ACCELERATED OPTIMIZATION METHODS ON CONVOLUTIONAL NEURAL NETWORKS

A PREPRINT

Zhiping Xiao (Patricia) *
Department of Computer Science
University of California, Los Angeles
patriciaxiao@g.ucla.edu

June 11, 2019

ABSTRACT

This is the course project report of ECE236C, Spring 2019. I am implementing and testing algorithms that utilize convex optimization methods to accelerate non-convex optimization problems. To be specific, I only focus on CNN (convolutional neural network), and implemented the models in PyTorch. [2] Similar optimizers should also be easily applied to GNN (graph neural network), a very popular extension of CNN on general node-link graphs. The results will be discussed in the discussion section.

***K*eywords** Convex Optimization · Non-convex optimization · CNN

1 Introduction

This is the final report of ECE236C, UCLA, done in 2019 Spring semester, instructed by Prof. Vandenberghe.

I've taken the ECE236 series so as to gain a better understanding in the optimization problems, which is expected to be helpful latter in my research works.

Most recently I focus on deep learning models, which means that I am working on the neural networks (NN), not that easily related to the standard convex optimization problems. However, in practice, we realize that we are actually using convex optimization tricks, such as Stochastic Gradient Descent (SGD). Unlike convex optimization problems, learning on NN could easily fail for many different reasons. Most importantly, it is not robust at all to its hyper-parameters, such as the learning rate. When the learning rate is too big, the model couldn't learn at all. These shortcomings, in my opinion, beside the nature of SGD, has something to do with that they are naturally non-convex models. One experimental evidence of my claim on "it has something to do with the nature of SGD" is that, Newton methods are generally robust to the parameters, given the same model. [3] I am curious about if these improved methods would achieve better robustness in hyper-parameter settings.

My goal is to analyze and implement the algorithm proposed in [1] and [2], together with some modified version and some more basic algorithm (such as the Nesterov's standard acceleration algorithm) and apply it to a simple Convolutional Neural Network (CNN) model to check the outcomes.

Basically, I am interested in:

*This is the ECE236C final project report, Spring 2019, instructed by Prof. Lieven Vandenberghe, and also lots of thanks to TA Xin Jiang.

[2] https://github.com/PatriciaXiao/AcceleratedSGDMethods

[3] Those are conclusions I came up from doing another optimization-related course's projects this quarter, *Optimization Methods for Deep Learning*, course website https://www.csie.ntu.edu.tw/~cjlin/courses/optdl2019/.

- The method's limitations, such as, is it robust, or very sensitive to initialization, are there certain cases where this method will completely not work at all? [4]
- In what cases does each of them works the best.

My implementation of the models is posted on my Github repository at `https://github.com/PatriciaXiao/AcceleratedSGDMethods`.

2 Related Work

Yair et al. proposed an accelerated gradient method in [1], which applies to general optimization problem:

$$\text{minimize} \quad f(x)$$

where $f : \mathbb{R}^d \to \mathbb{R}$ may be non-convex, as long as it has L_1-Lipschitz continuous gradient and L_2-Lipschitz continuous Hessian.

In [2], negative curvature descent is further dig into, and it is claimed to be helpful manner of escaping the small-gradient regions.

There are also other works such as [3], following another trend, introducing a proximal version of an existing primal-dual method and accelerate the method by applying an inner-outer iteration procedure. They focus mostly on simpler problems such as SVM and logistic regression, and it'll be hard to apply this kind of methods to non-convex models with complex structures, such as the neural networks in general.

In [4] the Lipschitz continuity on Neural Network is discussed. Basically they showed that, although the exact Lipschitz computation is NP-hard, there could be an algorithm that could find a Lipschitz upper bound through automatic differentiation, called AutoLip. Furthermore, they applied this algorithm they proposed, together with some detailed analysis on each type of layers in the standard neural networks, and conclude that it is computable. However, neither L_1 nor L_2 bounds was mentioned or estimated.

In another work [5], Lipschitz bound was connected with Neural Networks again, but still,it is hopeless in terms of computing the exact values of the L_1 and L_2 bounds we need.

