_p, z_p)$ parameterized by a vector $\mathbf{w} = \{\mathbf{w}_{xh}, \mathbf{w}_{hh}, \mathbf{w}_{ho}, \mathbf{b}_h, \mathbf{b}_o\} \in \mathbb{R}^d$, the objective function is defined as

$$\min_{\mathbf{w} \in \mathbb{R}^d} E(\mathbf{w}) = \frac{1}{|T_r|} \sum_{p \in T_r} E_p(\mathbf{w}). \quad (3)$$

In this paper, we define the error function $E_p(\mathbf{w})$ to be the cross entropy error given by,

$$E_p(\mathbf{w}) = - \sum_t \mathbf{z}_t \log \hat{\mathbf{z}}_t, \quad (4)$$

where $\mathbf{z}_t$ is the expected output and $\hat{\mathbf{z}}_t$ is the output predicted by the RNN at timestep t . In stochastic (mini-batch) methods, the objective function is minimized using $\nabla E_b(\mathbf{w})$, the gradient of the error function calculated on a mini-batch $X \subseteq T_r$ of batch size $b = |X|$ as shown below.

$$E_b(\mathbf{w}) = \frac{1}{b} \sum_{p \in X} E_p(\mathbf{w}). \quad (5)$$

In gradient based methods, the objective function $E(\mathbf{w})$ under consideration is minimized by updating the parameters $\mathbf{w}$ using the iterative formula

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \mathbf{v}_{k+1}, \quad (6)$$

where k is the iteration count and $\mathbf{v}_{k+1}$ is the update vector, which is defined for each algorithm. This paper proposes an accelerated second-order method for training RNNs. Hence a brief background of second-order quasi-Newton methods are described below.

2.1 BFGS quasi-Newton method

Quasi-Newton (QN) methods utilize the gradient of the objective function to achieve super-linear quadratic convergence. The Broyden-Fletcher-Goldfarb-Shanon (BFGS) algorithm is one of the most popular quasi-Newton methods for unconstrained optimization [24]. The update vector of the QN method is given as

$$\mathbf{v}_{k+1} = \alpha_k \mathbf{g}_k, \quad (7)$$

$$\mathbf{g}_k = -\mathbf{H}_k \nabla E(\mathbf{w}_k), \quad (8)$$

where $\nabla E(\mathbf{w}_k)$ denotes the gradient of the error function at $\mathbf{w}_k$. The Hessian matrix $\mathbf{H}_k$ is symmetric positive definite and is iteratively approximated by the BFGS formula [24],

$$\mathbf{H}_{k+1} = (\mathbf{I} - \rho_k \mathbf{s}_k \mathbf{y}_k^T) \mathbf{H}_k (\mathbf{I} - \rho_k \mathbf{y}_k \mathbf{s}_k^T) + \rho_k \mathbf{s}_k \mathbf{s}_k^T, \quad (9)$$

where $\mathbf{I}$ denotes identity matrix,

$$\rho_k = 1/\mathbf{y}_k^T \mathbf{s}_k \quad (10)$$

and $(\mathbf{s}_k, \mathbf{y}_k)$ is the curvature information pair given by

$$\mathbf{s}_k = \mathbf{w}_{k+1} - \mathbf{w}_k, \quad (11)$$

$$\mathbf{y}_k = \nabla E(\mathbf{w}_{k+1}) - \nabla E(\mathbf{w}_k). \quad (12)$$

2.1.1 Limited memory BFGS method (LBFGS)

As the scale of the problem increases, the cost of computation and storage of the Hessian matrix becomes expensive. Limited memory schemes help reduce the cost considerably, especially in stochastic settings where the computations are based on small mini-batches of size b . In the limited memory BFGS (LBFGS) method, the Hessian matrix is defined by applying m_L BFGS updates using only the last m_L curvature pairs $\{\mathbf{s}_k, \mathbf{y}_k\}$, where m_L denotes the memory size. The search direction $\mathbf{g}_k$ is evaluated using the two-loop recursion [24] as shown in Algorithm 1.

