ADVANCED ALGORITHMS PROJECT FINAL REPORT: IMPLEMENTING AND EVALUATING NATASHA2 WITH PYTORCH

Haixu Liu, Jiaqi Liu, Ruoshi Liu, Yihao Li
{hl3329, jl5518, rl3111, yl4326}@columbia.edu

ABSTRACT

Natasha 2 is a online optimization algorithm bases on Natasha1.5 and improves it's performance on saddle point by Oja's algorithm and the second order information, the result is an algorithm with better theoretical efficiency ($O(\epsilon^{-3.25})$) compared with SGD($O(\epsilon^{-4})$). In this project, we implemented both Natasha 1.5 and Natasha 2, and then quantitatively evaluated their performances as a PyTorch optimizer in CIFAR-10 and MNIST image classification. Our code is accessible on github: github.com/ruoshiliu/Natasha2.

1 INTRODUCTION

Natasha 2 is derived by Allen-Zhu (2018). In this algorithm, the problem that we are solving is finding approximate local minimum for:

$$f(x) \stackrel{def}{=} \frac{1}{n}\sum_{i=1}^{n} f_i(x)$$

where

1. each $f_i(x)$ is possibly nonconvex but L-smooth,
2. the average $f(x)$ is possibly nonconvex, but second-order smooth with parameter L_2, and
3. the stochastic gradients $\nabla f_i(x)$ have a bounded variance.

On other word, it trys to achieve two goals: 1. find an approximate stationary point; 2. find an approximate local minima when given it's a stationary point.

In order to avoid running into saddle points when trying to minimize a function, classical approach using random perturbation. However, such methods raise two main efficiency issues:

1. Random perturbation is "blind" to second order derivative information of the function
2. We do not need to first arrive the saddle point then move further, is there a way to avoid saddle points without ever moving close to them?

Following are the resolutions with respect to the two issues:

1. Negative eigenvector of hessian $\nabla^2 f(x)$ gives a direction to escape from saddle points. Therefore, Natasha 2 introduces Oja's algorithm to approximate power method, by which find most negative eigenvector. It is an online variant of power method and requires only matrix-vector product computations.
2. If a function is sufficiently smooth,
   - as long as "the position we are at" is close to saddle points, we can use Oja's algorithm to find a negative curvature, and move along its direction to decrease the objective.
   - When the point we are at is not close to any saddle points, with smoothness of the function, we can get a "safe zone" of this point, in which there is no strict saddle point. Using this safe zone, we can design an algorithm that is faster than SGD.

For the first goal, Allen introduce an algorithm named Natasha1.5, which tries to solve a different problem compared to Natasha 2, however, it will return an approximate stationary point of our function in the original point, and it will be the "optimization step" in Natasha 2. The main idea for Natasha 1.5 is that it consists of T' full epochs, and at each full epoch, we compute an average gradient of a batch of samples, that is

$$\nabla f_S(\widetilde{x}) \stackrel{def}{=} \frac{1}{|S|}\sum_{i\in S}\nabla f_i(x)$$

where S is a random subset of [n]. Each epoch is further divided into p sub-epochs, in each sub-epoch we start with a point and conceptually apply SVRG but replacing $f(x)$ with a regularized version, which can be defined as

$$f^S(x) \stackrel{def}{=} f(x) + \sigma||x - \hat{x}||^2$$

Finally, when the sub-epoch is over, we define $\hat{x}$ to be a random one from $\{x_0, ..., x_{m-1}\}$, when a full epoch is over, we define $\hat{x}$ to be the last $\hat{x}$.

Natasha 2 incorporates Oja's algorithm and Natasha 1.5, by using Oja's algorithm, it either finds a vector $v \in \mathbb{R}^d$ such that $v^T\nabla^2 f(y_k)v \leq -\frac{\delta}{2}$, which meaning the current point is near a saddle point, or conclude that $\nabla^2 f(y_k)$ are above $-\delta$, meaning current position are not on a path converging to a saddle point. In the first situation, we can do a little "kick" to the current point, which means a perturbation drive it away from the path leading to the saddle point. Therefore, the optimization path will have less chance reaching the saddle point, which intuitively means that the whole optimization path will reach the ideal minima more efficiently. In the second situation, we can simply apply Natasha 1.5 for one full epoch. The whole algorithm is described below:

