(20/1) Alex Smola's answer to What would be your advice to a software engineer who wants to learn machine learning?

This depends a lot on the background of the software engineer. And it depends on which part of machine learning you want to master. So, for the sake of concreteness, let's assume that we're talking about a junior engineer who has 4 years of university and a year of two in industry. And let's assume that this is someone who wants to work on computational advertising, natural language processing, image analysis, social networks, search and ranking. Let's start with the requirements for doing machine learning (disclaimer to my academic colleagues - this list is very incomplete, apologies in advance if your papers aren't included).

- **Linear algebra**
  A lot of machine learning, statistics and optimization needs this. And this is incidentally why GPUs are so much better than CPUs for doing deep learning. You need to have at least a basic proficiency in the following
  - **Scalars, vectors, matrices, tensors.** Think of them as zero, one, two, three and higher-dimensional objects that you can compose and use to transform another. A bit like Lego. They provide the basic data transformations.
  - **Eigenvectors, norms, matrix approximations, decompositions.** This is essentially all about getting comfortable with the things linear algebra objects do. If you want to analyze how a matrix works (e.g. to check why your gradients are vanishing in a recurrent neural network or why your controller is diverging in a reinforcement learning algorithm) you need to be able to understand by how much things can grow or shrink when applying matrices and vectors to it. Matrix approximations such as low rank or then Cholesky factorization help a lot when trying to get good performance and stability out of the code.
  - **Numerical Linear Algebra**
    This is relevant if you want to do something that's fairly optimization heavy. Think kernel methods and deep learning. Not quite so important for graphical models and samplers.
  - **Books**

    Serge Lang, Linear Algebra[1]
    A basic linear algebra book and it's well written for undergrads.

    Bela Bolobas, Linear Analysis[2]
    This is much more difficult and more relevant for anyone who wants to do a lot of math and functional analysis. Probably a good idea if you want to aim for a PhD.

    Lloyd Trefethen and David Bau, Numerical Linear Algebra[3]

    This is one of many books that you can use for this. Numerical Recipes[4] is another one but the algorithms in there are a bit dated. And then there's the Golub and van Loan book (Matrix Computations[5]).

- **Optimization (and basic calculus)**
  In many cases setting up the question to ask is rather easy, but getting the answer is not. For instance, if you want to perform linear regression (i.e. fit a line) to some data, you probably want to minimize the sum of the squared distances to the observations. Likewise, if you want to get a good model for click prediction, you want to maximize the accuracy of your probability estimates that someone will click on the ad. This means that we have the general problem of some objective, some parameters, lots of data, and we need a way to get there. This matters, in particular since we usually don't have a closed form solution.
  - **Convex Optimization**
    In many cases optimization problems are nice insofar as they don't have many local solutions. This happens whenever the problem is convex.
    (A set is convex if you can draw a line between any two points in the set and the entire line is in the set. A function is convex if you can draw a line between any two points on the graph and the line is above the graph.)

    Probably the canonical book in the field is the one by Steven Boyd and Lieven Vandenberghe[6]. It's free and awesome. Also, there are lots of great slide sets from Boyd's classes[7]. Dimitri Bertsekas[8] has generated a treasure trove of books on optimiation, control, etc. This should suffice to get anyone started in this area.

- **Stochastic Gradient Descent**
  Much of this started as a special case of convex optimization (at least the early theorems did) but it's taken off quite a bit, not the least due to the increase in data. Here's why - imagine that you have some data to go through and that your algorithm needs to look at all the data before it takes an update step. Now, if I maliciously give you 10 copies of the same data you'll have to do 10 times the work without any real benefit from this. Obviously reality isn't quite that bad but it helps if you take many small update steps, one after each instance is observed. This has been quite transformational in machine learning. Plus many of the associated algorithms are much easier.
  The challenge, however has been to parallelize this. Probably one of the first steps in this direction was our Slow Learners are Fast [9]paper from 2009. A rather pretty recent version of this are the lock free variants, such as the Hogwild[10] paper by Niu et al. in 2013. In a nutshell, these algorithms work by computing local gradients on worker machines and updating a consensus parameter set asynchronously.
  The other challenge is how to deal with ways of controlling overfitting e.g. by regularization. For convex penalties there are what is called proximal gradient algorithms. One of the more popular choices are the rahter unfortunately named FISTA algorithm[11] of Amir Beck and Marc Teboulle. For some code look at Francis Bach's SPAM toolbox[12].
- **Nonconvex Methods**
  Many machine learning problems are nonconvex. Essentially anything related to deep learning is. But so are clustering, topic models, pretty much any latent variable methods and pretty much anything else that's interesting in machine learning nowadays. Some of the acceleration techniques can help. For instance, my student Sashank Reddy[13] showed recently how to get good rates[14] of convergence[15] in this case.
  There are lots of techniques called Spectral Methods that can be used. Anima Anandkumar[16] has answered this in amazing detail in her recent Quora session[17]. Please read her responses since they're super detailed. In a nutshell, convex problems aren't the only ones that can be solved reliably. In some cases you can work out the mathematical equivalent of a puzzle to show that only a certain choice of parameters can make sense to find all the clusters, topics, relevant dimensions, neurons or whatever in your data. This is great if you are able and willing to throw a lot of math at it.
  There are lots of recent tricks when it comes to training Deep Networks. I'll get to them below but in some cases the goal isn't just optimization but to engineer a specific choice of solution (almost akin to *The Journey is the goal*).

