Chapter 11

This lecture considers distributed approaches in computing unconstrained optimization on very large datasets. We will discuss how to distribute optimization in large-scale settings, study synchrony vs. asynchrony in gradietn descent, provide rough theoretical results on how asynchrony affects performance, and take a look at alternatives and state of the art.

Distributed Gradient Descent

Recall: Stochastic gradient descent. (SGD) is used almost everywhere, both in training classical ML tasks (linear prediction, linear classification) and training modern ML tasks (non-linear classification, neural networks). Mathematically, it satisfies:

$$x_{t+1} = x_t - \eta \nabla f_{i_t} x_t$$

Intuitively, we select a training sample, compute the gradient on this sample, and update the model based on this gradient. We can observe these properties:

1. The current model x_t is used for the computation of $\nabla f_{i_t}(\cdot)$.
2. When we update the model, the state of the system is as when we read x_t
3. The whole process is sequential.

SGD as presented above operates on a single machine (single CPU, single memory, single communication bus line). Let us first observe where computation and communication happens in the SGD iteration.

1. Model x_t needs to be "transferred" to the location where computation of $\nabla f_{i_t}(\cdot)$ happens
2. Data point $f_{i_t}(\cdot)$ needs to be traansferred where the computation of $\nabla f_{i_t}(\cdot)$ happens
3. The update $x_t - \eta \nabla f_{i_t}(x_t)$ overwrites the current model

Note that SGD is not easily parallelizable on GPU due to its memory constraint; the model and data do not fit in the GPU memory.

Distributed computing. We can generally split distributed computing into two categories: **Single node distributed computing** and **Multi-node distributed computing**.

Single node distributed computing

1. Single machine, many cores (up to 100s)
2. Shared memory (all processors have access to it)
3. Communication to RAM is relatively cheap

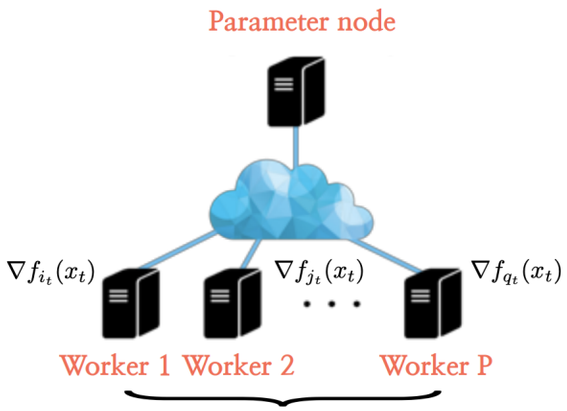
Multi-node distributed computing

1. Single machine, many cores (up to 100s)
2. Shared memory (all processors have access to it)
3. Communication to RAM is relatively cheap

It is not apparent how multi-node distributed computing could be utilized in stochastic gradient descent, but it is clear in the full gradient descent case:

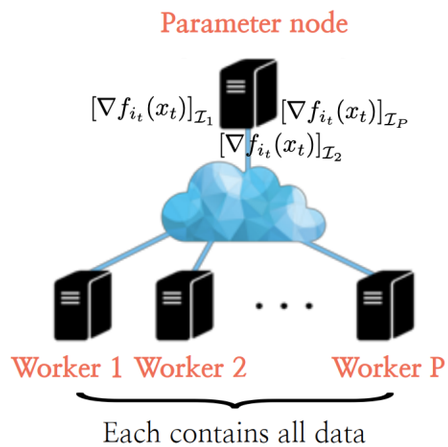
$$x_{t+1} = x_t - \eta \sum_{i=1}^{n} \nabla f_i(x_t)$$

Data-parallel distribution. We can consider a data-parallel system where a parameter node keeps and distributes model x_t to worker nodes at every iteration, then each worker node computes a part of the full gradient $(\nabla f_i(\cdot))$ based on the data they have, and finally, the parameter node waits for all gradient parts to be collected to update the model. This is illustrated in the diagram below.



Notice that synchronization is required at the end of each iteration; the parameter node has to wait for the slowest worker node to finish its computations. While gradient descent neatly distributes in this fashion, this model would not work in an online learning setting where data samples arrive one at a time, or if we don't have access to all data for any reason. Even if there is finite and fixed data, full gradient descent suffers from generalization error – it does not perform well on unseen data. This is because gradient descent converges to the "first-seen" stationary point, causing it to overfit the landscape of the training data. Meanwhile, SGD explores the landscape before converging.

Coordinate-parallel distribution. Since SGD inherently performs computations on a much smaller subset of the data compared to full GD, distributing SGD computations makes more sense when we consider a scenario where even calculating $\nabla f_{i_t}(x_t) \in \mathbb{R}^p$ is expensive for a single node. We can therefore introduce coordinate-based parallelism. Just like in the data-parallel system, a parameter node keeps and distributes model x_t to worker nodes at every iteration. In addition, the parameter node also distributes indices corresponding to coordinates to worker nodes. The worker nodes then compute part of the stochastic gradient based on the coordinates given by the parameter node. Finally, the parameter node waits for all gradient parts to be collected to update the model.



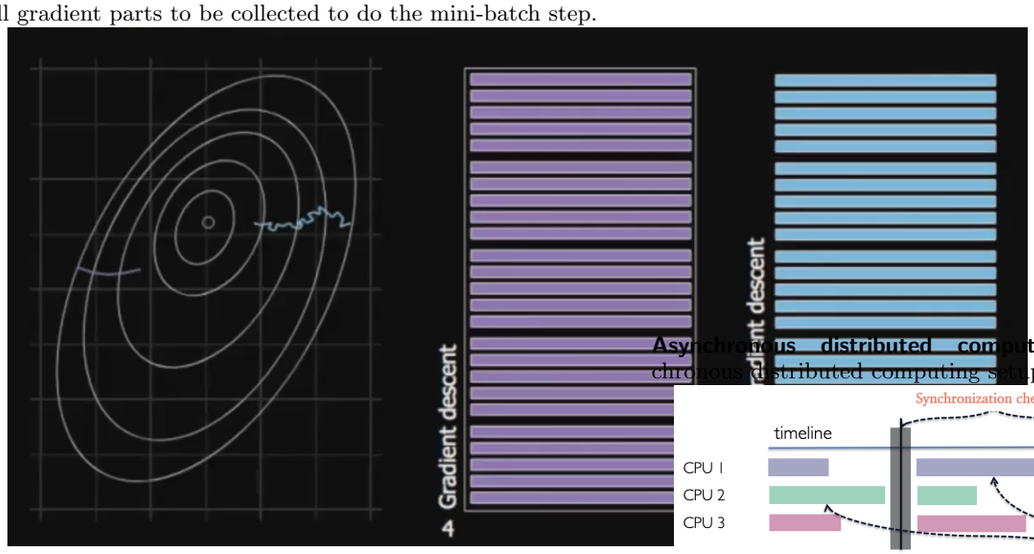
We ob-

serve that this model is related to coordinate descent algorithms. It is effective in a large-scale implementation, where just a part of the model is too large to be computed in a centralized fashion. However, in a smaller scale, it could be overkill to only compute updates for a small subset of entries.

