CHAPTER 14

You Are Here

Congratulations on making it this far! You now have a solid working knowledge of the current state of the art of convolutional neural networks for image recognition, using swift for tensorflow. Let's look toward the future by first looking at the past.

A (short and opinionated) history of computing

It is valuable to study the history of to understand its future. There are many trends that are obvious only in hindsight. So, let us go all the way back to the beginning. The birth of Silicon Valley was arguably an overflow of military computing funding in the aftermath of World War II. The military wanted to fund various things, but they could not build them themselves, and so they started buying hardware from various labs that were set up in the valley to construct transistors. This was the real genesis of Silicon Valley, the ability to build strange new things with the knowledge that there was a willing buyer for what were extremely beta technologies.

The Internet itself, then, was an outgrowth of the ARPANET project, an initiative by DARPA to network various previously unconnected servers. If we can connect computers together locally using a network, then extending the network a few miles down the road is a fairly logical next step. But to quote Metcalfe's law, as each new node was added, the value of the network grew exponentially. What is interesting then is that, at a certain point, the value of adding new nodes to the network exceeded the cost. At which point, the process of adding new computers to the network became self-sustaining and then grew to what we see today. Or rather, I would argue that at a certain point, the commercial value of the invention itself exceeded the cost to bootstrap it, and after that point, it was impossible to halt the growth of what became the Internet. The genie was out of the bottle, so to speak.

In the 1970s, a different phenomenon occurred with supercomputing and AI in particular. The military funded many different strategies in the field, which started making more and more outlandish claims in order to get a bigger piece of the pie. Once it became clear many of these approaches weren't going to work came the AI winter, when DARPA pulled funding for many of these projects and the field was forced to try and fend for itself. Without a wealthy benefactor, or more precisely a clear commercial plan, both supercomputing and AI fell on hard times. The UK and Japan experienced similar phenomena a decade later.

And so the supercomputer race failed for the most part. But computers had proven their value in general and so continued to become cheaper and cheaper in general. Personal computing took off and a similar scenario happened, whereas the value of a computer to individual users exceeded the threshold of cost, and so as a result, the personal computer revolution became self-sustaining. As a result of this massive interest into home computers came the PC revolution of the 1980s and 1990s. What is interesting to me in particular is the third-generation supercomputing wave of the late 1990s, which was largely the result of taking off the shelf commodity processors (which had progressed far faster than the specialized supercomputing manufacturers could ever dream of) and wiring them together using advanced networks in order to tackle problems in a distributed fashion. Commoditized general hardware beat building specialized processors and methodology. Most current/fourth-generation supercomputing follows this trend, using commodity computing hardware and focusing on custom networking to increase intraprocess communication.

History of GPUs

And so, to look at another wave, we can consider the story of video cards. Originally, monochrome color and basic text were all that computers could generate. Memory capacity then increased to where larger amounts of data could be stored, leading to color becoming possible and gradually increasing resolutions. At some point, rastering 3D graphics on the fly became possible, and the 3dfx brought the first real GPU to market. Using a graphics programming language, all of a sudden a whole new world of interactive experiences (aka games) became possible. And so, to mirror the Internet and personal computing waves of before, the commercial value of playing games created a self-sustaining revolution in chipsets, which is still going on today. The entire reason we are running models on graphics cards today is due to the popularity of video gaming decades ago.

GPUs are getting close to becoming consumed by commoditization as well. While the market for new experiences continues to grow at this point today, even budget cards support features such as 4k video, which would have been unthinkable a few years ago. Running nongame code (notably bitcoin and deep learning) on the GPU itself is an extremely recent innovation that has breathed new life into the market. The companies making these devices are quickly reaching the limits of raw processing to make all of this possible. They are trying to bring new hardware to market without straying too far from the gaming market which drives everything. This is a large part of the push for VR and AR experiences. As GPUs become more general, they are increasingly absorbing more and more of the compute stack previously only controlled by CPUs.

Cloud computing

Virtual machines have significantly changed how people interact with computing, even if they are not aware of it. At one point, setting up and configuring a server took days; now it can be done in seconds. This enables workflows where resources are spun up on demand and then promptly discarded. Software is increasingly run at higher and higher abstractions which has allowed entirely new approaches to become commonplace. This will have long-term ramifications that we cannot even fully comprehend today. The largest computing clusters in the world are not supercomputers but rather managed servers running thousands of virtual machines for the cloud providers.

Crossing the chasm

AI and ML are not new fields. Neural networks, in the form of the perceptron, were invented in 1958. Only recently with the mentioned advances in compute power and hardware have they become practical to implement. Moreover, I would argue that they have finally crossed the chasm from intellectual curiosity into something driving the bottom line at large companies. As such, they have made the necessary transition to become a self-sustaining technology like the given examples. Google could delete the tensorflow repository tomorrow. Nvidia could stop shipping video cards. But these techniques will continue to be refined and improved regardless because they have real-world practical use cases in the industry. As such, the genie is out of the bottle. There is no going back to the pre-AI world. One way or another, the gains that AI brings will be brought to every field.