1. Start from a initial vector $y_0 \in \mathbb{R}^d$ and is divided into iterations $k = 0, 1, ...$
2. In each interation, apply Oja's algorithm and find a vector $v \in \mathbb{R}^d$ such that:
   - if $v^T\nabla^2 f(y_k)v \leq -\frac{\delta}{2}$, we update $y_{k+1} \leftarrow y_k + \frac{\delta}{L_2}v$ or $y_{k+1} \leftarrow y_k - \frac{\delta}{L_2}v$ with probability $\frac{1}{2}$ since we do not care which direction we kick it. This is called second-order step.
   - if $\nabla^2 f(y_k) \succeq -\delta I$ we define a new function $F(x) = F^k(x) \stackrel{def}{=} f(x) + L(max\{0, ||x - u_k|| - \frac{\delta}{L_2}\})^2$, and apply Natasha 1.5 to get closer to a stationary point. This is called a first-order step.

   terminate the iteration when a specific number of first-order steps are met.
3. select a random $\hat{y}$ along the first order steps, prune it using convex SGD.

which achieves a time efficiency of $\tilde{O}(\epsilon^{-3.25})$ for finding $(\epsilon, \epsilon^{\frac{1}{4}})$-approximate local minima, which is better than stochastically controlled stochastic gradient (SCSG). According to Ge et al.'s work, for current SGD, there must be a ϵ^{-4} factor to find stationary point. Thus Natasha2 outperforms current SGD as well.

1.1 Explanation of variables and input parameters

There are numbers of variables and parameters defined by the author, some of them relate to the assumption of the problem, some of them are predefined hyper-parameters. In this section, we will give a general description about all these variables and parameters.

1. We define $f(x), F(x), \varphi(x)$ as the functions that we are studying, where in Natasha 1.5, $\varphi(x)$ is proper convex while each $f_i(x)$ is possibly non-convex and $F(x) \stackrel{def}{=} \varphi(x) + f(x) \stackrel{def}{=} \varphi(x) + \frac{1}{n}\sum_{i=1}^n f_i(x)$. While in Natasha 2, we define $\varphi(x) = 0$.
2. parameter ϵ: the approximation coefficient, used for finding approximate stationary point, in other word, in order to realize the first goal mentioned above, we just need to find a point where $||\nabla f(x)|| \leq \epsilon$.
3. V means a bounded variance, it is an assumption that need to be satisfied in Natasha 1.5, however, we do not have such assumption for Natasha 2, which is $\mathbb{E}_{i\in R[n]}[||\nabla f_i(x) - \nabla f(x)||^2] \leq V$.
4. σ means our function f is σ-strongly convex, L means our function f is L-Lipschitz smooth, which are same as we have learned in class.
5. L_2 means second-order smooth, which means $||\nabla^2 f(x) - \nabla^2 f(y)|| \leq L_2||x - y||$, in our experiemnt, we can simply assume $V = L = L_2 = 1$.

6. In Oja's algorithm, we define parameter δ, if the Hessian $\nabla^2 f(x) < -\delta$, we say that the current point is a saddle point.
7. In Natasha 1.5, epoch length is denoted as $B \in [n]$, sub-epoch count is denoted as $p \in [B]$, epoch count is denoted as $T' \geq 1$, learning rate is denoted as $\alpha > 0$.

2 Related Work: Other optimizer

We also try to find other optimizers that are suitable for solving a non-convex function optimization problem, we try to find out their ideas and difference compared to Natasha 2. Some discussion about other optimizer in recent years