Mini-batch SGD. The mini-batch SGD iteration can be mathematically described as

$$x_{t+1} = x_t - \eta \sum_{i \in \mathcal{I}_t} \nabla f_i(x_t)$$

This sits in between the previous two distributed computing schemes. As before, the parameter node keeps and distributes model x_t at every iteration. Worker nodes then compute their part of the mini-batch gradient. The parameter node waits for all gradient parts to be collected to do the mini-batch step.



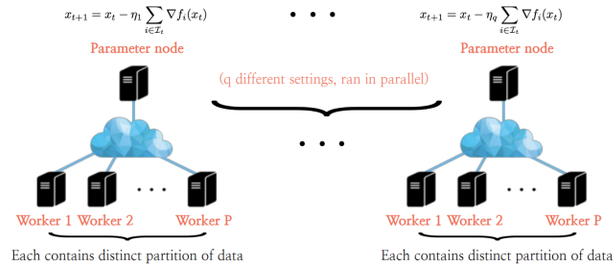
Like the previous schemes, this method still requires a synchronization step. Since each worker has less to do, and computing $\nabla f_{i_t}(x_t)$ is usually cheap per node, synchronization happens more often, thus introducing higher synchronization step overheads. This and highlights that there is a tradeoff between statistical efficiency, computations efficiency in terms of convergence, and communication efficiency.

Mini-batch SGD in parallel. Another scenario is to run mini-batch SGD in parallel and average combine the results at the end. Each worker node would independently run:

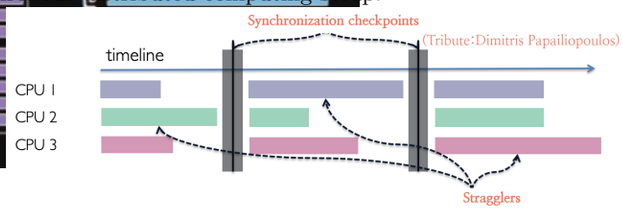
$$x_{t+1} = x_t - \eta \sum_{i \in \mathcal{I}_t} \nabla f_i(x_t)$$

The parameter node would be idle while workers nodes do mini=batch SGD as if there is no distributed computation. After collecting models from all workers, the parameter node averages the models. This facilitates minimal communication. Every node works on its own and only sends the model at the end of its execution. The final decision is prediction averaging – similar ideas hold for random forests. Designed for convex problems, the theoretical basis of this model is that each subproblem has a solution that is close to the global one, such that averaging models from independent mini-batch SGD executions does not hurt.

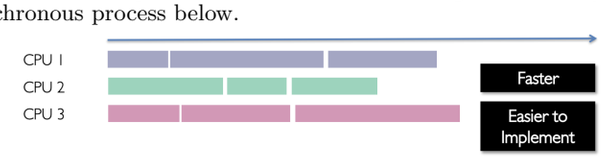
Running code in parallel for hyperparameter optimization. This is another way to use distributed computing in SGD. The code is run in parallel with different settings, without being aggregated in any way, to optimize hyperparameters.



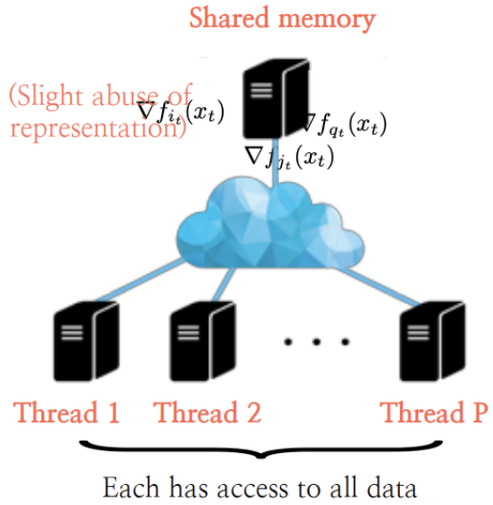
Asynchronous distributed computing. Consider the synchronous distributed computing setup.



As shown in the figure above, there are synchronization checkpoints that wait for every worker node at the end of every iteration. This means the parameter node waits for the slowest worker before synchronizing all workers, and keeps all nodes aware of each other's updates to the model. This makes synchronization very expensive. Consider a setting with P workers. If P–1 of them have already sent their updates to the parameter server, the whole system needs to wait for the one straggler to complete before proceeding. This is apparent when we compare the video above to the figure of an asynchronous process below.



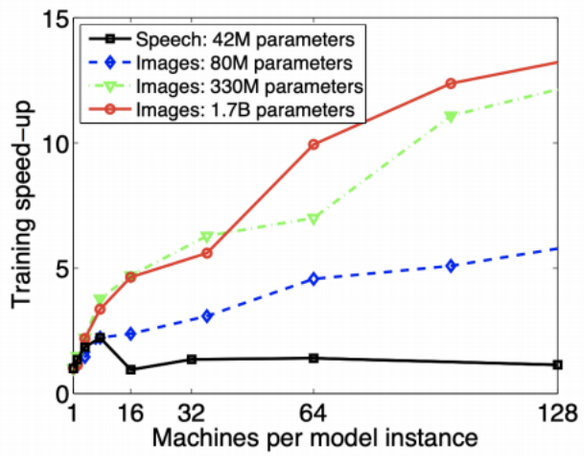
There are several ways to avoid this unnecessary overhead. Firstly, multicore systems can host lar-ge scale problems. After preprocessing, large-scale problems may involve only a few terrabytes of data. Thus, instead of running SGD across clusters of processing nodes, one can use a single inexpensive multicore work station. This model boasts advantages such as low latency and high throughput shared high memory, high bandwidth of multiple disks, and fast multithread processors. However, synchronization and locking amongst processors is still the main bottleneck in this model. This led to the idea of running SGD in parallel without locks.



In this model, a CPU core acts as the parameter server, and other CPU cores act as threads. All threads in a single machine have access to shared memory, each thread can independently ask for the current model in memory, and each thread computes a gradient and and updates the shared memory. Since the order that updates are sent is random due to the different runtimes of different worker nodes, and since this model is completely asynchronous, the controller in shared memory updates the model in a first-in-first-served fashion. Assuming that all threads have collected x_t, the model will be updated to

$$x_{t+1} = x_t - \eta(\nabla f_{i_t}(x_t) + \nabla f_{j_t}(x_t) + ... + \nabla f_{q_t}(x_t))$$

However, this is the most straightforward case. Due to asynchrony, threads might process an older model version and complete their tasks in an arbitrary order. It is also possible that threads read a model state that only saved in memory for a short time and between other memory writes. Despite this complexity and uncertainty, it has been shown in a paper by Google titled *Large Scale Distributed Deep Networks* that asynchronous SGD works very well for training deep networks, particularly when combined with Adagrad adaptive learning rates. Additionally, while asynchronous SGD is rarely applied to nonconvex problems and there is little theoretical grounding for the safety of these operations for these problems, they found that relaxing consistency requirements is remarkably effective in practice.