It would be the best if we make those bounds into parameters. We assume that they exist in the first place.

Another important work to mention is [6], where Nesterov's accelerated gradient method, which is referred to as *"function Accelerated-gradient-descent"* in section 2.1 of [1] and should be regarded as a base method of all the other algorithms, was derived as a momentum method. We could thereby implement the method accordingly, while try rephrasing the other algorithms using similar methods.

There's also some discussions on the limitations of the existing gradient methods [7]. Previously, with a toy CNN model and standard MNIST dataset, Adam + MSE loss doesn't work at times using certain parameters, namely, under certain learning rate. Adam should be, theoretically, a stable and efficient method for acceleration, and looking into the reason why it fails should be helpful.

3 Dataset

There are some standard, light-weight dataset that are guaranteed to have reasonable distribution and good features to learn with. Such as MNIST and CIFAR10.

Those are standard datasets readily available everywhere.

While debugging and playing tricks on small, toy dataset, MNIST is more helpful, since it is really simple, containing only hand-written numbers ($0 \sim 9$) with labels. However, almost all kinds of optimizers, all kinds of settings, work out well on this super-simple dataset. To show the difference among the settings, CIFAR10 dataset is needed, for it is more complex, having more channels (RGB) in the input images (10 categories are harder to distinguish here, such as *bird, car*, etc.).

4 Algorithm

This is also closely related with our course content of *Accelerated proximal gradient methods* (in lecture 7 note).

[4] I am pretty sure that Adam isn't working in some situations.

Among Nesterov's works in the early years [8], there's a very popular Accelerated Gradient method, which is mentioned and based the other algorithms on in [1].

This method was basically proposed in [1] as:

---

function Accelerated−gradient−descent($f, z_1, \varepsilon, L_1, \sigma_1$)

$\kappa = \frac{L_1}{\sigma_1}$

for $j = 1, 2, \ldots, \infty$:
  if $\|\nabla f(z_j)\| \leq \varepsilon$
    return z_j
  else
    $y_{j+1} = z_1 - \frac{1}{L_1}\nabla f(x_j)$
    $z_{j+1} = \left(1 + \frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right) y_{j+1} - \frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1} y_j$

---

where ε and σ_1 are parameters, and L_1 is a bound (f is L_1-smooth), z_1 is an initialized iterate, and f is a strongly convex function in this original expression.

It is also proved that if z_1 satisfies $f(z_1) - \inf_x f(x) \leq \Delta_f$ (Δ_f is the *optimality gap* of f), then the algorithm is guaranteed to converge within finite number of steps, and the upper-bound of the iterations is bounded by the parameters $L_1, \sigma_1, \varepsilon, \Delta_f$.

In our slides, it has another form of expression, where $x_0 = v_0$, $\theta_0 \in (0, 1]$, k being the iterator, and then repeatedly calculate the positive root of the following quadratic equation as θ_k:

$$\frac{\theta_k^2}{t_k} = (1 - \theta_k)\gamma_k + m\theta_k \qquad \gamma_k = \frac{\theta_{k-1}^2}{t_{k-1}}$$

and $y = x_0$ if $k = 0$, update x_k and v_k as:

$$y = x_k + \frac{\theta_k \gamma_k}{\gamma_k + m\theta_k}(v_k - x_k)$$
$$x_{k+1} = \text{prox}_{t_k h}(y - t_k \nabla g(y))$$
$$v_{k+1} = x_k + \frac{1}{\theta_k}(x_{k+1} - x_k)$$

Also in our slides, the momentum interpretation is shown as:

$$y = x_k + \frac{\theta_k \gamma_k}{\gamma_k + m\theta_k}(v_k - x_k) = x_k + \beta_k(x_k - x_{k-1})$$

where

$$\beta_k = \frac{\theta_k \gamma_k}{\gamma_k + m\theta_k}\left(\frac{1}{\theta_{k-1}} - 1\right) = \frac{t_k \theta_{k-1}(1 - \theta_{k-1})}{t_{k-1}\theta_k + t_k \theta_{k-1}^2}$$

This could be also shown following the same method as is proposed in [6], we have another form of Nesterov's Accelerated Gradient, and the proof is in their *Appendix A.1*. That is a different perspective expressing the same standard Nesterov's accelerated gradient method coming to the same conclusion.

