On the Evolution of Machine Learning: from Linear Models to Neural Networks

Author: Reza Bosagh Zadeh
Interviewed by David Beyer

We recently interviewed Reza Zadeh (@Reza_Zadeh). Reza is a Consulting Professor in the Institute for Computational and Mathematical Engineering at Stanford University and a Technical Advisor to Databricks. His work focuses on Machine Learning Theory and Applications, Distributed Computing, and Discrete Applied Mathematics. More information on his website: http://reza-zadeh.com

Tell us a bit about your work at Stanford and Databricks

At Stanford, I designed and teach distributed algorithms and optimization (CME 323) as well as a course called discrete mathematics and algorithms (CME 305). In the discrete mathematics course, I teach algorithms from a completely theoretical perspective, meaning that it is not tied to any programming language or framework, and we fill up whiteboards with many theorems and their proofs.

On the more practical side, in the distributed algorithms class, we work with the Spark cluster programming environment. I spend at least half my time on Spark. So all the theory that I teach in regard to distributed algorithms and machine learning gets implemented and made concrete by Spark, and then put in the hands of thousands of industry and academic folks who use commodity clusters.

I started running MapReduce jobs at Google back in 2006, before Hadoop was really popular or even known; but MapReduce was already mature at Google. I was 18 at the time, and even then I could see clearly that this is something that the world needs outside of Google. So I spent a lot of time building and thinking about algorithms on top of MapReduce, and always worked to stay current, long after leaving Google. When Spark came along, it was nice that it was open-source and one could see its internals, and contribute to it. I felt like it was the right time to jump on board because the idea of an RDD was the right abstraction for much of distributed computing.

From your time at Google up to the present work you're doing with Spark, you have had the chance to see some of the evolution of machine learning as it ties to distributed computing. Can you describe that evolution?

Machine learning has been through several transition periods starting in the mid 90's. From 1995 – 2005, there was a lot of focus on natural language, search, and information retrieval. The machine learning tools were simpler than what we're using today; they include things like logistic regression, SVMs (support vector machines), kernels with SVMs, and PageRank. Google became immensely successful using these technologies, building major success stories like Google News and the Gmail spam classifier using easy-to-distribute algorithms for ranking and text classification – using technologies that were already mature by the mid 90's.

Then around 2005 neural networks started making a comeback. Neural networks are a technology from the 80's – some would even date them back to the 60's – and they've become "retrocool" thanks to their important recent advances in computer vision. Computer vision makes very productive use of (convolutional) neural networks. As that fact has become better established, neural networks are making their way into other applications, creeping into areas like natural language processing and machine translation.

But there's a problem: neural networks are probably the most challenging of all the mentioned models to distribute. Those earlier models have all had their training successfully distributed. We can use 100 machines and train a logistic regression or SVM without much hassle. But developing a distributed neural network learning setup has been more difficult.

So guess who's done it successfully? The only organization so far is Google; they are the pioneers, yet again. It's very much like the scene back in 2005 when Google published the MapReduce paper, and everyone scrambled to build the same infrastructure. Google managed to distribute neural networks, get more bang for their buck, and now everyone is wishing they were in the same situation. But they're not.

Why is an SVM or logistic regression easier to distribute than a neural network?

First of all, evaluating an SVM is a lot easier. After you've learned an SVM model or logistic regression model — or any linear model — the actual evaluation is very fast. Say you built a spam classifier. A new email comes along; to classify it as spam or not it takes very little time, because it's just one dot product (in linear algebra terms). When it comes to a neural network, you have to do a lot more computation — even after you have learned the model — to figure out the model's output. And that's not even the biggest problem.

A typical SVM might be happy with just a million parameters, but the smallest successful neural networks I've seen have around 6 million--and that's the absolutely smallest. Another problem is that the training algorithms don't benefit from much of optimization theory. Most of the linear models that we use have mathematical guarantees on when training is finished. They can guarantee when you have found the best model you're going to find. But the optimization algorithms that exist for neural networks don't afford such guarantees. You don't know after you've trained a neural network whether, given your setup, this is the best model you could have found. So you're left wondering if you would have a better model if you kept on training.

Do you think that level of optimization is theoretically possible, and we just haven't gotten there yet?