We can see in the graph above that asynchrony can lead to upwards of 10x speedup in some cases, but in other cases there may not be any speedup because as previously mentioned, communication bottlenecks may cause workers to read an older model state and consequently overwrite progress.

HOGWILD!: an update scheme that allows processors access to shared memory with the possibility of overwriting each other's work. Consider the following setting:

$$\min_x f(x) := \sum_{e \in E} f_e(x_e)$$

where:

E is a collection of items, say samples$x \in \mathbb{R}^n$, $e \subset [n]$ each element e is a col

To clarify, while $f_e(\cdot)$ denotes a component of a sum of functions indexed by sample e, x_e is the sub-vector indexed by an index set e. In other words, each sample $e \in E$ corresponds to an index set $e \subset [n]$. This is more tangible when we consider the following key observation:

n and $|E|$ are large, while individual $f_e(\cdot)$ act on a small number of components $x \in \mathbb{R}^n$

Example: Sparse SVM

Given data $E = \{(z_1, y_1), ..., (z_{|E|}, y_{|E|})\}$ where y_i are labels and $z_i \in \mathbb{R}^n$ are features, we solve:

$$\min_x \sum_{\alpha \in E} \max(1 - y_\alpha \cdot x^\top z_\alpha, 0) + \lambda \|x\|_2^2$$

Often, z_α is sparse, meaning $x^\top z_\alpha = x_\alpha^\top z_\alpha$, where x_α is a sub-vector of x whose nonzero components correspond to the nonzero components of z_α. Analogously, for each sample $e \in E$, only a set of components of x indexed by elements of $e \subset [n]$ would be affected by the corresponding function $f_e(\cdot)$.

Algorithm 1 HOGWILD! update for individual processors

1: **loop**
2: Sample e uniformly at random from E
3: Read current state x_e and evaluate $G_e(x)$
4: **for** $v \in e$ **do** $x_v \leftarrow x_v - \gamma b_v^T G_e(x)$ ⟵ (coordinate−wise)
5: **end loop**

where $G_e \in \mathbb{R}^n$ is a gradient with non-zeros indexed by e and scaled such that $\mathbb{E}[G_e(x_e)] = \nabla f(x)$.

Verbally, we can describe this algorithm as follows:

1. Each processor samples e uniformly at random

2. Each processor computes the gradient ∇f_e at x_e
3. Each processor applies update on each coordinate in e

Notice that even if two processes update the model asynchronously, by construction, the sparsity of the update made by each process means that the probability of overlap between the indices affected by the two processes is very small. This allows computations to run in parallel without the need for any synchronization. This is reminiscent of the coordinate-parallel distribution of SGD mentioned earlier, except without the need for synchronization. This is because of the small overlap between indices affected by any two processes, and the assumption of a small delay between the models taken in by each process.

These properties hold for the asynchronous HOGWILD! algorithm:

1. When the data access is sparse, memory overwrites could be rare
2. This indicates that asynchrony introduces barely any error in the computations (this can explain why asynchronous SGD caused massive speedup in some cases and no speedup in others).
3. The authors show both theoretically and experimentally a near-linear speedup with the number of processors used.
4. In practice, lock-free SGD exceeds even theoretical guarantees

Alternatives to asynchrony. Asynchrony is used to reduce communication overheads. There are other ways to achieve this:

1. One form of lower level optimization that can be done is, instead of representing each entry of the gradient as a float number (the size of each gradient sent over the network is $O(32 \cdot p)$ bits) as done in Standard SGD, Quantized SGD quantizes each entry of some gradient to some levels such that the size of data sent over the network is $O(l \cdot p)$ bits, where $l << 32$ bits represent the levels of quantization. Intuitively, this introduces a tradeoff in the form of error, but it has been shown that 4-bit quantization with 16 GPUs could lead to 3.5x speedup.
2. In the synchronization step of each iteration, instead of waiting for stragglers, the parameter node should proceed when most of the worker nodes have finished their execution.
3. Instead of quantizing all entries of a gradient, only keep the most important entries and zero out the rest.
4. Give more "work" to workers by increasing the batch size. This allows the update to use a larger step size, thus reducing the number of iterations, communications and synchronizations. This needs careful parameter utning to work.
5. Variants of HOGWILD! that minimize communication conflicts: some computatioon is performed to distribute examples to different cores so that examples do not "conflict"

Conclusion. Distributed computing is at the heart of developments in modern ML; there are conferences dedicated to it, and there are a variety of approaches from high-level algorithms to low-level implementations such as quantized SGD. We have looked at the different ways that distributed computing could be exploited: **hyper parameter optimization, coordinate descent, mini-batch synchronous SGD, asynchronous SGD**. The type of distributed computing configurationt to use depends on the problem and resources at hand; asynchronous SGD is optimal for sparse problems, synchronous SGD running on multicore workstations may be more effective in handling non-sparse problems, and so on. Distributed computing for SGD is still a highly researched topic.

Chapter 11

The algorithms discussed so far, such as gradient descent, momentum in gradient descent, second order methods, and quasi Newton methods, can be applied to neural network training, and several optimization techniques have been developed to train these powerful structures. These optimization techniques will be the topic of this chapter. Additionally, convergence guarantees will be revisited since neural networks are, in general, nonconvex functions.

Neural Networks, deep learning, perceptron

Perceptron. Deep Neural Networks are preceded by a simple model called perceptron [127]. This is a single layer neural network originally used for binary classification. The model setting consists in multiplying a set of signals $x = (x_1, x_2, \ldots, x_n)^T$, by the trainable parameters w_i called weights, summing them up, and applying a decision function, such as the step function, to determine the binary class of the input (Figure 56).

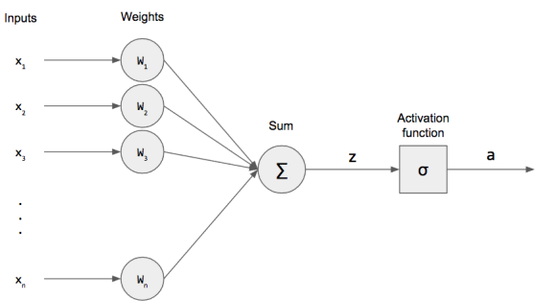


Fig. 56. Perceptron: $z = \sum_{i=1}^{n} w_i x_i, \longrightarrow a = \sigma(z)$

The algorithm to train the perceptron is fairly intuitive: start with a random initial weights w_i. Then, for each data vector $x = (x_1, \ldots, x_n)$, calculate the output.