All in all, what is so good about this momentum interpretation is that it naturally extend this method so that it could be applied to non-convex cases, and on the other hand, simplified the parameters so as to make it practical for implementation.

Assuming that momentum β_k is constant and learning rate γ is also constant, specifying the maximum number of iterations for learning as T and initialize x as x_0, we simplify and rephrase the algorithm into:

---

function Accelerated−gradient−descent(f, γ, β, x_0, T)
  $y_0 = x_0$
  for $k = 0, 1, 2, \ldots, T-1$:
    $y_{k+1} = x_k - \gamma \nabla f(x_k)$
    $x_{k+1} = y_{k+1} + \beta(y_{k+1} - y_k)$
  return x_T

From [1], the Almost-convex-AGD, as an extension of the standard Accelerated-gradient-descent, basically changes the objective function, so that the optimization problem becomes:

$$\text{minimize } g_j(x) = f(x) + \gamma\|x - x_k\|^2$$

at the k^{th} iteration. The reason for doing so is to make the γ-almost-convex function f being γ-strongly convex. It is also obvious that in recent neural network structures, it is hopeless to compute an exact γ here. But we could estimate its impact on the momentum interpretation, based on that:

$$\nabla g_j(x) = \nabla f(x) + \gamma(x - x_k)$$

and thus,

$$\nabla g_j(x_k) = \nabla f(x_k)$$

Plugging it back into the previous expression, we conclude that, although mathematically the Almost-convex-AGD is an extension of the standard Accelerated-gradient-descent, and they have different properties in theory, but when making them practical, say, implemented by momentum, we have the same expression.

---

function Almost−convex–AGD(f, γ, β, x_0, T)
  $y_0 = x_0$
  for $k = 0, 1, 2, \ldots, T-1$:
    $y_{k+1} = x_k - \gamma\nabla f(x_k)$
    $x_{k+1} = y_{k+1} + \beta(y_{k+1} - y_k)$
  return x_T

One thing to notice is that, the γ we used as step-size and the γ we used to express "almost-convex" are not the same thing. The γ shows up in pseudo code is always the step size. We're no longer talking about the "almost-convex" thing from this point on.

Another important base algorithm in [1] is the *Negative curvature descent method*, which is also essential in [2].

This algorithm involves some computation. As is directly from the paper [1], we have the algorithm as:

---

function Negative−curvature−descent($z_1, f, L_2, \alpha, \Delta_f, \delta$)
  $\delta' = \frac{\delta}{1 + \frac{12L_2^2\Delta_f}{\alpha^3}}$
  for $j = 1, 2, \ldots, \infty$:
    Find v_j such that:
      1.) $\|v_j\| = 1$
      2.) $\mathbb{P}[\lambda(\nabla^2 f(z_j)) \geq v_j^T\nabla^2 f(z_j)v_j - \frac{\alpha}{2}] \geq 1 - \delta'$
      (use approximate leading eigenvector computation algorithm)
    if $v_j^T\nabla^2 f(z_j)v_j \leq -\frac{\alpha}{2}$
      $z_{j+1} = z_j - \frac{2|v_j^T\nabla^2 f(z_j)v_j|}{L_2} sign(v_j^T\nabla f(z_j))v_j$
    else
      return z_j

Previous works have brought some insights into the usage of the negative curvature descent. In [2], it is clearly stated that, based on the division of the entire domain of the objective function into small and large gradient regions, they perform only gradient descent based procedure in the large gradient region, and only negative curvature descent in the small gradient region. Their analysis also shows that it could escape the small gradient region in only one negative curvature descent step whenever entering the region.

Following similar ways like *Algorithm 3* from [2], first of all, we simplify the parameters. This step would be useful for implementation.

Instead of letting those variables being random values, we have $\alpha \to 0$, thus $\delta' \to 0$ and δ is also omitted. Furthermore, v_j is computed by using *ApproxNC-Stochastic* algorithm.