Algorithm 1 Direction Update: Two-loop Recursion

Require: current gradient $\nabla E(\theta_k)$, curvature pair (S,Y) buffer

```
1:  $\tau = \text{length}(S)$ 
2:  $\eta_k = \nabla E(\theta_k)$ 
3: for  $i = \tau, \dots, 2, 1$  do
4:    $\sigma_i = (\mathbf{s}_i^T \eta_k) / (\mathbf{y}_i^T \mathbf{s}_i)$ 
5:    $\eta_k = \eta_k - \sigma_i \mathbf{y}_i$ 
6: end for
7:  $\eta_k = \mathbf{H}_k^{(0)} \eta_k$ 
8: for  $i := 1, 2, \dots, \tau$  do
9:    $\beta = (\mathbf{y}_i^T \eta_k) / (\mathbf{y}_i^T \mathbf{s}_i)$ 
10:   $\eta_k = \eta_k - (\sigma_i - \beta) \mathbf{s}_i$ 
11: end for
12:  $\mathbf{g}_k = -\eta_k$ 
13: return  $\mathbf{g}_k$ 
```

2.2 adaQN

adaQN is a recently proposed method that was shown to be suitable for training RNNs as well [16]. It builds on the algorithmic framework of SQN [20] by decoupling the iterate and update cycles. adaQN targets the vanishing/exploding gradient issue by initializing $\mathbf{H}_k^{(0)}$ in the two-loop recursion

Algorithm 2 adaQN Algorithm

Require: minibatch X_k , aFIM buffer F of size m_F and curvature pair buffer (S, Y) of size m_L

Initialize: $\mathbf{w}_o = \mathbf{w}_k \in \mathbb{R}^d$, $\mathbf{w}_s = 0$, $k = 0$ and $t = 0$

```
1: while  $k < k_{max}$  do
2:   Calculate  $\nabla E(\mathbf{w}_k)$ 
3:   Determine  $\mathbf{g}_k$  using  $\nabla E(\mathbf{w}_k)$  in two-loop recursion (Algorithm 2)
4:    $\mathbf{w}_{k+1} \leftarrow \mathbf{w}_k + \alpha_k \mathbf{g}_k$ 
5:   Store  $\nabla E(\mathbf{w}_k)$  in  $F$ 
6:    $\mathbf{w}_s = \mathbf{w}_s + \mathbf{w}_{k+1}$  ▷ Weight aggregation
7:   if  $\text{mod}(k, L) = 0$  then
8:     Compute average  $\mathbf{w}_n = \mathbf{w}_s / L$ 
9:      $\mathbf{w}_s = 0$ 
10:    if  $t > 0$  then
11:      if  $E(\mathbf{w}_n) > \gamma E(\mathbf{w}_o)$  then
12:        Clear  $(S, Y)$  and  $F$  buffers
13:        Reset  $\mathbf{w}_k = \mathbf{w}_o$ 
14:        continue
15:      end if
16:       $\mathbf{s} = \mathbf{w}_n - \mathbf{w}_o$ 
17:       $\mathbf{y} = \frac{1}{|F|} \left( \sum_{i=1}^{|F|} F_i \cdot \mathbf{s} \right)$ 
18:      if  $\mathbf{s}^T \mathbf{y} > \epsilon \cdot \mathbf{y}^T \mathbf{y}$  then
19:        Store curvature pairs  $(\mathbf{s}, \mathbf{y})$  in  $(S, Y)$ 
20:      end if
21:    end if
22:    Update  $\mathbf{w}_o = \mathbf{w}_n$ 
23:     $t \leftarrow t + 1$ 
24:  end if
25:   $k \leftarrow k + 1$ 
26: end while
```

(Algorithm 1, step 7) based on the accumulated gradient information as shown below.

$$[H_k^{(0)}]_{ii} = \frac{1}{\sqrt{\sum_{j=0}^k \nabla E(\mathbf{w}_j)_i^2 + \epsilon}}. \quad (13)$$

adaQN proposes the use of an accumulated Fisher Information matrix (aFIM) that stores at each iteration the $\nabla E(\mathbf{w}_k)\nabla E(\mathbf{w}_k)^T$ matrix in a FIFO memory buffer F of size m_F . This is used in the computation of the curvature information pair $(\mathbf{s}, \mathbf{y})$ for Hessian approximation as