- If the output $a = 1$ when true label is -1, then lower the weights
- If the output is $a = -1$ when the true label is $+1$, then increase values of weights.

Repeat this process until all data points are classified correctly, or after a fixed number of iterations.

Binary classification limits the applicability of the perceptron, but this can be improved in a straightforward manner with multiclass classification by using one-hot encoder for the ouput variables, that is, a class is represented by a vector with only one non-zero entry, as shown in the figure below.

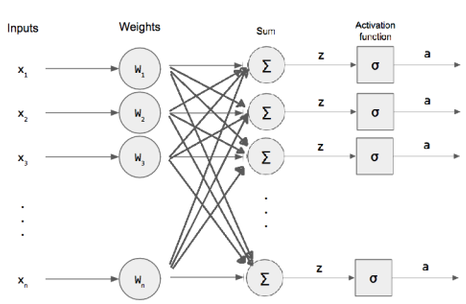


Fig. 57. Perceptron: $z = \sum_{i=1}^{n} w_i x_i, \longrightarrow a = \sigma(z)$

Issues of Perceptron. The main issue why the perceptron lost research interest is that it worked only for linearly separable data, (see Figure 58). For more complex data the perceptron failed to give accurate classification. Indeed, an easy problem that the perceptron failed to solve is the XOR logical function, implying that it would not be able to solve far more complex real-world problems.

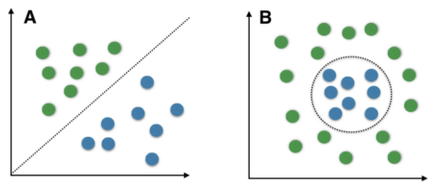
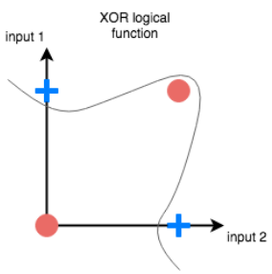


Fig. 58. Plot on the left shows linearly separable data, while on the left it is not linearly separable.

An extension of the perceptron that would solve the XOR problem is the multilayer perceptron, but there was no clear algorithm to train the weights associated with the latent variables, as it was not known what is the optimal label for the latent variables.



The multilayer perceptron, also known as feedforward neural network or fully connected neural network regained attention with the apparition of the backpropagation algorithm and the development of powerful GPUs. In the context of neural network, forward operation is the function evaluation, and the backward operation is gradient calculation .

By adding multiple layers, there are two issues that one has to take care of:

- If activation function is linear, then multilayer perceptron is equivalent to one layer perceptron:

$$W_L \ldots W_2 \cdot W_1 x = Wx$$

  Hence, the activation functions used are nonlinear, such as the sigmoid or RELU function.
- Because many multiplications are performed, vanishing grading and exploding values arise.

With this, the perceptron and the neural networks have ramped up. The **universal approximation theorem** motivates even more the use of neural network. It states that a one layer neural network wide enough can approximate any well-behaved function. Same happens for multiple layer (deep) neural networks.

Optimization algorithms. As mentioned before, neural networks resurged with the introduction of the backpropagation algorithm, that relies on the chain rule. To find the update

Table 2. Convergence guarantees for GD and SGD

	Non accelerated	Accelerated
Strongly Convex GD	$O\left(\kappa \log \frac{f(x_0)-f*}{\epsilon}\right)$	$O\left(\sqrt{\kappa} \log \frac{f(x_0)-f*}{\epsilon}\right)$
Nonconvex GD	$O\left(\frac{1}{\epsilon^2}\right)$	$O\left(\frac{1}{\epsilon^{7/4}} \log(1/\epsilon)\right)$
Strongly Convex SGD	$O\left(\frac{1}{\epsilon}\right)$	Open question
Nonconvex SGD	$O\left(\frac{1}{\epsilon^2}\right)$	Open question

equations for each weight, one had to compute all of the gradients by hand, but now it is easily done with automatic differentiation implemented with Pytorch or Tensorflow.

There are many options of optimization algorithms and activation functions to use for training a neural network, each having their own advantage and performance results. The choice of which to use depends on the researcher. In this chapter we will cover Accelerated Stochastic Gradient Descent, AdaGrad, RMSProp, and Adam. An extensive list can be found in [128]. Also, a nice blog discussing some algorithms is `https://ruder.io/optimizing-gradient-descent/`. (**Note:** this blog post has 4706 citations! And skyrocketed the carreer of the author, Sebastian Ruder. So writing blog posts is highly recommended! It can be with the purpose of educating people of your own work or the work of others.)

Accelerated Stochastic Gradient Descent. Stochastic gradient descent was already explained in detail for convex functions, see chapter 7. The algorithm remains the same for neural networks. The accelerated SGD refers to SGD with Momentum, and the update formula is:

$$x_{t+1} = x_t - \eta \nabla f(w_t) + \beta(w_t - w_{t-1})$$

Some key points for SGD are

- Stochastic gradient descent is gradient descent but using only part of the data available to get a noisy estimate of the gradient, while avoiding redundant computations for large datasets.
- The noisy gradient estimation introduces fluctuations, which helps better explore the function and jumping to potentially better local minima.

Since neural networks are non convex functions, the theoretical guarantees proved in chapter 7 differ, and can be summarized in Table 49.

AdaGrad. Stochastic Gradient Descent assumes a common (and often fixed) step size for all the weights in the neural network. To the contrary, AdaGrad [129] adapts the step size for each of the components:

- Associates small step sizes to frequently occuring features
- Associates large step sizes to rarely occuring features, in order to exploit them when they occur.

AdaGrad is a preconditioning algorithm:

$$x_{t+1} = x_t - \eta B_t^{-1} \nabla f(x_t)$$

where B_t is defined as the square root of the sum of gradient outer products, until current iteration:

$$B_t = \left(\sum_{j=1}^{t} \nabla f_{i_j}(x_j) \cdot \nabla (f_{i_j}(x_j)^T \right)^{1/2} \quad [1]$$

The density of the gradient vector implies that B_t is a dense matrix. Therefore, the calculation of the inverse of B_t at each iteration is computationally expensive to do, specially for large neural networks. To bypass this problem, the authors approximate the B_t matrix with $\text{diag}(B_t)$ by keeping the elements on the diagonal and setting the rest of the elements to zero. The inverse of $\text{diag}(B_t)$ is much easier to compute; it equals to inverting the entries of $\text{diag}(B_t)$, and the AdaGrad update formula is hence

$$x_{t+1,i} = x_{t,i} - \frac{\eta}{\sqrt{B_{t,ii} + \epsilon}} \cdot \nabla f_{i_t}(x_t)_i \quad [2]$$

The ϵ in the denominator is used to avoid division by 0.