Previously, finding a v is equivalent with finding an element from the following problem's feasible set:

$$\begin{aligned} &\text{minimize } \mathbf{0} \\ &\text{subject to } \|v\|_2 = 1 \\ &\qquad \lambda(\nabla^2 f(z_j)) \geq v_j^T\nabla^2 f(z_j)v_j \end{aligned}$$

The *ApproxNC-Stochastic* algorithm will return:

$$\hat{v} = \begin{cases} \perp & \lambda_{min}(\nabla^2 f(z_j)) \geq -\varepsilon_H \\ else & \|v\|_2 = 1, -\frac{\varepsilon_H}{2} \geq v_j^T \nabla^2 f(z_j) v_j \end{cases}$$

which will perfectly fit for our usage by setting $\varepsilon_H = \alpha$. Previously I claimed that we want to simplify the implementation by setting $\alpha \to 0$, then the same applies to ε_H. For details, this is the online Neon2 method proposed in [9].

In general, we simplify the previous expression into:

```
function Negative-curvature-descent(f, x_0, η, T_2, δ, ε_H)
    for j = 0, 1, 2, ..., T_2 - 1:
        v̂_{j+1} = ApproxNC-Stochastic(f, x_j, δ, ε_H)
        if v̂_{j+1} ≠ ⊥
            x_{j+1} = x_j + η sign(-v_j^T ∇f(z_j)) v_j
        else
            return x_j
    return x_{T_2}
```

where T_2 is the maximum number of iterations in negative curvature descent calculation.

The algorithm we have in [1] put them together, by having:

```
function Accelerated-non-convex-method(f, x_0, η, γ, β, T, T_2, δ, ε_H, ε)
    for k = 1, 2, ..., T
        x̂_k = Negative-curvature-descent(f, x_{k-1}, η, T_2, δ, ε_H)
        x_{k+1} = Almost-convex-AGD(f, γ, β, x̂_k, T)
        if ||x̂_k - x_{k+1}|| < ε
            return x_{k+1}
    return x_T
```

In fact, when it is not simplified this much, $\varepsilon = \frac{\alpha}{L_2}$.

There's another algorithm from [2], which basically applies standard optimizer (such as SGD or Adam, etc.) for certain steps, and then, check the gradient, if the gradient is small, call the negative curvature descent optimizer for gradient region, escaping the small-gradient area in only one negative curvature descent step. This method is called GOSE.

And there naturally comes an idea that, how about combining the two, GOSE idea of checking at certain interval so as to use negative curvature descent to escape small gradient areas, and the AGD together? I also tried this out, and will be discussed in following sections.

5 Implementation

My code is released on Github at `https://github.com/PatriciaXiao/AcceleratedSGDMethods`, it is a public repository. Due to the time-limitation issues, it is not yet tested against other environment, but it is guaranteed to work under macOS 10.13.3 with Python 3.7 and PyTorch 1.0.1.post2.

Using Python3.7 and the PyTorch library, I work on implementing the new optimization algorithms. [5]

The network structures are fixed to be the same as specified in CS269 by Prof. Lin. [6] That is, the model has three convolutional layers followed by a fully-connected layer. The detailed structure is as specified in the code.

Fortunately, there's an existing implementation of GOSE on Github, [7] although a little bit buggy and mostly outdated, thus some efforts were made to correct and addapt part of their code into mine, it saved me a lot of time from implementing the troublesome negative-curvature-descent.

The loss function is chosen to be the most-commonly used one in the entity-classification tasks: the cross-entropy loss.

As for learning rate, I tested 0.3, 0.03, 0.003, 0.0003 for CIFAR10, and 0.2, 0.02, 0.002, 0.0002 for MNIST.

[5] *Regarding implementation of optimization algorithm*, *How could I design my own optimizer scheduler*, *Custom Optimizer in PyTorch*

[6] `https://www.csie.ntu.edu.tw/~cjlin/courses/optdl2019/`

[7] `https://github.com/uclaml/GOSE/blob/master/cnn_gose.py`

6 Results and Discussion

Give in to the time limit (we are due soon), I run only 6 epochs for each setting, repeat 3 times on CIFAR10 and 2 times on MNIST. In practice, if given adequate time, we should run at least 200 epochs on each setting, and repeat at least 6 times to get rid of the impact of randomness.