- **Systems**
  Much of the reason why machine learning is becoming the key ingredient in pretty much anything related to people, measurements, sensors and data has to do with the breakthroughs over the past decade in scaling algorithms. It isn't a complete coincidence that Jeff Dean[18] gave half a dozen machine learning tutorials in the past year. For anyone who slept the past decade under a rock - he's the man behind MapReduce, the Google File System, BigTable, and another dozen of key technologies that have made Google great. Some facts about him can be found here[19].
  Joking aside, systems research offers a treasure trove of tools for solving problems that are *distributed, asynchronous, fault tolerant, scalable, and simple*. The latter is something that machine learning researchers often overlook. Simplicity is a feature, not a bug. Some of the basic techniques that will get you quite far:
  - **Distributed hash table**
    This is essentially what methods such as memcached[20], dynamo[21], pastry[22], or ceph[23] are built around. They all solve the problem - how to distribute objects over many machines in such a way as to avoid having to ask a central repository where things went. To make this work you need to encode the location in a randomized yet deterministic fashion (hence hashing). Moreover, you need to figure out who will take care of things if any machine fails.
    This is what we used for the data layout[24] in the Parameter Server. My student Mu Li[25] is the brains behind this project. See DMLC[26] for a collection of tools.

- **Consistency and Messaging**

  The godfather of all of this is Leslie Lamport's PAXOS[27] protocol. It solves the problem of having machines come to a consensus while not all machines are available at all times and some might fail (yes, I'm playing fast and loose here). If you've ever used version control you probably know how it works intuitively - you have lots of machines (or developers) generating updates (or pieces of code) and you want to combine them all in a way that makes sense (e.g. you shouldn't apply a diff twice) while not requiring that everyone talks to everyone all the time.

  In systems the solution is to use what's called a vector clock (see e.g. Google's Chubby[28]). We used a variant of this in the Parameter Server. The key difference was (all credits to Mu Li) to use vector clocks for parameter ranges. This ensures that you don't run out of memory for timestamps, just like a file system doesn't need a timestamp for every single byte.
- **Fault Tolerance, Scaling and the Cloud**

  The easiest way to teach yourself this is to run algorithms on Amazon AWS[29], Google GWC[30], Microsoft Azure[31], or on the many other providers[32] that you can find. It's quite exciting the first time you fire up 1,000 servers and you realize that you're now commanding what amounts to essentially a perfectly legal botnet. In one case while I used to work at Google we took over 5,000 high end machines somewhere in Europe for inference in topic models. This was a sizable fraction of a nuclear power plant what we racked up for an energy bill. My manager took me aside and told me that this was an expensive experiment ...

  Probably the easiest way to get started is to learn about docker[33]. They've been on a tear to develop lots of tools of making scaling easier. In particular Docker Machine[34] and Docker Cloud[35] are probably some of their nicest recent additions that allow you to connect to different clouds just like swapping printer drivers.
- **Hardware**
  It kind of sounds obvious but it really helps if you know what hardware your algorithms run on. This helps you to find out whether your code is anywhere near peak performance. For starters look at Jeff Dean's Numbers every engineer should know[36]. One of my favorite interview questions is (probably by now *was*) how fast the candidate's laptop is. Knowing what the limitations for the algorithm are really helps. Is it cache, memory bandwidth, latency, disks, etc. Anandtech[37] has really great introductory articles and reviews of microprocessor architectures and related stuff. Check them out whenever Intel / ARM / AMD releases new hardware.