Why is the matrix B_t defined like that? The motivation comes from the fact that the Hessian of f can be approximated by the Fisher Information matrix, which in turn is approximated by the sum of the outer product of the gradients. For a more detailed discussion, see the Natural Gradient section in chapter 5.

AdaGrad gained popularity with its successful performance in [130]. Its main advantage is that the step size is adapted, while SGD needs scrupulous hyperparameter tuning. Nevertheless, if hyperparameter tuning is done properly, it is possible to attain the same or better performance than AdaGrad (See figure 59).

RMSprop. Let's analyze the update rule 2, in particular the evolution of B_t. Note that $\lim_{t\to\inf} B_t = \infty$ because B_t is the sum of squared, positive numbers. Therefore the step size gets lower after each iteration, and it is indistinguishable after a large enough number of iterations. This is the drawback that RMSprop solves, by changing B_t to a weighted moving average, as will be explained in the following.

To do that, we can rewrite equation 1 as $E[g^2]_{t+1} = E[g^2]_t + g_t^2$, with $E[g^2]_t = B_t$, and $g_t^2 = \nabla f_{i_t}(x_t) \cdot \nabla f_{i_t}(x_t)$. Then, in the RMSprop algorithm, the definition of B_t is changed to a weighted sum:

$$E[g^2]_{t+1} = \beta E[g^2]_t + (1-\beta) g_t^2$$

usually with $\beta = 0.9$.

Adam. Adam [131] is an extension of RMSprop, with a momentum term added.

The algorithm uses the following update formula:

$$x_{t+1} = x_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \cdot \hat{m}_t$$

with

$$E[g^2]_t = \beta_2 \cdot E[g^2]_{t-1} + (1-\beta_2) \cdot g_t^2 \quad \textit{as in RMSprop}$$

and a type of momentum term, which is a summation of gradients in the past iterations:

$$m_t = \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot \nabla f_{i_t}(x_t)$$

Debiasing the terms,

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{E[g^2]_t}{1-\beta_2^t}$$

Here, the values that are commonly used are $\beta_1 = 0.9$, $\beta_2 = 0.999$.

No free lunch. There is no best algorithm for all possible applications. Some perform better in. It is data dependent, and one has to choose, or train with several algorithms to see which one works best for each particular case.