$$\mathbf{s} = \mathbf{w}_t - \mathbf{w}_{t-1}, \quad (14)$$

$$\mathbf{y} = \frac{1}{|F|} \sum_{i=1}^{|F|} F_i \cdot \mathbf{s}, \quad (15)$$

where $\mathbf{w}_t$ is the average aggregated weight (Algorithm 2, steps 6–8), t is the curvature pair update counter, $F_i = \nabla E(\mathbf{w}_k)\nabla E(\mathbf{w}_k)^T$ and $|F|$ is the number of F_i entries present in F . In practice, $\mathbf{y}$ is computed without explicitly constructing the $\nabla E(\mathbf{w}_k)\nabla E(\mathbf{w}_k)^T$ matrix. Hence it is sufficient to just store $\nabla E(\mathbf{w}_k)$. The curvature pairs $(\mathbf{s}, \mathbf{y})$ are computed every L steps and stored in (S, Y) buffer only if they are sufficiently large. Further, adaQN performs a control condition by comparing the error at current and previous aggregated weights on a monitoring dataset. If the current error is larger than the previous error by a factor γ , the aFIM and curvature pair buffers are cleared and the weights are reverted to the previous aggregated weights. This heuristic though adds to the cost, avoids further deterioration of the performance due to noisy or stale curvatures.

3. Proposed algorithm

The Nesterov’s accelerated quasi-Newton (NAQ) [19] method achieves faster convergence compared to the standard quasi-Newton methods by quadratic approximation of the objective function at $\mathbf{w}_k + \mu_k \mathbf{v}_k$ and by incorporating the Nesterov’s accelerated gradient $\nabla E(\mathbf{w}_k + \mu_k \mathbf{v}_k)$ in its Hessian update. μ_k represents the momentum with a value $0 < \mu < 1$ and is usually chosen closer to 1 [25]. The update vector of NAQ can be written as

$$\mathbf{v}_{k+1} = \mu_k \mathbf{v}_k + \alpha_k \mathbf{g}_k, \quad (16)$$

$$\mathbf{g}_k = -\mathbf{H}_k \nabla E(\mathbf{w}_k + \mu_k \mathbf{v}_k). \quad (17)$$

The Hessian matrix $\mathbf{H}_{k+1}$ is updated using

$$\mathbf{H}_{k+1} = (\mathbf{I} - \rho_k \mathbf{s}_k \mathbf{y}_k^T) \mathbf{H}_k (\mathbf{I} - \rho_k \mathbf{y}_k \mathbf{s}_k^T) + \rho_k \mathbf{s}_k \mathbf{s}_k^T, \quad (18)$$

where

$$\rho_k = 1/\mathbf{y}_k^T \mathbf{s}_k, \quad (19)$$

$$\mathbf{s}_k = \mathbf{w}_{k+1} - (\mathbf{w}_k + \mu_k \mathbf{v}_k), \quad (20)$$

$$\mathbf{y}_k = \nabla E(\mathbf{w}_{k+1}) - \nabla E(\mathbf{w}_k + \mu_k \mathbf{v}_k). \quad (21)$$

Note that the curvature information pair $(\mathbf{s}_k, \mathbf{y}_k)$ calculated in Eq. (20) and Eq. (21) is different from the BFGS method discussed earlier as it contains $\mu_k \mathbf{v}_k$, an additional term with momentum. From Eq. (21) it can be noted that NAQ involves two gradient calculations per-iteration i.e. the normal gradient $\nabla E(\mathbf{w}_{k+1})$ and the Nesterov’s accelerated gradient $\nabla E(\mathbf{w}_k + \mu_k \mathbf{v}_k)$. Though this increases the number of function evaluations per-iteration, it is well compensated by faster convergence.

In the limited memory form, LNAQ [26] uses the last m_L curvature pairs for the Hessian calculation. The curvature pairs that are used incorporate the momentum and Nesterov’s accelerated gradient term, thus accelerating LBFGS. The search direction $\mathbf{g}_k$ is computed using the two-loop recursion as in Algorithm 1.

3.1 Adaptive stochastic Nesterov’s accelerated quasi-Newton (aSNAQ)

In this paper, we propose a stochastic QN method by combining (L)NAQ and adaQN. The proposed method - adaptive stochastic Nesterov’s accelerated quasi-Newton (aSNAQ) incorporates the Nesterov’s accelerated gradient term and a simple adaptively tuned momentum term. The algorithm is shown in Algorithm 3.