- **Statistics**
  I've deliberately put this last. Simply since everyone thinks that this is key (yes, it is) and overlooks all the rest. Statistics is the key to let you ask good questions. It also helps you to understand how much of an approximation you're making when you are modeling some data.
  A lot of the improvements in anything from graphical models, kernel methods, deep learning, etc. really arise from being able to ask the right questions, aka, defining sensible objective functions that you can then optimize.
  - **Statistics Proper**

    A good intro is Larry Wasserman[38]'s book on All of Statistics[39]. Alternatively you can check out David McKay's Machine Learning[40] book. It's free (and huge and comprehensive). There are plenty of other great books around, such as those by Kevin Murphy[41], Chris Bishop[42], Trevor Hastie, Rob Tibshirani and Jerome Friedman[43]. And yes, Bernhard Scholkopf and I wrote one, too[44].
  - **Randomized Algorithms and Probabilistic Computing**
    This is essentially computer science addressing the same problems. The key difference is that they're used as a tool to design algorithms rather than a problem to fit parameters to. I really love the one by Michael Mitzenmacher and Eli Upfal[45]. It's deceptively easy to read even though it covers lots of profound problems. Another one, if you want to go deeper into tools is the one by the late Rajeev Motwani and Prabhakar Raghavan[46]. It's well written but harder to understand without having a good statistics background.

This reply is probably long enough now that hardly anyone will have made it to here. Hence I should keep the rest short. There is lots of awesome video content online. Many faculty by now have a YouTube channel where they post their classes. This helps when going through the sometimes tricky set of tools. Here's mine[47]. Nando de Freitas[48]' is much better.

And then there are tools. DMLC[49] is a good place to start. It has plenty of algorithms for distributed scalable inference. Including neural networks via MXNET.

Lots more things missing: programming languages, data sources, etc. But this reply is already way too long. More in the other answers.

80.6k ViewsView Upvotes[50]Answer requested by 352 people

Links

1. http://www.amazon.com/Linear-Algebra-Undergraduate-Texts-Mathematics/dp/0387964126
2. http://www.amazon.com/Linear-Analysis-Introductory-Cambridge-Mathematical/dp/0521655773
3. http://www.amazon.com/Numerical-Linear-Algebra-Lloyd-Trefethen/dp/0898713617
4. http://www.amazon.com/Numerical-Recipes-Scientific-Computing-Second/dp/0521431085/
5. http://www.amazon.com/Computations-Hopkins-Studies-Mathematical-Sciences/dp/1421407949/
6. http://stanford.edu/~boyd/cvxbook/
7. http://web.stanford.edu/~boyd/
8. http://www.mit.edu/~dimitrib/home.html
9. http://arxiv.org/abs/0911.0491
10. https://www.eecs.berkeley.edu/~brecht/papers/hogwildTR.pdf
11. http://people.rennes.inria.fr/Cedric.Herzet/Cedric.Herzet/Sparse_Seminar/Entrees/2012/11/12_A_Fast_Iterative_Shrinkage-Thresholding_Algorithmfor_Linear_Inverse_Problems_(A._Beck,_M._Teboulle)_files/Breck_2009.pdf
12. http://spams-devel.gforge.inria.fr/
13. http://www.cs.cmu.edu/~sjakkamr/
14. http://arxiv.org/abs/1603.06159
15. http://arxiv.org/abs/1603.06160
16. http://newport.eecs.uci.edu/anandkumar/
17. https://www.quora.com/profile/Anima-Anandkumar-1
18. http://research.google.com/pubs/jeff.html
19. http://www.informatika.bg/jeffdean
20. https://memcached.org/
21. http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf
22. http://research.microsoft.com/en-us/um/people/antr/PAST/pastry.pdf
23. http://docs.ceph.com/docs/hammer/rados/
24. https://www.cs.cmu.edu/~dga/papers/osdi14-paper-li_mu.pdf
25. http://www.cs.cmu.edu/~muli/
26. http://dmlc.ml/
27. http://research.microsoft.com/en-us/um/people/lamport/pubs/paxos-simple.pdf
28. http://blogoscoped.com/archive/2008-07-24-n69.html
29. http://aws.amazon.com/

30. http://console.google.com/
31. http://azure.microsoft.com/
32. http://serverbear.com/
33. http://www.docker.com/
34. https://docs.docker.com/machine/
35. https://docs.docker.com/cloud/
36. https://gist.github.com/jboner/2841832
37. http://www.anandtech.com/
38. http://www.stat.cmu.edu/~larry/
39. http://www.stat.cmu.edu/~larry/all-of-statistics/
40. http://www.inference.phy.cam.ac.uk/itprnn/book.pdf
41. https://mitpress.mit.edu/books/machine-learning-0
42. http://research.microsoft.com/en-us/um/people/cmbishop/prml/
43. http://statweb.stanford.edu/~tibs/ElemStatLearn/
44. https://mitpress.mit.edu/books/learning-kernels
45. http://www.amazon.com/Probability-Computing-Randomized-Algorithms-Probabilistic/dp/0521835402
46. http://www.amazon.com/Randomized-Algorithms-Rajeev-Motwani/dp/0521474655
47. https://www.youtube.com/user/smolix/playlists
48. https://www.youtube.com/user/ProfNandoDF
49. http://www.dmlc.ml/
50. https://www.quora.com/api/mobile_expanded_voter_list?key=ofTKrOfppsW&type=answer