Computer vision

Let us look at the big areas that I believe will be important for the next decade.

Direct applications

Many of the more advanced forms of computer vision are finally seeing the hardware and compute capacity needed to run them become mainstream. I am particularly interested in the field of real-time systems, be it cameras on self-driving cars, being able to analyze medical data in the field, or even simply finding new ways to use the cameras on mobile phones. This area is only just now beginning to be touched.

Indirect applications

A number of interesting problems that are not necessarily image related can be converted into images and then solved using CNN-style approaches. Historically, many of these techniques have been impractical from a resource standpoint, but as more and more AI-specific hardware becomes mainstream, a lot of approaches that were previously infeasible become doable. AlphaGo, as an example, is a large-scale reinforcement algorithm that converts the board game go's game state into an image representation and then applies an extremely large convolutional neural network to it. The basic approach, though, is a convolutional neural network built using residual layers and large-scale compute. When average researchers gain access to similar amounts of resources, I think many interesting new approaches will be found in fields that are just now starting to experiment with AI.

Natural language processing

By using big data approaches (e.g., data corpuses from Wikipedia, scanned books, and gathered from the Internet at large), simpler approaches suddenly become powerful by virtue of giving the machine a lot more information to work with. This in turn has direct financial ramifications (e.g., improving search and recommendation engines), and so a lot of resources are being poured into this right now. It is going to become commonplace eventually.

Reinforcement learning and GANs

I am somewhat bearish on these fields in the short term, in that they still seem to require massive amounts of resources and there are still not a lot of clear commercial applications at this point in time. Having said that, I believe that in the long term, this is the field that is most going to drive progress in AI/ML in general. Most improvements in computer vision are now very small incremental tweaks, and any time an idea shows promise upstream in RL, then very quickly people will be trying to use it elsewhere. Using synthetic data to train neural networks is the area that seems most poised to become a commercial driver in the near future. Supersampling/resolution is making its way into silicon and is clearly here to stay.

Simulations in general

The other interesting area that I think is poised to be revolutionized by neural techniques is physical simulations in general. A very large amount of compute power is thrown regularly at performing complicated simulations of interactions based upon physics. I'm bearish on neural networks replacing physical simulations directly, because there will always be a place for raw math, but using networks to simulate real-world datasets opens up an interesting window of being able to simulate simulations, so to speak, and by extension being able to build approximately correct models much, much more quickly than traditional approaches. If the neural network–based simulation proves itself, then the traditional method can be run as the final phase, giving the best of both worlds (e.g., fast experimentation and fundamental rigor when desired). There is a danger of the networks losing touch with reality (e.g., simulating the wrong things), but I believe that having domain experts will obviate this problem.

To infinity and beyond

My experience is that this field as a whole has no shortage of ideas right now. There are thousands of papers being published each year on arXiv, and the rate of submissions only continues to grow. Many other fields, in particular mathematics, seem finally convinced that deep learning techniques are here to stay and that they need to get on the bandwagon, and so many extremely smart people are out there doing these hello world exercises, the same as you. In the short term, this is creating a lot of churn. There are countless blog posts by people attempting to explain their new ideas and online debates over the best approaches. Every new major release of pytorch or tensorflow breaks existing projects in all sorts of exciting new ways. People throw up their hands at the complexity and decide they're going to create a new unified system for doing things, and voilà, there's yet another new framework. This is literally going on as we speak. The industry as a whole is lurching from shiny thing to shiny thing. The simple truth is that nobody really knows what the right path forward is. New techniques are being discovered daily, and deep learning approaches have brought together dozens of related fields. Neural network and big data approaches have proven themselves on disparate problems such as biology, astronomy, physics, and economics. Every field now has to learn computer science or they will get left behind by those who do.

And so let me tell you grizzled programmer story of the early days of iOS. With the second generation, Apple let people submit apps. There was a massive gold rush where people could (and did try to) ship almost everything under the sun. The next few years were interesting as more and more of the approaches finally stabilized and became popular. After a while, libraries and frameworks became standardized. To me, all of this deep learning hullabaloo is very much the same experience of yore.

Why Swift

Swift has been an interesting revolution within the iOS ecosystem. Objective-C was showing its age, and swift brought iOS programmers a long way forward in a hurry. Garbage collection is a traditional approach in this field that works well on systems with large amounts of memory and spare cycles to run garbage collection. But in production systems with hard real-time requirements, be they servers providing 24/7 packet handling guarantees or mobile devices with quasi-random use patterns, this approach doesn't work as well as would be desired. Android has tried to cover up this gap by getting manufacturers to ship more and more RAM with their devices, but this makes devices cost more, which is often not viable in the real world.