The search direction $\mathbf{g}_k = -\mathbf{H}_k \nabla E(\mathbf{w}_k + \mu \mathbf{v}_k)$ is computed using the two-loop recursion (Algorithm 1). Similar to adaQN, aSNAQ also initializes $\mathbf{H}_k^{(0)}$ based on accumulated gradient information as shown in Eq. (22)

$$[H_k^{(0)}]_{ii} = \frac{1}{\sqrt{\sum_{j=0}^k \nabla E(\mathbf{w}_j)_i^2 + \epsilon}}, \quad (22)$$

and uses an accumulated Fisher matrix aFIM for computing the curvature information pair $(\mathbf{s}, \mathbf{y})$ for the Hessian computation as shown in Eq. (23) and Eq. (24)

Algorithm 3 Proposed Algorithm - aSNAQ

Require: minibatch X_k , μ_{min}, μ_{max} , k_{max} , aFIM buffer F of size m_F and curvature pair buffer (S, Y) of size m_L , momentum update factor ϕ

Initialize: $\mathbf{w}_o = \mathbf{w}_k \in \mathbb{R}^d$, μ , $\mathbf{v}_k$, $\mathbf{v}_o$, $\mathbf{w}_s$, $\mathbf{v}_s$, $k = 0$ and $t = 0$

```

1: while  $k < k_{max}$  do
2:   Calculate  $\nabla E(\mathbf{w}_k + \mu \mathbf{v}_k)$ 
3:   Determine  $\mathbf{g}_k$  using  $\nabla E(\mathbf{w}_k + \mu \mathbf{v}_k)$  in Algorithm 1
4:    $\mathbf{g}_k = \mathbf{g}_k / \|\mathbf{g}_k\|_2$  ▷ Direction normalization
5:    $\mathbf{v}_{k+1} \leftarrow \mu \mathbf{v}_k + \alpha_k \mathbf{g}_k$ 
6:    $\mathbf{w}_{k+1} \leftarrow \mathbf{w}_k + \mathbf{v}_{k+1}$ 
7:   Calculate  $\nabla E(\mathbf{w}_{k+1})$  and store in  $F$ 
8:    $\mathbf{w}_s = \mathbf{w}_s + \mathbf{w}_k$ 
9:    $\mathbf{v}_s = \mathbf{v}_s + \mathbf{v}_k$ 
10:  if  $\text{mod}(k, L) = 0$  then
11:    Compute average  $\mathbf{w}_n = \mathbf{w}_s / L$  and  $\mathbf{v}_n = \mathbf{v}_s / L$ 
12:     $\mathbf{w}_s = 0$  and  $\mathbf{v}_s = 0$ 
13:    if  $t > 0$  then
14:      if  $E(\mathbf{w}_n) > \gamma E(\mathbf{w}_o)$  then
15:        Clear  $(S, Y)$  and  $F$  buffers
16:        Reset  $\mathbf{w}_k = \mathbf{w}_o$  and  $\mathbf{v}_k = \mathbf{v}_o$ 
17:        Update  $\mu = \max(\mu / \phi, \mu_{min})$ 
18:        continue
19:      end if
20:       $\mathbf{s} = \mathbf{w}_n - \mathbf{w}_o$ 
21:       $\mathbf{y} = \frac{1}{|F|} \left( \sum_{i=1}^{|F|} F_i \cdot \mathbf{s} \right)$ 
22:      Update  $\mu = \min(\mu \cdot \phi, \mu_{max})$ 
23:      if  $\mathbf{s}^T \mathbf{y} > \epsilon \cdot \mathbf{y}^T \mathbf{y}$  then
24:        Store curvature pairs  $(\mathbf{s}, \mathbf{y})$  in  $(S, Y)$ 
25:      end if
26:    end if
27:    Update  $\mathbf{w}_o = \mathbf{w}_n$  and  $\mathbf{v}_o = \mathbf{v}_n$ 
28:     $t \leftarrow t + 1$ 
29:  end if
30:   $k \leftarrow k + 1$ 
31: end while

```

$$\mathbf{s} = \mathbf{w}_t - \mathbf{w}_{t-1}, \quad (23)$$

$$\mathbf{y} = \frac{1}{|F|} \sum_{i=1}^{|F|} F_i \cdot \mathbf{s}, \quad (24)$$

where $\mathbf{w}_t$ is the average aggregated weight, t is the curvature pair update counter, $F_i = \nabla E(\mathbf{w}_{k+1}) \nabla E(\mathbf{w}_{k+1})^T$ and $|F|$ is the number of F_i entries present in F . The maximum size of the aFIM buffer F is denoted as m_F . The $\mathbf{y}$ vector is computed without explicitly constructing the $\nabla E(\mathbf{w}_{k+1}) \nabla E(\mathbf{w}_{k+1})^T$ matrix by just storing the $\nabla E(\mathbf{w}_{k+1})$ vector. Note that in aSNAQ, the Nesterov's accelerated gradient $\nabla E(\mathbf{w}_k + \mu_k \mathbf{v}_k)$ is used in the computation of the search direction while the normal gradient $\nabla E(\mathbf{w}_{k+1})$ is stored in the aFIM for Hessian approximation. Thus, aSNAQ also involves two gradient computations per iteration just like in NAQ.