- diffGrad (Dubey et al. (2019)): it is an improvement of the recent optimization techniques such as Adam and AMSGrad, these methods suffer from the problem of automatic adjustment of the learning rate. The main problem is with controlling friction for the first moment in order to avoid over shooting near to an optimum solution. The diffGrad optimization technique is based on the change in short-term gradients and controls the learning rate based on the need of dynamic adjustment of learning rate. The parameter update should be smaller in low gradient changing regions and vice-versa. DiffGrad compute the 1^{st} and 2^{nd} order moments for the i^{th} parameter at the t^{th} iteration similar to Adam. The mean idea is to introduce a diffGrad friction coefficient(DFC) to control the learning rate using information of short-term gradient behavior. The DFC is represented by ξ and defined as $\xi = AbsSig(\Delta g_{t,i})$, $AbsSig$ is a non-linear sigmoid function and $\Delta g_{t,i}$ is the change in gradient between immediate past the current iteration. The same as Adam, it uses moment variables to avoid saddle points.
- Neumann optimizer (Krishnan et al. (2017)): Neumann optimizer use the second order derivative information from another approach: it used a linear approximation to the inverse of hessian, and use the batch estimator as a unbiased estimator to the whole data. It gave a very tidy final result in pure linear operations form, meanwhile, its unique setting can have a extremely large batch (32000 sample per batch in the article), which result in a 10%-30% decrease in the training epochs, alternatively, when have the training epochs same, it will result in a 0.9% improvement on accuracy when compared with RMSProp.
- LARS(You et al. (2017)): LARS (Layer-wise Adaptive Rate Scaling) is a novel algorithm designed for the training with large batch size without loss in accuracy by adapting learning rate for each layer. It uses a separate learning rate for each layer to guarantee stability and controls the magnitude of the update with respect to the weight norm to control the training speed. The update for per-layer learning rate is as follow:
$$\lambda^l \leftarrow \frac{||w_t^l||}{||g_t^l|| + \beta||w_t^l||}$$
where g_t^l is the current stochastic gradient, w_t^l is the current weights and β is the weight decay. LARS can scale Alexnet to a batch size of 8K as well as Resnet-50 to a batch size of 32K with negligible loss in accuracy.

3 Method

3.1 Torch Optimizer Class

To implement Natasha2, we decided to use PyTorch as our framework. For this project, we focus on the optimizer module in which other commonly-used non-convex optimization algorithm has been implemented including SGD, SGD-momentum, Adam, etc. So far, from documentation of the torch.optim module and here's the structure of the module with explanation of variables:

```
class Optimizer(object)

    def initialize(parameters, defaults)
```

To initialize an optimizer class, one needs to pass in the 'parameter' variable, which by default is an iterable variable containing the 'torch.Tensor' to be optimized.

```
    def getstate()
```

This method wrap three variables in a dictionary: 'defaults', 'state', 'param_groups' and return the dictionary.

```
def setstate(state)
```

'state' stores the state of the optimizer which contains temporal information of the optimization process.

```
def state_dict()
```

This function returns the state of the optimizer and the model parameters in a dictionary.

```
def load_state_dict(state_dict)
```

This function loads the state dictionary into the optimizer. This method is particularly useful in resuming a training process. The gradient and version history of the model parameters will be reloaded to continue the optimization process.

```
def zero_grad()
```

Initialize the gradients of the parameters as zero.

```
def step(closure)
```

This is the main function we are going to implement in this project. The method should initialize the state in the first iteration according to Natasha2's initialization process. It should then perform updates of the parameters.

In addition, if closure is set to 'True', the model will evaluate the weights and return a loss value.

3.2 Experimental Setup

To test the efficiency, we will have several widely used optimizer as the benchmark. Among all the build-in optimizers implemented in torch.optim module, we selected widely used Adam algorithm and SGD(with and without momentum version).

The test of our implementation will be tested on the image classification problem on MNIST and CIFAR-10 datasets. Both datasets has 60000 samples in 10 classes, MNIST is made of 28*28*1 gray-scale images while and CIFAR-10 is of 32*32*3 RGB-channel images.

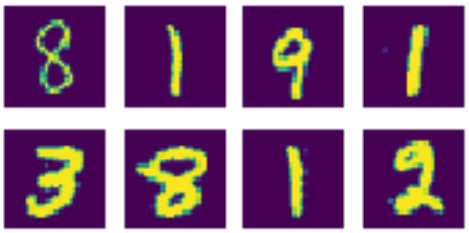

Figure 1: MNIST image sample

The classification task will be operated on the architecture on different scales of numbers of parameters. Because of the restricted computational resources, we use LeNet (LeCun et al.) and ResNet-18(He et al. (2016)), cross-entropy as the loss function.

In order to create a quantitative criterion measuring the efficiency, we followed Rick Wierenga's idea (Wierenga), tracking the performance criterion of the architecture after some specific numbers of epochs.