The optimization algorithms that would be necessary hit some of the biggest open problems in computer science, so I don't think there's much hope for machine learning researchers to try and find algorithms that are provably optimal for neural networks. They would have to really change neural networks in a drastic way. Neural networks are not difficult to optimize because of the shortcomings of machine learning researchers. It's because optimization is very difficult; it hits that NP-hard problem that we have difficulty solving in all of computer science, not just machine learning.

Okay, so given those difficulties and constraints ...

Those three things are the big problems for neural networks. We largely get past them by way of engineering and mathematical tricks, heuristics, adding more machines, and using new hardware (e.g., the use of GPUs).

Of those tricks, what do you think shows the most promise for being generally applicable across the industry?

GPUs have been tremendously successful in adding compute horsepower to neural networks, allowing us to get past many problems. As I said, you have to do a lot of computation to even evaluate a neural network. If you have a GPU, you can speed those computations many times over, because the same operations that go into drawing lots of pixels on your screen can be used to evaluate the neural network model very efficiently. That approach has been tremendously successful to the point where Nvidia — a primarily *hardware* development company — has started building specialized software tools for people who do computer vision and deep learning for neural networks.

Nvidia's big selling points are its graphics cards and its CUDA library. Researchers now buy the most expensive GPUs they can to take full advantage of Nvidia's software. And new software packages are being built on top of what Nvidia offers. Outside of Google, this is the biggest advance that we've seen — in that we can train these models with GPUs.

Meanwhile, Google is also talking about using GPUs. Jeff Dean gave a talk in my workshop at the NIPS (Neural Information Processing Systems) Conference last December in which he mentioned that they are beginning to use GPUs. Up to now they have been CPU-bound, which means they need to use tens of machines to get results comparable to what others are getting with just one or two graphics cards.

As neural networks become more powerful, do you see them subsuming more and more of the work that used to be the bread and butter of linear methods?

I think so, yes. Actually that's happening right now. There's always this issue that linear models can only discriminate linearly. In order to get non-linearities involved, you would have to add or change features, which involves a lot of work. For example, computer vision scientists spent a decade developing and tuning these things called SIFT features, which enable image classification and other vision tasks using linear methods. But then neural networks came along, SIFT features became unnecessary; the neural network approach is to make features automatically as part of the training.

But I think it's asking for too much to say neural networks can replace all feature construction techniques. I don't think that will happen. There will always be a place for linear models and good human-driven feature engineering. Having said that, pretty much any researcher who has been to the NIPS Conference is beginning to evaluate neural networks for their application. Everyone is testing whether their application can benefit from the non-linearities that neural networks bring.

It's not like we never had non-linear models before. We have had them, many of them. It's just that the neural network model happens to be particularly powerful. It can really work for some applications, and so it's worth trying. That's what a lot of people are doing. And when they see successes, they write papers about them. So far, I've seen successes in speech recognition, in computer vision, and in machine translation. It is a very wide array of difficult tasks, so there is good reason to be excited.

Why is a neural network so powerful compared to the traditional linear and non-linear methods that have existed up until now?

When you have a linear model, every feature is either going to hurt or help whatever you are trying to score. That's the assumption inherent in linear models. So the model might determine that if the feature is large, then it's indicative of class 1; but if it's small, it's indicative of class 2. Even if you go all the way up to very large values of the feature, or down to very small values of the feature, you will never have a situation where you say, "In this interval, the feature is indicative of class 1; but in another interval it's indicative of class 2."

That's too limited. Say you are analyzing images, looking for pictures of dogs. It might be that only a certain subset of a feature's values indicate whether it is a picture of a dog, and the rest of the values for that pixel, or for that patch of an image, indicate another class. You can't draw a line to define such a complex set of relationships. Non-linear models are much more powerful, but at the same time they're much more difficult to train. Once again, you run into those hard problems from optimization theory. That's why for a long while we thought that neural networks weren't good enough, because they would over-fit, or they were too powerful. We couldn't do precise, guaranteed optimization on them. That's why they died.

Neural networks were effectively dead for that period that I mentioned, from the mid-90's to about 2005. The NIPS Conference had a decline, a rapid decline, in papers that had the word "neural" in their titles. Ironically, if you had the word "neural" in your title at NIPS, you were

less likely to have your paper accepted. And then we started figuring out slightly better algorithms to train these things and also more data became available to make the training a little bit easier. Finally, neural networks were reborn as deep learning, and specifically used for computer vision, because that's where the first big successes occurred with these larger amounts of data and with the new training algorithms.