The curvature pairs are computed every L steps and stored in the $(\mathbf{S}, \mathbf{Y})$ buffer only if sufficiently large. The size of the $(\mathbf{S}, \mathbf{Y})$ buffer is set to m_L . The weights are aggregated and updated every L steps. The curvature pair computed based on aggregated weights and Hessian-vector product reduces the effect of noise [16]. Further, updating the Hessian information once every L iteration also reduces the computation cost. The momentum term μ is tuned by a momentum update factor ϕ as shown in step 22 of Algorithm 3. aSNAQ also performs an error control check as shown in steps 14–18. In addition to resetting the aFIM and curvature pair buffers and restoring old parameters, the momentum is also scaled down (step 17). Thus, there is an adaptive tuning of the momentum parameter μ in the range (μ_{min}, μ_{max}) . Unlike adaQN, the error control check is carried out on the same mini-batch. Further, to improve the stability and solve the vanishing and exploding gradient issue, we introduce direction normalization in step 4 as

$$\mathbf{g}_k = \frac{\mathbf{g}_k}{\|\mathbf{g}_k\|_2}, \quad (25)$$

where $\mathbf{g}_k$ is the l_2 norm of the search direction $\mathbf{g}_k$. Normalizing the search direction at each iteration ensures that the algorithm does not move too far away from the current objective [27].

4. Simulation results

In this section, we evaluate the performance of the proposed method on benchmark problems. The results of the proposed aSNAQ algorithm are compared with first-order Adagrad [28], Adam [29] and second-order adaQN [16] algorithms. The simulations are performed using Tensorflow. For all simulations, we choose the aFIM buffer F size as $m_F = 100$ and the limited memory size for the curvature pairs as $m_L = 10$. The update frequency is chosen to be $L = 5$, learning rate $\alpha = 0.01$ and $\gamma = 1.01$. The momentum update factor ϕ is set to 1.1. All weights are initialized with random normal distribution with zero mean and 0.01 standard deviation. The activation function used is tanh. The hyperparameters of Adagrad and Adam were set to their default values. The performance metrics used for evaluation are accuracy and error. Accuracy is evaluated as a percentage of the number of correct predictions by the RNN compared to the expected output while the error is evaluated by the error function defined for the problem. For all simulations, softmax cross entropy error function is used.

4.1 Sequence counting problem

The performance of the proposed method is first evaluated on a toy example problem of sequence counting. Given a binary string (a string with just 0s and 1s) of length T , the task is to determine the count of 1s in the binary string. A simple one layer RNN with 24 hidden neurons is chosen. The batch size is set to $b = 50$, sequence length $T = 20$, $\mu_{min} = 0.1$ and $\mu_{max} = 0.99$. Figure 1 shows the mean squared error (MSE) over 75 epochs. It can be observed that the proposed method clearly outperforms adaQN and Adagrad. On comparison with Adam, aSNAQ is faster in the initial iterations and becomes gradually close to Adam.

Fig. 1. MSE for sequence counting problem.

Fig. 2. Error and accuracy for 28×28 MNIST row by row sequence on training data.

4.2 Image classification

RNNs can be used in image classification problems as well. Since RNNs require a sequence input, for image classification problems, the image is broken into a sequence of pixel values. There are two ways in sequencing images – row-by-row sequence and pixel-by-pixel sequence. In row-by-row sequencing, at each timestep one row is fed as input to the RNN while in pixel-by-pixel sequencing, at each timestep one pixel value is fed as input to the RNN in scanline order starting from the top left to the bottom right pixel.