Considering the large number of variable(when compared with the number of parameters), we use a relatively unique training setting: For MNIST, in each epoch we select 10% of the training set by bootstrapping, run 30 epochs for LeNet, 10 epochs for ResNet-18 (the number is decided based on the converge speed); For CIFAR-10, we choose a more typical setting: select 90% of the training set by bootstrapping run 20 epochs for ResNet (LeNet cannot take CIFAR10 and the result is not informative at all), and compare the learning

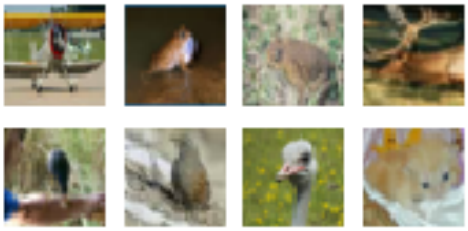

Figure 2: CIFAR-10 image sample

curve (loss vs. number of epochs) between Adam, SGD, SGD-momentum, and Natasha algorithm. For specific hyperparamaters, please check the appendix.

3.3 ALGORITHM SIMPLIFICATION

The original Natasha 2, when unified with Nathasha 1.5 and Oja's algorithm, is as following:

After having the theoretical form, we met problems as following:

1. For various neural networks with arbitrary loss function, the assumption on smoothness, convexity, variance bounds are unknown.
2. A lot of value are given in form of $\Theta(\cdot)$ form, the actual value is undetermined.
3. In PyTorch, The stopping criterion N_1 will not be counted in a training process.

To simplify the algorithm, we use the following settings:

- Line 1-11: initialization
  1. Assumption: to simplify all the terms, we set $V = L = L_2 = 1$, which is proven working in practice
  2. line 1-9: As a neural network optimizer, we no longer have the restriction on ϵ, thus we no longer define B, p, and α as a dependent of ϵ, all of these variables are set as the hyper-parameters.
  3. line 10: follows the author's instruction, no longer use list X as a record of past step, and return the last result as the final output instead of a random pair in X.
  4. line 11: As a optimizer, we no longer need stop criterion.
- line 13-18: Oja's Algorithm
  1. for $v_{t+1} = M_i^* \times v_t$ will greatly increase the magnitude of v(in practice, by 10^3 in each entries), which will have PyTorch's float32 tensor exceed the upper limit in less than 10 steps, thus we normalize the vector v_i by its Frobenius norm each time.
  2. line 15: we used $\frac{1}{\delta^2}$ as the actual value of $O(\frac{1}{\delta^2})$. However, this value needs further discussion.
- line 22-36 (Natasha 1.5)
  1. From our experiment, Natasha 1.5 can be a valid optimizer for neural networks, thus we did not add the additional convex term to the function.
  2. we replaced the original $\arg\min$ operation with an update in the analytical solution form: $x_{t+1} = x_t - \alpha\tilde{\nabla}$.
  3. follows the author's instruction: $\hat{x}$ will be updated as the average of $\{x_0, x_1, ..., x_{m-1}\}$ instead of a random choice.
- follows the author's instruction: we no longer hold a list X in Natasha 1.5 we simply update $y_{k+1} \leftarrow \hat{x}$, the last y_{k+1} will be the output of Natasha 2.

After all of these operation, we obtained the following implementation version of Natasha 1.5 and Natasha 2:

Some intuition to all the hyperparameters:

Algorithm 1 Natasha2-Original(f, y_0, ϵ, δ)