6.1 The Models

There are five different kinds of algorithms, applying them get different outcomes. Standard SGD works well, and Accelerated Gradient Descent (with Nesterov's algorithm in momentum form) turns out to be more stable (and also faster). ANCM (the algorithm proposed in [1]) shows a very stable trend of increasing, and it is clearly not converged yet, unlike standard SGD, whose accuracy almost starts dropping. GOSE is more stable than standard SGD and generally works a little bit better than the standard version.

By combing the GOSE idea of checking upon a certain interval and utilize NCD (negative curvature descent) to escape the small gradient area, and ANCM's idea of utilizing AGD, we have a combined approach, which runs almost as fast as GOSE (which is 1.5 times slower than standard SGD), and shows better test accuracy.

Comparing the outcome of the models, I plotted several plots.

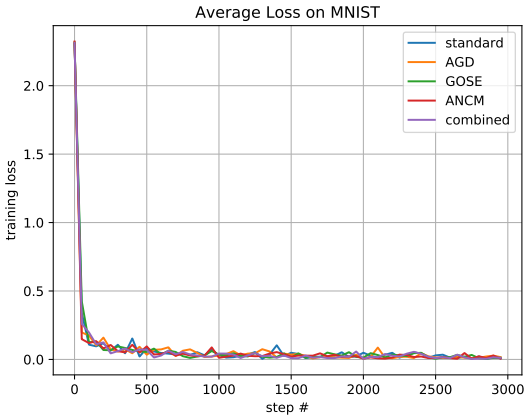

Figure 1: Loss on MNIST dataset

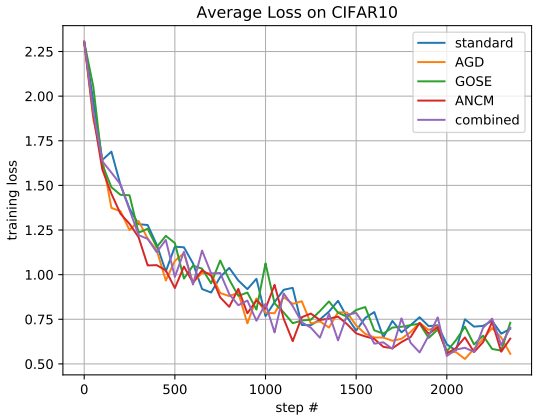

Figure 2: Loss on CIFAR10 dataset

The loss every step on the two datasets are as shown in Figure 1 and 2 respectively.

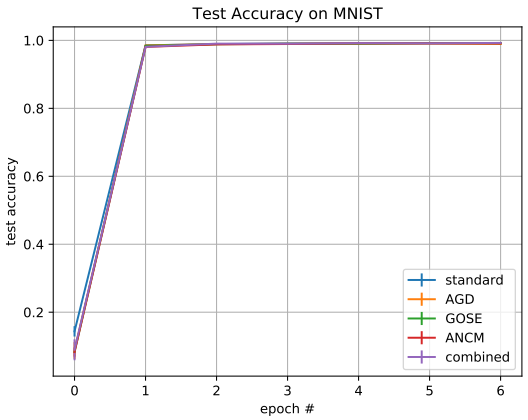

Figure 3: Accuracy on MNIST dataset

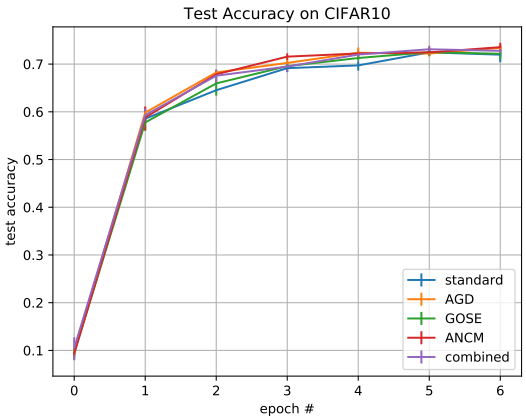

Figure 4: Accuracy on CIFAR10 dataset

The test accuracy on the two datasets are as shown in Figure 3 and 4 respectively (epoch-level).