4.2.1 Results on 28×28 MNIST row by row sequence

We study the performance of the proposed algorithm on the standard MNIST [30] image classification problem. The input to the RNN is 28 pixels fed row-wise at each time step, with a total of 28 time steps. We choose batch size $b = 128$, $\mu_{min} = 0.1$, $\mu_{max} = 0.99$. A single layer RNN with 100 hidden neurons is used. Figure 2 shows the training error and accuracy over 35 epochs. As seen from the results, we can observe that Adagrad performs poorly and stagnates close to its initial error value. On comparing the results with Adam and adaQN, it can be noted that aSNAQ performs better than adaQN and is almost on par with Adam.

4.2.2 Results on 28×28 MNIST pixel by pixel sequence

We further extend to study the performance of the proposed algorithm on pixel-by-pixel sequential MNIST. The pixel-by-pixel sequence based classification is a challenging task where the 784 pixels are fed to the RNN sequentially in scanline order. Since it involves 784 time steps, it is a long range

Fig. 3. Error and accuracy for 28×28 MNIST pixel by pixel sequence on training data.

dependency problem and is much harder compared to the regular classification methods. We evaluate the performance on a simple one layer RNN with 100 hidden neurons. We choose batch size $b = 128$, $\mu_{min} = 0.1$, $\mu_{max} = 0.99$. Figure 3 shows the training error and accuracy over 35 epochs. In pixel by pixel sequence, both Adam and Adagrad methods perform poorly. Though the overall training accuracies are low, aSNAQ shows significant improvement in training compared to adaQN, Adam and Adagrad.

4.3 Character level language modeling

RNNs are widely used in a number of natural language processing tasks. In this paper, we evaluate the performance of the proposed aSNAQ method on character level language modeling problem. The dataset used is The Tale of Two Cities by Charles Dickens. The dataset contains 757,222 characters, split into 80%–20% for train and test samples. The batch size is set to 0.5% of the training set. The vocabulary size, i.e. the number of unique characters including numbers and special characters was 83. The sequence length is set to 50. A 5-layer RNN network with 100 nodes in each layer was used. Figure 4 shows the error and accuracy of the character level modeling over 30 epochs. From the graph it can be observed that the second-order adaQN and aSNAQ methods perform better compared to the first-order Adagrad and Adam. Further, aSNAQ shows significant acceleration compared to adaQN, thus confirming that the proposed aSNAQ method is suitable for training RNNs at a much faster rate.

4.4 Performance on LSTM

To further validate the performance of the proposed method on other RNN architectures, we consider the character level language modeling problem using LSTM architecture. LSTMs (Long short term memory) [10] were especially designed to model long range sequences with long-term dependencies by introducing gate units. A typical LSTM cell consists of an input gate, forget gate and output gate, that filters the information being passed, and thus help in solving the vanishing and exploding gradient issue. In this problem, we consider a two layer LSTM network with 100 hidden neurons each. Same as in section 4.3, the dataset used is The Tale of Two Cities by Charles Dickens. The dataset is split into 80%–20% for train and test samples and the batch size is set to 0.5% of the training set. The vocabulary size is 83 and the sequence length is set to 50. Figure 5 shows the error and accuracy over 30 epochs. The results obtained confirm that the proposed aSNAQ algorithm can be effectively used even for other RNN architectures. It can be observed that in this problem Adam performs the best. However, aSNAQ still performs better than Adagrad and adaQN.

Fig. 4. Error and accuracy for Character Level Language modeling (5-layer RNN) on test data.

Fig. 5. Error and accuracy for Character Level Language modeling on 2-layer LSTM network on test data.

Table I. Summary of Computational and Storage Cost.

Algorithm	Computational Cost	Storage
BFGS	$nd + d^2 + \zeta nd$	d^2
NAQ	$2nd + d^2 + \zeta nd$	d^2
adaQN	$bd + (4m_L + m_F + 2)d + (b + 4)d/L$	$(2m_L + m_F)d$
aSNAQ	$2bd + (4m_L + m_F + 3)d + (b + 4)d/L$	$(2m_L + m_F)d$

4.5 Computational cost

The computation cost is given in Table I. Typical second-order methods such as the BFGS method incur a cost of $nd + d^2 + \zeta nd$ in gradient, Hessian and line-search computation respectively, where $n = |T_r|$ is the number of training samples and d is the number of parameters. In case of NAQ, an additional nd cost is incurred due to twice gradient computation. adaQN and the proposed aSNAQ being stochastic methods, the computation cost in gradient calculation is bd where b is the minibatch size and d is the number of parameters. The Hessian approximation is carried out using the aFIM

and two-loop recursion, thus reducing the computation cost to $(4m_L + m_F + 2)d$. However, the error control check adds to an additional cost of $(b + 4)d/L$. aSNAQ has an additional cost bd and d due to twice gradient computation and direction normalization. Furthermore, the storage cost of BFGS and NAQ is d^2 while adaQN and aSNAQ is $(2m_L + m_F)d$. Overall, the cost of both adaQN and aSNAQ are of the order $O(d)$ complexity and hence comparable to that of first-order methods.