if $1 \geq \frac{\delta}{\epsilon^{\frac{1}{3}}}$ **then**
  $\tilde{L} = \tilde{\sigma} \leftarrow \Theta(\frac{\epsilon^{\frac{1}{3}}}{\delta}) \geq 1$
else
  $\tilde{L} \leftarrow 1$ and $\tilde{\sigma} \leftarrow \Theta(\max\{\frac{\epsilon}{\delta^3}, \epsilon\}) \in [\delta, 1]$
end if
$B \leftarrow \Theta(\frac{1}{\epsilon^2})$
$p \leftarrow \Theta((\frac{\tilde{\sigma}^2}{\tilde{L}^2}B)^{\frac{1}{3}})$
$\alpha \leftarrow \Theta(\frac{\tilde{\sigma}}{p^2\tilde{L}^2})$
$m \leftarrow B/p$
X = []
$N_1 = \Theta(\frac{\tilde{\sigma}\Delta_f}{p\epsilon^2})$, where Δ_f is an upper bound on $f(y_0) - \min_y f(y)$
for $k \leftarrow 0$ to ∞ **do**
  $v_0 \leftarrow$ random vector
  $M_i = -\nabla^2 f_i(y_k)$
  for $t \leftarrow 0$ to $O(1/\delta^2)$ **do**
    $v_{t+1} = (I + \eta \dot{M}_i) \times v_t$
  end for
  $v_T = v_T/||v_T||$
  if ${v_T}^T \nabla^2 f(y_k) v_T \leq -\frac{\delta}{2}$ **then**
    $y_{k+1} \leftarrow y_k + (-1)^j \delta v_T$, where j is a random integer
  else
    $F(x) = F^k(x) \stackrel{def}{=} f(x) + L(\max\{0, \|x - y_k\| - \delta\})^2$
    $\hat{x} \leftarrow y_k; \hat{X} \leftarrow []$
    $\tilde{x} \leftarrow \hat{x}; \mu \leftarrow \frac{1}{B}\sum_{i \in S} \nabla f_i(\tilde{y})$ where S is a uniform random subset of $[n]$ with $|S| = B$
    for $s \leftarrow 0$ to $p - 1$ **do**
      $x_0 \leftarrow \hat{x}; \hat{X} \leftarrow [\hat{X}, \hat{x}]$
      for $t \leftarrow 0$ to $m - 1$ **do**
        $i \leftarrow$ a random index from $[n]$.
        $\tilde{\nabla} \leftarrow \nabla f_i(x_t) - \nabla f_i(\tilde{x}) + \mu + 2\sigma(x_t - \hat{x})$
        $x_{t+1} = \arg\min_{y \in \mathbb{R}^d}\{\phi(y) + \frac{1}{2\alpha}||y - x_t||^2 + \langle\tilde{\Delta}, y\rangle\}$
      end for
      $\hat{x} \leftarrow$ average of $\{x_0, x_1, ..., x_{m-1}\}$
    end for
    $\hat{y}_k \leftarrow$ the last vector of $\hat{X}$
    $y_{k+1} \leftarrow \hat{x}$
    X append $(y_k, \hat{y}_k)$
    break the for loop if have performed N1 first-order steps.
  end if
end for
return a random pair in X

- B: the total gradient descent operated.
- p: the number of average gradient descent/number of actual update.
- α: "learning rate", but will be mitigated by m = B/p, the lower m, the stronger the mitigation is.
- δ: "prophet coefficient" the coefficient controls Oja's algorithm's part and affect the threshold deciding if the algorithm "kick" or "optimize".
- η: the coefficient controls how strong the hessian participated into the Oja's algorithm.

During the training process, as our expectation, the computation on Hessian is extremely slow. Followed the author's guide(Allen-Zhu (2018)), we use $\lim_{q \to 0+} \frac{\nabla f(x+qk) - \nabla f(x)}{q}$ as an approximation to $\nabla^2 f(x) \cdot k$, and

Algorithm 2 Natasha1.5-implementation(f, y_0, B, p, α)

$m \leftarrow B/p$
for $k \leftarrow$ 0 to k^* **do**
  $\hat{x} \leftarrow y_k$
  for $s \leftarrow 0$ to $p-1$ **do**
    $x_0 \leftarrow \hat{x}$;
    for $t \leftarrow 0$ to $m-1$ **do**
      $\tilde{\nabla} \leftarrow \nabla f_i(x_t) + \sigma(x_t - \hat{x})$
      $x_{t+1} = x_t - \alpha\tilde{\nabla}$
    end for
    $\hat{x} \leftarrow$ average of $\{x_0, x_1, ..., x_{m-1}\}$
  end for
  $y_{k+1} \leftarrow \hat{x}$
end for
return y_{k^*+1}

Algorithm 3 Natasha2-implementation($f, y_0, B, p, \alpha, \delta, \eta$)

$m \leftarrow B/p$
for $k \leftarrow$ 0 to k^* **do**
  $v_0 \leftarrow$ random vector
  $M_i = -\nabla^2 f_i(y_k)$
  $M_i^* = (I + \eta\dot{M}_i)$
  for $t \leftarrow 0$ to $(1/\delta^2)$ **do**
    $v_{t+1} = M_i^* \cdot v_t$
    $v_T = v_T/||v_T||$
  end for
  if ${v_T}^T \nabla^2 f(y_k) v_T \leq -\frac{\delta}{2}$ **then**
    $y_{k+1} \leftarrow y_k + (-1)^j \delta v_T$, where j is a random integer
  else
    $\hat{x} \leftarrow y_k$
    for $s \leftarrow 0$ to $p-1$ **do**
      $x_0 \leftarrow \hat{x}$;
      for $t \leftarrow 0$ to $m-1$ **do**
        $\tilde{\nabla} \leftarrow \nabla f_i(x_t) + \sigma(x_t - \hat{x})$
        $x_{t+1} = x_t - \alpha\tilde{\nabla}$
      end for
      $\hat{x} \leftarrow$ average of $\{x_0, x_1, ..., x_{m-1}\}$
    end for
    $y_{k+1} \leftarrow \hat{x}$
  end if
end for
return y_{k^*+1}