Within neural network theory, there are multiple branches and approaches to computer learning. Can you summarize some of the key approaches?

By far the most successful approach has been a supervised approach where an older algorithm, called backpropagation, is used to effectively build a neural network that has many different outputs. Let's look at a neural network construction that has become very popular, called Convolutional Neural Networks. The idea is that the machine learning researcher builds a model constructed of several layers, each of which handles connections from the previous layer in a different way.

In the first layer, you have a window that slides a patch across an image, which becomes the input for that layer. This is called a convolutional layer because the patch “convolves”, it overlaps with itself. Then several different types of layers follow. Each have different properties, and pretty much all of them introduce non-linearities.

The last layer has 10,000 potential neuron outputs; each one of those activations correspond to a particular label which identifies the image. The first class might be a cat; the second class might be a car; and so on for all the 10,000 classes that ImageNet has. If the first neuron is firing the most out of the 10,000 then the input is identified as belonging to the first class, a cat.

This kind of approach has been around for a long time, and it's been very popular. Now we can train it with new training algorithms and accelerate it with GPUs. We can use libraries to build and mix and match different layers, so that we can see what kinds of neural networks work best. This approach, neural networks with backpropagation, is by far the the most successful of any currently being used. The drawback of the supervised approach is that you must apply labels to images while training. This is a car. This is a zoo. Etc.

Right. And the unsupervised approach?

A less popular approach involves “autoencoders”, which are unsupervised neural networks. Here the neural network is not used to classify the image, but to compress it. You read the image in the same way I just described, by identifying a patch and feeding the pixels into a convolutional layer. Several other layers then follow, including a middle layer which is very small compared to the others. It has relatively few neurons. Basically you’re reading the image, going through a bottleneck, and then coming out the other side and trying to reconstruct the image.

No labels are required for this training, because all you are doing is putting the image at both ends of the neural network and training the network to make the image fit, especially in the

middle layer. Once you do that, you now have a neural network that knows how to compress images. And it's effectively giving you features that you can use in other classifiers. So if you have only a little bit of labeled training data, no problem — you always have a lot of images. Think of these images as non-labeled training data. You can use images to build an autoencoder, then from the autoencoder pull out features that are a good fit using a little bit of training data to find the neurons in your auto-encoded neural network that are susceptible to particular patterns.

This is what Google did. They effectively built an unsupervised network off of a large collection of images from the web; it helps that they have access to a big chunk of the world's images. With this network, they were able to determine, for example, which neurons are susceptible to firing when a cat is in the image. And from that they now have a very good cat classifier. That's a bit of a more "researchy" approach. You still need a little bit of labeled data to make it work, and it would be pretty hard to do outside of Google, since it requires the scale to read a lot more images than most organizations can handle.

There are some other approaches as well. Machine translation is one example; although it also uses a supervised technique.

Of the two approaches I just described, the one that I'm primarily focused on is the first type, supervised learning via backpropagation. I believe this approach makes what you're trying to learn clear, as opposed to have faith that an autoencoder will learn what you want. Plus, once you're done training, the neural network becomes immediately useful. You don't have to then find some auxiliary training data to identify which neurons are particularly sensitive to a particular class. So most of what I'm focusing on building on top of Spark leverages the supervised learning approach.

Speaking of Spark, can you provide a broad overview of the kinds of research that you published? Plus, what got you into to Spark? And where do you see that set of technologies heading?

Sure. I have published in distributed machine learning. And that obviously ties very nicely with Spark, because what I work on within Spark is the machine learning library. So whether it be writing code myself, reviewing code, or soliciting contributions from people who are experts in a particular machine learning area, I'm all over the board. My students spend time on this at Stanford. They spend time on the linear algebra primitives, because that's something we're particularly good at. I also work with folks in Berkeley; right now I'm trying to get some deep learning into Spark and neural networks, and that happens to be something that the folks in Berkeley and Stanford work on very actively.