4.6 Discussion

From the above simulation results, it can be observed that the proposed aSNAQ method shows good performance with sufficiently small errors and good accuracies compared to the second-order adaQN and first-order Adagrad and Adam methods. RNNs commonly used in modeling long sequences with long-term dependencies are difficult to train due to the vanishing and exploding gradient issue. Both, the first and second order methods were able to train a vanilla RNN for a simple toy example such as the sequence counting problem. For examples with short dependencies (section 4.1 and 4.2.1), Adam performed the best with errors smaller than aSNAQ. However, as the sequence length or the scale of the network increases, the first-order methods do not perform well on vanilla RNN. On the other hand, both the second-order methods – adaQN and aSNAQ were consistent and effective in training the RNN. For the same language modeling problem described in section 4.3 and 4.4, Adam performed the best on the 2-layer LSTM network, whereas on the 5-layer vanilla RNN it performed the worst. This could be possibly because LSTMs are resilient to the vanishing and exploding gradient problem. However, in all the examples, aSNAQ showed better performance compared to adaQN with significantly higher accuracies and low errors. Also, the direction normalization implemented in aSNAQ is similar to gradient clipping, which is a popular solution to the vanishing and exploding gradient issue. Hence, it can be deduced from the consistent performance of aSNAQ that it is efficient in combating the vanishing and exploding gradient issue. On comparing the computation cost, it can be noted that adaQN and aSNAQ have a low per-iteration cost of the order $O(d)$ since $m_L, m_F \ll d$ and is comparable in terms of the order of computational complexity of first-order methods. Though aSNAQ has a slightly higher computation cost compared to adaQN, it is well compensated by its accelerated performance. Furthermore, aSNAQ shows to be effective not only on vanilla RNNs, but also LSTM and hence suggest a good feasibility to be applied to different problems and architectures. A detailed analysis of the proposed method on more examples and different network structures should be performed to further validate the effectiveness of the proposed aSNAQ algorithm.

5. Conclusion

In this paper we have proposed an adaptive stochastic Nesterov’s accelerated quasi-Newton method for training recurrent neural networks (RNNs). The proposed aSNAQ method is an accelerated method that combines adaQN with NAQ by introducing the Nesterov’s accelerated gradient and momentum term. The proposed method attempts to incorporate second-order curvature information while maintaining a low per-iteration cost which is of the order $O(d)$. The performance of the proposed method was verified on benchmark image classification and language modeling problems. From the simulation results, it was confirmed that incorporating the Nesterov’s accelerated gradient and momentum term improves the performance in the training of RNNs compared to adaQN and other popular first-order methods such as Adagrad and Adam. Further evaluation on LSTM network validates the effectiveness of the proposed method on other architectures as well. Further analysis of the proposed algorithm along with convergence property analysis are left for future work.

References

- [1] I. Sutskever, J. Martens, and G.E. Hinton, “Generating text with recurrent neural networks,” *Proc. 28th ICML’11*, pp. 1017–1024, June 2011.
- [2] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan, “Show and tell: A neural image caption generator,” *Proc. IEEE Conf. CVPR’15*, pp. 3156–3164, 2015.
- [3] A. Karpathy and L. Fei-Fei, “Deep visual-semantic alignments for generating image descriptions,” *Proc. IEEE Conf. CVPR’15*, pp. 3128–3137, 2015.