review line 4-11 as follows

$$\begin{aligned} v_{t+1} &= M_i^* \cdot v_t \\ &= (I - \eta\nabla^2 f_i(y_k))v_t \\ &\approx I \cdot v_t - \eta \lim_{q \to 0+} \frac{\nabla f(x + qv_t) - \nabla f(x)}{q} \end{aligned}$$

We have the following version without evaluations to Hessian, in this version we set q = 0.0001.

All the implementation can be seen in the github repo: https://github.com/ruoshiliu/Natasha2 with examples. Because of the setting of torch.optim, we need some additional helper function, please check util.py.

Algorithm 4 Natasha2-hessian-vector-product-implementation($f, y_0, B, p, \alpha, \delta, \eta$)

$m \leftarrow B/p$
for $k \leftarrow$ 0 to k^* **do**
  $v_0 \leftarrow$ random vector
  for $t \leftarrow 0$ to $(1/\delta^2)$ **do**
    $v_{t+1} = I \cdot v_t - \eta \frac{\nabla f(x+qv_t) - \nabla f(x)}{q}$
    $v_T = v_T / ||v_T||$
  end for
  if ${v_T}^T \frac{\nabla f(x+qv_t) - \nabla f(x)}{q} \leq -\frac{\delta}{2}$ **then**
    $y_{k+1} \leftarrow y_k + (-1)^j \delta v_T$, where j is a random integer
  else
    $\hat{x} \leftarrow y_k$
    for $s \leftarrow 0$ to $p-1$ **do**
      $x_0 \leftarrow \hat{x}$;
      for $t \leftarrow 0$ to $m-1$ **do**
        $\tilde{\nabla} \leftarrow \nabla f_i(x_t) + \sigma(x_t - \hat{x})$
        $x_{t+1} = x_t - \alpha\tilde{\nabla}$
      end for
      $\hat{x} \leftarrow$ average of $\{x_0, x_1, ..., x_{m-1}\}$
    end for
    $y_{k+1} \leftarrow \hat{x}$
  end if
end for
return y_{k^*+1}

4 Experimental Result

4.1 Natasha 1.5

Different from $p = O(B^{\frac{1}{3}})$ provided in original paper, B and p presented a strong linear correlation for all the valid setting: when $m = \frac{B}{p} \geq 4$(number of inner loops), Natasha1.5 tends to be "stuck" at some point even when momentum term σ is set to 0. This performance is kind of abnormal and may need further discussion in the future.

Figure 3: Different optimizer's learning curve on LeNet, MNIST dataset

For a quantitative comparison, we visualized the training loss against epoch number, as a more complicated mean-form SGD, Natasha1.5 naturally gave a more steady pattern than SGD and Adam. In LeNet, it converges to a significant lower loss. With a momentum term $\sigma = 0.5$, its performance sometimes can be on par with SGD-momentum with $momentum = 0.9$. In ResNet, Natasha1.5's performance is generally the same as all the other benchmarks. LeNet is too small for CIFAR10 classification, thus the result is not very informative.