Note that I've plotted the test accuracy using error bar. Zooming them in, we see Figure 5 and 6. Those images shows the detailed difference more precisely.

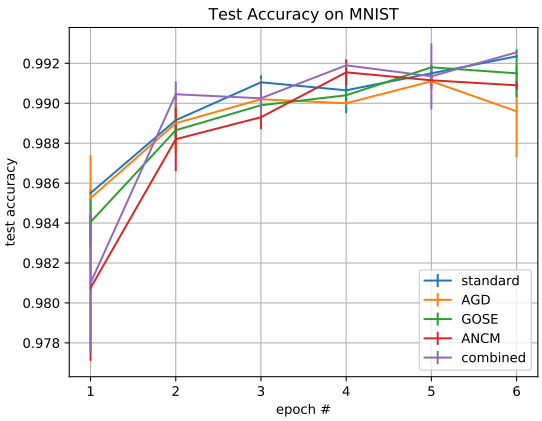

Figure 5: Test Accuracy on MNIST dataset, zoomed-in.

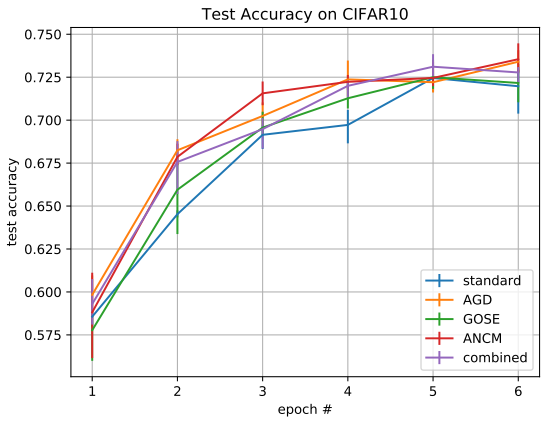

Figure 6: Test Accuracy on CIFAR10 dataset, zoomed-in.

6.2 Parameter Adaption

It is needless to say that in this case, actually the parameters need adaption, including momentum and learning rate and all thresholds etc.

The reason why I have this statement is that, as is discussed in previous papers, especially the theoretical analysis in [1], it claims that those parameters aren't randomly selected. Instead, the parameters we have came from some transformation from the unknown and almost immeasurable L_1 and L_2 values and other properties like such. In practice, I have no other choice but making them into hyper-parameters.

The model definitely needs tuning. We need to adapt the hyper-parameters so as to get the best performance and fulfill its powerfulness. However, time is limited this quarter, and other projects have taken too much computational resources these days, all those issues made it impossible for me to tune it carefully.

As I said, I've put my code on Github, if anyone is interested in these algorithms, please feel free to use it, and modify it if needed.

6.3 Running Time

The accelerated algorithms aren't necessarily running faster each epoch. Among them all, the most basic accelerated gradient descent from Nesterov still pops out. It has promising outcomes in general, and wouldn't bring heavy burden to the training time.

The barrier of the running time lies in calling Negative-curvature-descent. Whenever it is called, in GOSE, in ANCM, or in combined method, it always slows down the whole process dramatically. However, one interesting finding is that although generally speaking, the more inner iteration and the more epochs used to calculate the negative curvature descent affect the running time, their influences aren't significant. The most significant property is *how many times* we call the Negative-curvature-descent, no matter what parameters we set for this call.

Average running time for each algorithm is as listed in Table 1.

6.4 Robustness to Learning Rate

CIFAR10 fails on learning rate 0.3 whatever algorithm used, as long as it is SGD-based. It fails entirely: whenever learning rate is 0.3, the loss will be exploded and becomes **nan** within the first epoch. Same thing happens to MNIST with learning rate 0.2.

This phenomenon indicates that SGD-family is generally not robust to hyper-parameters, especially learning rate. And this issue wouldn't be solved by applying the new algorithms we have.