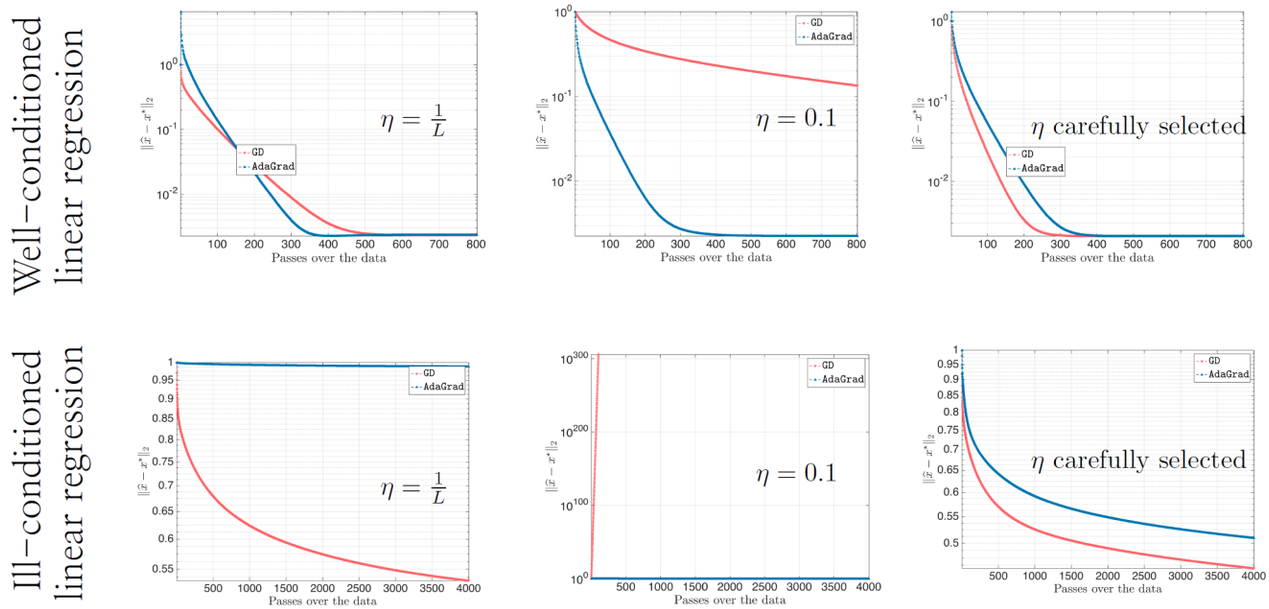


Fig. 59. Comparison of AdaGrad and SGD performances

Appendix

1. J. Nocedal and S. Wright. Numerical optimization. Springer Science & Business Media, 2006.
2. Y. Nesterov. Introductory lectures on convex optimization: A basic course, volume 87. Springer Science & Business Media, 2013.
3. S. Boyd and L. Vandenberghe. Convex optimization. Cambridge university press, 2004.
4. D. Bertsekas. Convex optimization algorithms. Athena Scientific Belmont, 2015.
5. Sébastien Bubeck. Convex optimization: Algorithms and complexity. Foundations and Trends® in Machine Learning, 8(3-4):231–357, 2015.
6. S. Weisberg. Applied linear regression, volume 528. John Wiley & Sons, 2005.
7. T. Hastie, R. Tibshirani, and M. Wainwright. Statistical learning with sparsity: the lasso and generalizations. CRC press, 2015.
8. J. Friedman, T. Hastie, and R. Tibshirani. The elements of statistical learning, volume 1. Springer series in statistics New York, 2001.
9. M. Paris and J. Rehacek. Quantum state estimation, volume 649. Springer Science & Business Media, 2004.
10. M. Daskin. A maximum expected covering location model: formulation, properties and heuristic solution. Transportation science, 17(1):48–70, 1983.
11. I. Goodfellow, Y. Bengio, and A. Courville. Deep learning. MIT press, 2016.
12. L. Trefethen and D. Bau III. Numerical linear algebra, volume 50. Siam, 1997.
13. G. Strang. Introduction to linear algebra, volume 3. Wellesley-Cambridge Press Wellesley, MA, 1993.
14. G. Golub. Cmatrix computations. The Johns Hopkins, 1996.
15. A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet classification with deep convolutional neural networks. In Advances in neural information processing systems, pages 1097–1105, 2012.
16. K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 770–778, 2016.
17. S. Ren, K. He, R. Girshick, and J. Sun. Faster R-CNN: Towards real-time object detection with region proposal networks. In Advances in neural information processing systems, pages 91–99, 2015.
18. T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In Advances in neural information processing systems, pages 3111–3119, 2013.
19. Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. arXiv preprint arXiv:1409.0473, 2014.
20. Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N Dauphin. Convolutional sequence to sequence learning. In Proceedings of the 34th International Conference on Machine Learning-Volume 70, pages 1243–1252. JMLR. org, 2017.
21. Haşim Sak, Andrew Senior, and Françoise Beaufays. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In Fifteenth annual conference of the international speech communication association, 2014.
22. Tom Sercu, Christian Puhrsch, Brian Kingsbury, and Yann LeCun. Very deep multilingual convolutional neural networks for LVCSR. In 2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pages 4955–4959. IEEE, 2016.
23. Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. page arXiv:1706.03762, 2017.
24. Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. page arXiv:1810.04805, 2018.
25. Luowei Zhou, Hamid Palangi, Lei Zhang, Houdong Hu, Jason J Corso, and Jianfeng Gao. Unified vision-language pre-training for image captioning and VQA. In AAAI, pages 13041–13049, 2020.
26. Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. arXiv preprint arXiv:2005.14165, 2020.
27. Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using gpu model parallelism. arXiv preprint arXiv:1909.08053, 2019.
28. Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. arXiv preprint arXiv:1910.10683, 2019.
29. Gary Marcus, Ernest Davis, and Scott Aaronson. A very preliminary analysis of DALL-E 2. arXiv preprint arXiv:2204.13807, 2022.
30. John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, et al. Highly accurate protein structure prediction with AlphaFold. Nature, 596(7873):583–589, 2021.
31. Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.
32. Or Sharir, Barak Peleg, and Yoav Shoham. The cost of training nlp models: A concise overview. arXiv preprint arXiv:2004.08900, 2020.
33. H. Karimi, J. Nutini, and M. Schmidt. Linear convergence of gradient and proximal-gradient methods under the Polyak-Łojasiewicz condition. In Joint European Conference on Machine Learning and Knowledge Discovery in Databases, pages 795–811. Springer, 2016.
34. Philip Wolfe. Convergence conditions for ascent methods. SIAM review, 11(2):226–235, 1969.
35. Larry Armijo. Minimization of functions having Lipschitz continuous first partial derivatives. Pacific Journal of mathematics, 16(1):1–3, 1966.
36. Stephen Wright and Jorge Nocedal. Numerical optimization. Springer Science, 35(67-68):7, 1999.
37. B. Polyak. Introduction to optimization. Inc., Publications Division, New York, 1, 1987.
38. Stephen Boyd, Lin Xiao, and Almir Mutapcic. Subgradient methods. lecture notes of EE392o, Stanford University, Autumn Quarter, 2004:2004–2005, 2003.
39. Marguerite Frank, Philip Wolfe, et al. An algorithm for quadratic programming. Naval research logistics quarterly, 3(1-2):95–110, 1956.
40. M. Jaggi. Revisiting Frank-Wolfe: Projection-free sparse convex optimization. In Proceedings of the 30th international conference on machine learning, number CONF, pages 427–435, 2013.
41. J. Duchi, S. Shalev-Shwartz, Y. Singer, and T. Chandra. Efficient projections onto the ℓ_1-ball for learning in high dimensions. In Proceedings of the 25th international conference on Machine learning, pages 272–279, 2008.
42. Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. Computer, (8):30–37, 2009.
43. A. Mnih and R. Salakhutdinov. Probabilistic matrix factorization. In Advances in neural information processing systems, pages 1257–1264, 2008.
44. T. Booth and J. Gubernatis. Improved criticality convergence via a modified Monte Carlo power iteration method. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.
45. S. Zavriev and F. Kostyuk. Heavy-ball method in nonconvex optimization problems. Computational Mathematics and Modeling, 4(4):336–341, 1993.
46. E. Ghadimi, H. Feyzmahdavian, and M. Johansson. Global convergence of the heavy-ball method for convex optimization. In 2015 European control conference (ECC), pages 310–315. IEEE, 2015.
47. Y. Nesterov. A method of solving a convex programming problem with convergence rate $O(\frac{1}{k^2})$. In Soviet Mathematics Doklady, volume 27, pages 372–376, 1983.
48. B. O'Donoghue and E. Candes. Adaptive restart for accelerated gradient schemes. Foundations of computational mathematics, 15(3):715–732, 2015.
49. O. Devolder, F. Glineur, and Y. Nesterov. First-order methods of smooth convex optimization with inexact oracle. Mathematical Programming, 146(1-2):37–75, 2014.
50. L. Bottou, F. Curtis, and J. Nocedal. Optimization methods for large-scale machine learning. Siam Review, 60(2):223–311, 2018.
51. S. Chen, D. Donoho, and M. Saunders. Atomic decomposition by basis pursuit. SIAM review, 43(1):129–159, 2001.
52. R. Tibshirani. Regression shrinkage and selection via the lasso. Journal of the Royal Statistical Society: Series B (Methodological), 58(1):267–288, 1996.
53. P. Hoff. Lasso, fractional norm and structured sparse estimation using a Hadamard product parametrization. Computational Statistics & Data Analysis, 115:186–198, 2017.
54. S. Becker, J. Bobin, and E. Candès. NESTA: A fast and accurate first-order method for sparse recovery. SIAM Journal on Imaging Sciences, 4(1):1–39, 2011.
55. T. Blumensath and M. Davies. Iterative hard thresholding for compressed sensing. Applied and computational harmonic analysis, 27(3):265–274, 2009.
56. D. Needell and J. Tropp. CoSaMP: Iterative signal recovery from incomplete and inaccurate samples. Applied and computational harmonic analysis, 26(3):301–321, 2009.
57. S. Foucart. Hard thresholding pursuit: an algorithm for compressive sensing. SIAM Journal on Numerical Analysis, 49(6):2543–2563, 2011.
58. J. Tanner and K. Wei. Normalized iterative hard thresholding for matrix completion. SIAM Journal on Scientific Computing, 35(5):S104–S125, 2013.
59. K. Wei. Fast iterative hard thresholding for compressed sensing. IEEE Signal processing letters, 22(5):593–597, 2014.
60. Rajiv Khanna and Anastasios Kyrillidis. Iht dies hard: Provable accelerated iterative hard thresholding. In International Conference on Artificial Intelligence and Statistics, pages 188–198. PMLR, 2018.
61. Jeffrey D Blanchard and Jared Tanner. GPU accelerated greedy algorithms for compressed sensing. Mathematical Programming Computation, 5(3):267–304, 2013.
62. A. Kyrillidis, G. Puy, and V. Cevher. Hard thresholding with norm constraints. In 2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pages 3645–3648. Ieee, 2012.
63. A. Kyrillidis and V. Cevher. Recipes on hard thresholding methods. In Computational Advances in Multi-Sensor Adaptive Processing (CAMSAP), 2011 4th IEEE International Workshop on, pages 353–356. IEEE, 2011.
64. X. Zhang, Y. Yu, L. Wang, and Q. Gu. Learning one-hidden-layer ReLU networks via gradient descent. In The 22nd International Conference on Artificial Intelligence and Statistics, pages 1524–1534, 2019.
65. Emmanuel J Candès, Justin Romberg, and Terence Tao. Robust uncertainty principles: Exact signal reconstruction from highly incomplete frequency information. IEEE Transactions on information theory, 52(2):489–509, 2006.
66. Joseph B Altepeter, Daniel FV James, and Paul G Kwiat. 4 qubit quantum state tomography. In Quantum state estimation, pages 113–145. Springer, 2004.