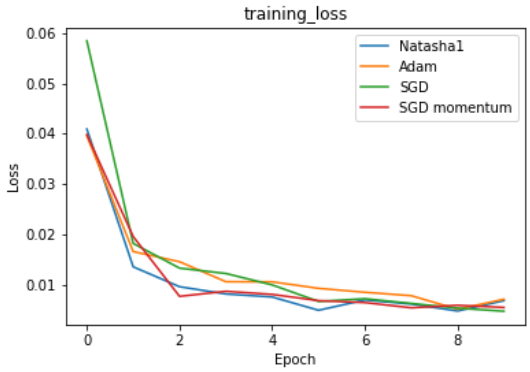

Figure 4: Different optimizer's learning curve on ResNet, MNIST dataset

Figure 5: Different optimizer's learning curve on LeNet, CIFAR10 dataset

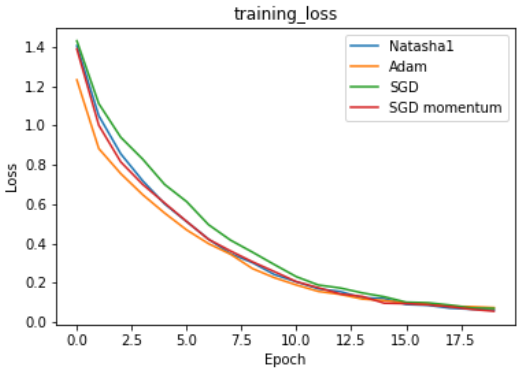

Figure 6: Different optimizer's learning curve on ResNet, CIFAR10 dataset

4.2 NATASHA 2

For the version with actual hessian, calculating the hessian for any often used neural network (ResNet. EfficientNet, etc.)is, as other paper(Krishnan et al. (2017)) mentioned, practically impossible. We did an experimental result on a very small CNN (n_param = 684) with traditional nested loop calculating the exact hessian, it took unacceptably long time (2 min) for the calculation.

In hessian-vector-product version(Algorithm 4), the efficiency is greatly improved: for LeNet with n_param = 684,the time cost is omitable (compared with about 2 minutes). For ResNet18 with n_param $\approx$11M, the time

cost is in tens of seconds(the exact hessian is impossible to calculate for this), that's also why we did not put it into the visualization of ResNet result. when $\delta \geq 0.05$ i.e. the number of evaluation in Oja loop ≤ 400. However, considering the number of loop has a quadratic relation to the inverse of δ, we don't recommend setting δ to lower than 0.05.

From all the existed experiment, we did not find a good combination of δ(the threshold deciding "kick" or "optimize") $O(\delta^{-2})$(number of Oja loop), and η(the scaler on hessian matrix) that can have Natasha2 have expected performances. From all the existed result, the "kick" is very frequently operated in the start of training, as the training goes on, the number of "kicking" is diminishing. Meanwhile, the kick is somehow "countering" rather than benefiting the optimization work, which result in the failure in convergence. This result is definitely different from the behaviour expected. The effect of hyperparameters setting on Natasha2 need more computational experiments.

5 Discussion

Because of the limited computational resources, we cannot give a exhaustive search on the all the hyperparamters setting. We implemented $Natasha1.5$, $Natasha2$, $Natasha2\_hp$ algorithm, however, in their application, we only proved that

1. Natasha1.5 is a valid optimizer(say, $B/p \leq 4$)
2. Natasha 2 can be implemented, but not in formal torch.optim way, the code is not in the standard from because the hessian-approximating part is unable to be implement inside the optimizer.step() function).

The exploration to the best default setting on α, δ, η and σ need tough further computational approaches. We may need some techniques like Bayesian optimization, Multi-fidelity optimization, Hyperband, etc.

For Natasha 2, the hyperparameter setting in Oja's algorithm part need to be further adjusted to have it work in the expected way.

References

Zeyuan Allen-Zhu. Natasha 2: Faster non-convex optimization than sgd. In *Advances in neural information processing systems*, pp. 2675–2686, 2018.

Shiv Ram Dubey, Soumendu Chakraborty, Swalpa Kumar Roy, Snehasis Mukherjee, Satish Kumar Singh, and Bidyut Baran Chaudhuri. diffgrad: An optimization method for convolutional neural networks, 2019.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Shankar Krishnan, Ying Xiao, and Rif A Saurous. Neumann optimizer: A practical optimization algorithm for deep neural networks. *arXiv preprint arXiv:1712.03298*, 2017.

Yann LeCun et al. Lenet-5, convolutional neural networks.

Rick Wierenga. An empirical comparison of optimizers for machine learning models. URL `https://heartbeat.fritz.ai/an-empirical-comparison-of-optimizers-for-machine-learning-models-b86f29957050`.

Yang You, Igor Gitman, and Boris Ginsburg. Large batch training of convolutional networks. *arXiv preprint arXiv:1708.03888*, 2017.

6 Appendix

Setting and environments: PyTorch Version 1.4.0

- SGD: learning rate = 0.01
- SGD-momentum: learning rate = 0.01, momentum = 0.9
- Adam: default
- Natasha 1: $\alpha = 0.01, B = 27, p = 9, \sigma = 0.5$