As far as what got me into Spark — I've known Matei Zaharia, the creator of Spark, since we were both undergraduates at Waterloo. We would practice on the ACM programming team, back when I was a sophomore and he was a junior. And we actually interned at Google at the same time. He was working on developer productivity tools, completely unrelated to big data.

He worked at Google and never touched MapReduce, which was my focus--kind of funny given where he ended up.

Then Matei went to Facebook, where he worked on Hadoop and became immensely successful. During that time, I kept thinking about distributing machine learning and none of the frameworks that were coming out — including Hadoop — looked exciting enough for me to build on top of because I knew from my time at Google what was really possible.

At the highest scale?

Yes, and I was underwhelmed by Hadoop. Not Hadoop as the ecosystem as it's evolved to become these days, but by Hadoop MapReduce in particular. I knew how to build machine learning on top of it, but I concluded that MapReduce presented an inefficient way to address those challenges, and so I wasn't excited about trying that.

And then Spark came along. I first saw it in 2011, while I was still a student. I was running a group meeting, lining up speakers at Stanford, and I invited Matei (while he was a student at Berkeley) to come tell us about Spark. This was long before it was a very well-known project. I got excited when I saw how easy it was to build machine learning algorithms that were fault tolerant and distributed using, say, eight lines of code. I was hooked. I realized that there's some future in this, and decided I was going to try and help him out with it.

Tell us a bit about what Spark is, how it works, and why it's particularly useful for distributed machine learning.

Spark is a cluster computing environment that gives you a distributed vector that works similar to the vectors you're used to programming with on a single machine. Spark provides a distributed vector. You can't do everything you could with a regular vector; for example, you don't have arbitrary random access via indices. But you can, for example, intersect two vectors; you can union; you can sort. You can do all kinds of things that you would expect from a regular vector. These distributed vectors are called RDDs (resilient distributed datasets).

One reason Spark makes machine learning easy is that it works by keeping some important parts of the data in memory as much as possible without writing to disk. In a distributed environment, a typical way to get fault resilience is to write to disk, to replicate a disk across the network three times using HDFS. An alternative way to get fault resilience is to remember how you built your data, which is what Spark does.

What makes this suitable for machine learning is that the data can come into memory and stay there. If it doesn't fit in memory, that's fine too. It will get paged on and off a disk as needed. But the point is while it can fit in memory, it will stay there. This benefits any process that will go through the data many times — and that's most of machine learning. Almost every machine learning algorithm needs to go through the data tens, if not hundreds, of times.

So as memory technologies improve at the hardware level, Spark becomes more powerful.

Yes, memory is always getting cheaper, more available, and bigger. And that absolutely benefits Spark. Also SSDs (Solid State Drives) enable Spark, again because we do page to disk when data doesn't fit in memory. If you have an SSD, that's going to work much faster. More and more cluster computing environments are going to include SSDs. And SSDs are going to become faster themselves, so that will definitely be helpful.

Where do you see Spark vis-a-vis MapReduce? Is there a place for both of them for different kinds of workloads and jobs?

To be clear, Hadoop as an ecosystem is going thrive and be around for a long time. I don't think the same is true for the MapReduce component of the Hadoop ecosystem.

With regard to MapReduce, to answer your question, no, I don't think so. I honestly think that if you're starting a new workload, it makes no sense to start in MapReduce unless you have an existing code base that you need to maintain. Other than that, there's no reason. It's kind of a silly thing to do MapReduce these days: it's the difference between assembly and C++. It doesn't make sense to write assembly code if you can write C++ code.

If you had to project out just a little bit, what do you see the biggest developments ahead on the timeline for Spark and the Spark ecosystem? And what are the things you're most excited about?

Spark itself is pretty stable right now. The biggest changes and improvements that are happening right now and happening in the next couple years are in the libraries. The machine learning library, the graph processing library, the SQL library, and the streaming libraries are all being rapidly developed, and every single one of them has an exciting roadmap for the next two years at least. These are all features that I want, and it's very nice to see that they can be easily implemented. I'm also excited about community-driven contributions that aren't general enough to put into Spark itself, but that support Spark as a community-driven set of packages on http://spark-packages.org. I think those will also be very helpful to the long-tail of users.

Over time, I think Spark will become the de facto distribution engine on which we can build machine learning algorithms, especially at scale.

I really enjoyed this, Reza. Thanks so much for your time!