- [4] D. Bahdanau, K. Cho, and Y. Bengio, “Neural machine translation by jointly learning to align and translate,” *arXiv preprint* arXiv:1409.0473, September 2014.
- [5] I. Sutskever, O. Vinyals, and Q.V. Le, “Sequence to sequence learning with neural networks,” *Advances in Neural Information Processing Systems*, pp. 3104–3112, 2014.
- [6] A. Graves, A.R. Mohamed, and G. Hinton, “Speech recognition with deep recurrent neural networks,” *IEEE Intl. Conference on Acoustics, Speech and Signal Processing*, pp. 6645–6649, May 2013.
- [7] I. Goodfellow, Y. Bengio, and A. Courville, “Deep learning,” MIT press, 2016.
- [8] R. Pascanu, T. Mikolov, and Y. Bengio, “On the difficulty of training recurrent neural networks,” *Proc. ICML’13*, pp. 1310–1318, 2013.
- [9] I. Sutskever, “Training recurrent neural networks,” University of Toronto, Ontario, Canada, January 2013.
- [10] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
- [11] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, “Learning phrase representations using rnn encoder-decoder for statistical machine translation,” *arXiv preprint* arXiv:1406.1078, June 2014.
- [12] J. Martens and I. Sutskever, “Learning recurrent neural networks with hessian-free optimization,” *Proc. 28th ICML’11*, pp. 1033–1040, June 2011.
- [13] A.C. Tsoi, “Gradient based learning methods,” *Intl. School on Neural Networks, Initiated by IIASS and EMFCSC*, Springer, pp. 27–62, 1997.
- [14] C. Peng and G. Magoulas, “Effective modification of the bfgs method for training recurrent neural networks,” *Organization of the Conference*, p. 114, September 2007.
- [15] Q.V. Le, N. Jaitly, and G.E. Hinton, “A simple way to initialize recurrent networks of rectified linear units,” *arXiv preprint* arXiv:1504.00941, April 2015.
- [16] N.S. Keskar and A.S. Berahas, “adaQN: An adaptive quasi-newton algorithm for training RNNs,” *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, Springer, pp. 1–16, September 2016.
- [17] J. Martens, “New perspectives on the natural gradient method,” *arXiv preprint* arXiv:1412.1193, 2015.
- [18] J. Martens and R. Grosse, “Optimizing neural networks with kronecker-factored approximate curvature,” *Proc. ICML’15*, pp. 2408–2417, June 2015.
- [19] H. Ninomiya, “A novel quasi-newton-based optimization for neural network training incorporating nesterov’s accelerated gradient,” *NOLTA*, vol. 8, no. 4, pp. 289–301, October 2017.
- [20] R.H. Byrd, S.L. Hansen, J. Nocedal, and Y. Singer, “A stochastic quasi-newton method for large-scale optimization,” *SIAM Journal on Optimization*, vol. 26, no. 2, pp. 1008–1031, 2016.
- [21] S. Indrapriyadarsini, S. Mahboubi, H. Ninomiya, and H. Asai, “An adaptive stochastic nesterov accelerated quasi newton method for training RNNs,” *Proc. NOLTA’19*, pp. 208–211, December 2019.
- [22] D.E. Rumelhart, G.E. Hinton, and R.J. Williams, “Learning representations by back-propagating errors,” *Nature*, vol. 323, no. 6088, pp. 533–536, October 1986.
- [23] P.J. Werbos, “Backpropagation through time: what it does and how to do it,” *Proc. the IEEE*, vol. 78, no. 10, pp. 1550–1560, October 1990.
- [24] J. Nocedal and S.J. Wright, “Numerical optimization,” Springer, second edition, 2006.
- [25] I. Sutskever, J. Martens, G. Dahl, and G.E. Hinton, “On the importance of initialization and momentum in deep learning,” *Proc. ICML’13*, pp. 1139–1147, February 2013.
- [26] S. Mahboubi and H. Ninomiya, “A novel training algorithm based on limited-memory quasi-newton method with nesterov’s accelerated gradient in neural networks and its application to highly-nonlinear modeling of microwave circuit,” *IARIA Intl. Journal on Advances in Software*, vol. 11, no. 3-4, pp. 323–334, 2018.
- [27] Y. Li and H. Liu, “Implementation of stochastic quasi-newton’s method in pytorch,” *arXiv preprint* arXiv:1805.02338, May 2018.

- [28] J. Duchi, E. Hazan, and Y. Singer, “Adaptive subgradient methods for online learning and stochastic optimization,” *Journal of machine learning research*, vol. 12, pp. 2121–2159, July 2011.
- [29] D.P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint* arXiv:1412.6980, December 2014.
- [30] Y. LeCun, C. Cortes, and C. Burges, “Mnist handwritten digit database,” *AT&T Labs [Online]*, Available: <http://yann.lecun.com/exdb/mnist>, 2010.