DATASET	ALGORITHM	TIME (s)
MNIST	standard SGD	101.8398
	accelerated AGD	100.1971
	GOSE	150.2517
	ANCM	911.5307
	combined (GOSE + ANCM)	152.0238
CIFAR10	standard SGD	102.7590
	accelerated AGD	104.7911
	GOSE	155.1117
	ANCM	943.9628
	combined (GOSE + ANCM)	160.1029

Table 1: Average Running Time per Epoch

Since the number of epochs we run is too limited, having too small a learning rate is not a good idea, since smaller learning rate (or step size) will typically converge slower. That is why I finally reported only the results with $lr = 0.02$ for MNIST and $lr = 0.03$ for CIFAR10. They are the largest learning rates that work.

6.5 Future Work

In fact, in previous experiments done in Prof. Lin's Deep-Learning Optimization course (CS269 Spring 19), we observed that the test accuracy might drop after several hundreds of epochs, using the same model as I implemented in this project's code.

Could it be the case that the SGD algorithm was trapped in local minimum and couldn't come out? If that is the case, wouldn't it be helpful if we apply the algorithms we have for now to learn the parameters? I mean, the negative curvature descent, as is stated in [2], should be helpful in escaping the local minimal areas.

It is worthwhile running, say, 500 epochs, when time permitted, and see if this assumption is correct.

Also, I observed that the bottleneck of the performance lies in the calculation of the negative curvature descent. If we manage to find a proximal solution or any ways of solving it faster, then these algorithms will all be dramatically accelerated.

Acknowledgement

First of all, thanks a lot to Prof. Vandenberghe for providing the ECE236 series in UCLA. This is one of the most fruitful courses I've ever taken in my lifetime. Also thanks a lot to Prof. Vandenberghe for introducing the interesting work of [1] to me.

Second, I thanks a lot to Xin Jiang, our TA, for helping us a lot in discussion sessions and off-school. Beside being intelligence and inspiring, he is very helpful and nice, one of the best TAs I've ever met.

Also, thanks a lot to Prof. Quanquan Gu, for his paper ([2]) really helped me understand the models, and also for that he is the one who convinced me math could be interesting instead of scary, back in 2018 Fall, during his CS260. Special thanks to Jinghui Chen, a student of Prof. Gu's, for generously releasing his code online.

Thanks a lot to my friend Yewen Wang, for taking CS260 and ECE236C with me, and sometimes helping me understand the math problems.

References

[1] Yair Carmon, John C Duchi, Oliver Hinder, and Aaron Sidford. Accelerated methods for nonconvex optimization. *SIAM Journal on Optimization*, 28(2):1751–1772, 2018.

[2] Yaodong Yu, Difan Zou, and Quanquan Gu. Saving gradient and negative curvature computations: Finding local minima more efficiently. *arXiv preprint arXiv:1712.03950*, 2017.

[3] Shai Shalev-Shwartz and Tong Zhang. Accelerated proximal stochastic dual coordinate ascent for regularized loss minimization. In *International Conference on Machine Learning*, pages 64–72, 2014.

[4] Aladin Virmaux and Kevin Scaman. Lipschitz regularity of deep neural networks: analysis and efficient estimation. In *Advances in Neural Information Processing Systems*, pages 3835–3844, 2018.

[5] Thomas Wiatowski and Helmut Bölcskei. A mathematical theory of deep convolutional neural networks for feature extraction. *IEEE Transactions on Information Theory*, 64(3):1845–1866, 2017.

[6] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147, 2013.

[7] Jason D Lee, Max Simchowitz, Michael I Jordan, and Benjamin Recht. Gradient descent only converges to minimizers. In *Conference on learning theory*, pages 1246–1257, 2016.

[8] Yurii Nesterov. *Introductory lectures on convex optimization: A basic course*, volume 87. Springer Science & Business Media, 2013.

[9] Zeyuan Allen-Zhu and Yuanzhi Li. Neon2: Finding local minima via first-order oracles. In *Advances in Neural Information Processing Systems*, pages 3716–3726, 2018.