67. Jens Eisert, Dominik Hangleiter, Nathan Walk, Ingo Roth, Damian Markham, Rhea Parekh, Ulysse Chabaud, and Elham Kashefi. Quantum certification and benchmarking. arXiv preprint arXiv:1910.06343, 2019.
68. Masoud Mohseni, AT Rezakhani, and DA Lidar. Quantum-process tomography: Resource analysis of different strategies. Physical Review A, 77(3):032322, 2008.
69. D. Gross, Y.-K. Liu, S. Flammia, S. Becker, and J. Eisert. Quantum state tomography via compressed sensing. Physical review letters, 105(15):150401, 2010.
70. Y.-K. Liu. Universal low-rank matrix recovery from Pauli measurements. In Advances in Neural Information Processing Systems, pages 1638–1646, 2011.
71. K Vogel and H Risken. Determination of quasiprobability distributions in terms of probability distributions for the rotated quadrature phase. Physical Review A, 40(5):2847, 1989.
72. Miroslav Ježek, Jaromír Fiurášek, and Zdeněk Hradil. Quantum inference of states and processes. Physical Review A, 68(1):012305, 2003.
73. Konrad Banaszek, Marcus Cramer, and David Gross. Focus on quantum tomography. New Journal of Physics, 15(12):125020, 2013.
74. A. Kalev, R. Kosut, and I. Deutsch. Quantum tomography protocols with positivity are compressed sensing protocols. Nature partner journals (npj) Quantum Information, 1:15018, 2015.
75. Giacomo Torlai, Guglielmo Mazzola, Juan Carrasquilla, Matthias Troyer, Roger Melko, and Giuseppe Carleo. Neural-network quantum state tomography. Nat. Phys., 14:447–450, May 2018.
76. Matthew JS Beach, Isaac De Vlugt, Anna Golubeva, Patrick Huembeli, Bohdan Kulchytskyy, Xiuzhe Luo, Roger G Melko, Ejaaz Merali, and Giacomo Torlai. Qucumber: wavefunction reconstruction with neural networks. SciPost Physics, 7(1):009, 2019.
77. Giacomo Torlai and Roger Melko. Machine-learning quantum states in the NISQ era. Annual Review of Condensed Matter Physics, 11, 2019.
78. M. Cramer, M. B. Plenio, S. T. Flammia, R. Somma, D. Gross, S. D. Bartlett, O. Landon-Cardinal, D. Poulin, and Y.-K. Liu. Efficient quantum state tomography. Nat. Comm., 1:149, 2010.
79. BP Lanyon, C Maier, Milan Holzäpfel, Tillmann Baumgratz, C Hempel, P Jurcevic, Ish Dhand, AS Buyskikh, AJ Daley, Marcus Cramer, et al. Efficient tomography of a quantum many-body system. Nature Physics, 13(12):1158–1162, 2017.
80. D. Gonçalves, M. Gomes-Ruggiero, and C. Lavor. A projected gradient method for optimization over density matrices. Optimization Methods and Software, 31(2):328–341, 2016.
81. E. Bolduc, G. Knee, E. Gauger, and J. Leach. Projected gradient descent algorithms for quantum state tomography. npj Quantum Information, 3(1):44, 2017.
82. Jiangwei Shang, Zhengyun Zhang, and Hui Khoon Ng. Superfast maximum-likelihood reconstruction for quantum tomography. Phys. Rev. A, 95:062336, Jun 2017.
83. Zhilin Hu, Kezhi Li, Shuang Cong, and Yaru Tang. Reconstructing pure 14-qubit quantum states in three hours using compressive sensing. IFAC-PapersOnLine, 52(11):188 – 193, 2019. 5th IFAC Conference on Intelligent Control and Automation Sciences ICONS 2019.
84. Zhibo Hou, Han-Sen Zhong, Ye Tian, Daoyi Dong, Bo Qi, Li Li, Yuanlong Wang, Franco Nori, Guo-Yong Xiang, Chuan-Feng Li, et al. Full reconstruction of a 14-qubit state within four hours. New Journal of Physics, 18(8):083036, 2016.
85. C. Riofrío, D. Gross, S.T. Flammia, T. Monz, D. Nigg, R. Blatt, and J. Eisert. Experimental quantum compressed sensing for a seven-qubit system. Nature Communications, 8, 2017.
86. Martin Kliesch, Richard Kueng, Jens Eisert, and David Gross. Guaranteed recovery of quantum processes from few measurements. Quantum, 3:171, 2019.
87. S. Flammia, D. Gross, Y.-K. Liu, and J. Eisert. Quantum tomography via compressed sensing: Error bounds, sample complexity and efficient estimators. New Journal of Physics, 14(9):095022, 2012.
88. A. Kyrillidis, A. Kalev, D. Park, S. Bhojanapalli, C. Caramanis, and S. Sanghavi. Provable quantum state tomography via non-convex methods. npj Quantum Information, 4(36), 2018.
89. B. Recht, M. Fazel, and P. Parrilo. Guaranteed minimum-rank solutions of linear matrix equations via nuclear norm minimization. SIAM review, 52(3):471–501, 2010.
90. N. Srebro, J. Rennie, and T. Jaakkola. Maximum-margin matrix factorization. In Advances in neural information processing systems, pages 1329–1336, 2004.
91. J. Rennie and N. Srebro. Fast maximum margin matrix factorization for collaborative prediction. In Proceedings of the 22nd international conference on Machine learning, pages 713–719. ACM, 2005.
92. D. DeCoste. Collaborative prediction using ensembles of maximum margin matrix factorizations. In Proceedings of the 23rd international conference on Machine learning, pages 249–256. ACM, 2006.
93. J. Bennett and S. Lanning. The Netflix prize. In Proceedings of KDD cup and workshop, volume 2007, page 35, 2007.
94. M. Jaggi and M. Sulovsk. A simple algorithm for nuclear norm regularized problems. In Proceedings of the 27th International Conference on Machine Learning (ICML-10), pages 471–478, 2010.
95. R. Keshavan. Efficient algorithms for collaborative filtering. PhD thesis, Stanford University, 2012.
96. R. Agrawal, A. Gupta, Y. Prabhu, and M. Varma. Multi-label learning with millions of labels: Recommending advertiser bid phrases for web pages. In Proceedings of the 22nd international conference on World Wide Web, pages 13–24. International World Wide Web Conferences Steering Committee, 2013.
97. K. Bhatia, H. Jain, P. Kar, M. Varma, and P. Jain. Sparse local embeddings for extreme multi-label classification. In Advances in Neural Information Processing Systems, pages 730–738, 2015.
98. G. Carneiro, A. Chan, P. Moreno, and N. Vasconcelos. Supervised learning of semantic classes for image annotation and retrieval. Pattern Analysis and Machine Intelligence, IEEE Transactions on, 29(3):394–410, 2007.
99. A. Makadia, V. Pavlovic, and S. Kumar. A new baseline for image annotation. In Computer Vision–ECCV 2008, pages 316–329. Springer, 2008.
100. C. Wang, S. Yan, L. Zhang, and H.-J. Zhang. Multi-label sparse coding for automatic image annotation. In Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on, pages 1643–1650. IEEE, 2009.
101. J. Weston, S. Bengio, and N. Usunier. WSABIE: Scaling up to large vocabulary image annotation. In IJCAI, volume 11, pages 2764–2770, 2011.
102. Andrew I. Schein, Lawrence K. Saul, and Lyle H. Ungar. A generalized linear model for principal component analysis of binary data. In AISTATS, 2003.
103. K.-Y. Chiang, C.-J. Hsieh, N. Natarajan, I. Dhillon, and A. Tewari. Prediction and clustering in signed networks: A local to global perspective. The Journal of Machine Learning Research, 15(1):1177–1213, 2014.
104. C. Johnson. Logistic matrix factorization for implicit feedback data. Advances in Neural Information Processing Systems, 27, 2014.
105. Koen Verstrepen. Collaborative Filtering with Binary, Positive-only Data. PhD thesis, University of Antwerpen, 2015.
106. N. Gupta and S. Singh. Collectively embedding multi-relational data for predicting user preferences. arXiv preprint arXiv:1504.06165, 2015.
107. Y. Liu, M. Wu, C. Miao, P. Zhao, and X.-L. Li. Neighborhood regularized logistic matrix factorization for drug-target interaction prediction. PLoS Computational Biology, 12(2):e1004760, 2016.
108. S. Aaronson. The learnability of quantum states. In Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences, volume 463, pages 3089–3114. The Royal Society, 2007.
109. E. Candes, Y. Eldar, T. Strohmer, and V. Voroninski. Phase retrieval via matrix completion. SIAM Review, 57(2):225–251, 2015.
110. I. Waldspurger, A. d'Aspremont, and S. Mallat. Phase recovery, MaxCut and complex semidefinite programming. Mathematical Programming, 149(1-2):47–81, 2015.
111. P. Biswas, T.-C. Liang, K.-C. Toh, Y. Ye, and T.-C. Wang. Semidefinite programming approaches for sensor network localization with noisy distance measurements. IEEE transactions on automation science and engineering, 3(4):360, 2006.
112. K. Weinberger, F. Sha, Q. Zhu, and L. Saul. Graph Laplacian regularization for large-scale semidefinite programming. In Advances in Neural Information Processing Systems, pages 1489–1496, 2007.
113. F. Lu, S. Keles, S. Wright, and G. Wahba. Framework for kernel regularization with application to protein clustering. Proceedings of the National Academy of Sciences of the United States of America, 102(35):12332–12337, 2005.
114. H. Andrews and C. Patterson III. Singular value decomposition (SVD) image coding. Communications, IEEE Transactions on, 24(4):425–432, 1976.
115. M. Fazel, H. Hindi, and S. Boyd. Rank minimization and applications in system theory. In American Control Conference, 2004. Proceedings of the 2004, volume 4, pages 3273–3278. IEEE, 2004.
116. E. Candès and B. Recht. Exact matrix completion via convex optimization. Foundations of Computational mathematics, 9(6):717–772, 2009.
117. P. Jain, R. Meka, and I. Dhillon. Guaranteed rank minimization via singular value projection. In Advances in Neural Information Processing Systems, pages 937–945, 2010.
118. S. Becker, V. Cevher, and A. Kyrillidis. Randomized low-memory singular value projection. In 10th International Conference on Sampling Theory and Applications (Sampta), 2013.
119. L. Balzano, R. Nowak, and B. Recht. Online identification and tracking of subspaces from highly incomplete information. In Communication, Control, and Computing (Allerton), 2010 48th Annual Allerton Conference on, pages 704–711. IEEE, 2010.
120. K. Lee and Y. Bresler. ADMiRA: Atomic decomposition for minimum rank approximation. Information Theory, IEEE Transactions on, 56(9):4402–4416, 2010.
121. A. Kyrillidis and V. Cevher. Matrix recipes for hard thresholding methods. Journal of mathematical imaging and vision, 48(2):235–265, 2014.
122. Z. Lin, M. Chen, and Y. Ma. The augmented Lagrange multiplier method for exact recovery of corrupted low-rank matrices. arXiv preprint arXiv:1009.5055, 2010.
123. S. Becker, E. Candès, and M. Grant. Templates for convex cone problems with applications to sparse signal recovery. Mathematical Programming Computation, 3(3):165–218, 2011.
124. J. Cai, E. Candès, and Z. Shen. A singular value thresholding algorithm for matrix completion. SIAM Journal on Optimization, 20(4):1956–1982, 2010.
125. Y. Chen, S. Bhojanapalli, S. Sanghavi, and R. Ward. Coherent matrix completion. In Proceedings of The 31st International Conference on Machine Learning, pages 674–682, 2014.
126. A. Yurtsever, Q. Tran-Dinh, and V. Cevher. A universal primal-dual convex optimization framework. In Advances in Neural Information Processing Systems 28, pages 3132–3140. 2015.
127. F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. Psychological Review, 65(6):386–408, 1958.
128. Robin M. Schmidt, Frank Schneider, and Philipp Hennig. Descending through a crowded valley - benchmarking deep learning optimizers. CoRR, abs/2007.01547, 2020.
129. John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. J. Mach. Learn. Res., 12(null):2121–2159, jul 2011.

130. **Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc' aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc Le, and Andrew Ng. Large scale distributed deep networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, Advances in Neural Information Processing Systems, volume 25. Curran Associates, Inc., 2012.**

131. **Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings, 2015.**