LLVM initially snuck into iOS in the form of automatic reference counting, a feature added to Objective-C to count/track memory cycles and by extension be able to manually add malloc and free calls for the developer. Once this tech had proven itself, by eliminating memory management from the day-to-day workflow of programmers, Lattner et al. set their sights significantly higher.

Swift is designed to be a modern language that does not look out of place to existing Objective-C programmers, and I feel like at this point it succeeded extremely well. It brought functional programming ideas and concepts into the world of iOS by making it easy to bridge between the worlds. Have a particular section of code that needs C raw memory access? Just drop down to raw memory access directly, and the compiler can put boundary checking on that entire region of code. Have an existing C library that needs to be brought to swift? Simply write a simple API layer that encapsulates your library. Then all the system-level communication for iOS (and Mac proper, eventually) was forced to go through a swift layer of indirection. In the short term, this was painful in that it forced coders to no longer be able to do direct system calls. But over time, this approach drastically modularized the codebase at the system level and isolated many different bugs in their own particular islands.

While Apple was eating their own dogfood, iOS developers were going through a similar transition. Many open source libraries sprung up in the early days, each with their own set of trade-offs and patterns. By moving to swift, this forced much of the ecosystem to either evolve or get stuck in the past. In turn, though, this transition allowed people to concentrate on higher-level problems and not get stuck on low-level details.

And so then Apple did the crucial final step of making the language open source and opening it up fully to outside developers to make contributions and shape its future. Anybody can contribute and thousands have now. It is extremely hard for new programming languages to come into being. Small niche languages usually toil in obscurity. Large companies push new languages on the world, but this top-down approach usually only works so long as the original company is driving progress.

So to me then the strengths of swift are manyfold. It is an easy-to-learn language for beginners. It has the support of a large benefactor (Apple) that is committed to its success, but not technically in charge. It has a diverse ecosystem of open source contributors and is solving real problems in the real world daily while utilizing decades of experience building C libraries. It brings functional programming concepts to the procedural world in a pragmatic way without forcing people to completely change how they have been doing things.

Why LLVM

Swift's real magic power, though, is that it is the original language of LLVM. Compilers have historically focused on generating really, really fast code. This is great for progress but also means that many implementations chase speed over doing things correctly, so to speak. What happened as a result is that we ended up with many different compilers generating slightly different code for dozens of slightly different computers, and then build systems became really large and complicated. Generating a new programming language became very difficult because people demanded performance out of the gate.

LLVM rebuilt the foundations of compiler theory and has spawned a renaissance in new languages by reunifying these worlds. At a high level, all you have to do is generate an IR, and then LLVM can figure out how to get it to run on your device. This means that many, many different languages are using LLVM now. As a direct result, by using LLVM, you get the collective improvements of many, many different ecosystems.

This is a little bit more work up front for the programmer in terms of complexity, but as a result fundamentally makes it possible for the compiler to do much, much more. We've seen amazing progress in the LLVM world; people have demonstrated running gigantic jobs on large clusters and other approaches.

Machine learning is still in its infancy in many ways. Single-GPU code is the largest paradigm. People write stuff for clusters, but it is still very much custom code most of the time. We've got a ton of experience with single instruction and single variable code (CPU-style programming), but historically single-instruction multidata code has been really hard to write. We end up with lots of hand-customized kernels for different things. This works fine in a general sense, in that programmers can make system calls and get optimized code, but it means that it is hard for programmers to easily take advantage of whatever hardware they have at hand.

Why MLIR

The end result is that there's been a tremendous amount of churn in the machine learning ecosystem in general the past few years. Each manufacturer ends up trying to build libraries to provide an optimal experience for their hardware. Researchers have tried to make tensorflow do many things it was never designed to do, and so trying to support every permutation has been difficult for Google. Pytorch effectively rebuilt a framework just to make generating CUDA code simpler. MLIR provides a convenient bridge between these worlds. Hardware manufacturers can simply focus on getting an IR together that generates code for their device. Coders can write in arguably whatever language they prefer, and then language wonks simply need to find a way to convert their LLVM AST to an MLIR syntax. Then we can dream of a future in which we take our swift (or any language that supports LLVM) code and can compile it for whatever back end we desire.

Why ML is the most important field

Machine learning has the ability to absorb all of the world's compute capacity for the next few decades. This quiet revolution will have ramifications in dozens of fields and domains. The more and more we make it easier to use these tools and make them able to flexibly scale up to work on larger and larger compute systems, the greater the long-term potential of humanity as a whole. Large-scale compute has the ability to fundamentally do things that have never been possible before.

Clusters are only increasing in size. But all this emphasis on scaling ignores the reality that more compute resources are available to the individual today than at any point in history. If you are willing to invest the time and energy now, then as these things continue to improve, you will be the first to be able to take advantage of this revolution.

Toward this end, you can take two paths. One is to pick a particular horse, be it hardware or framework, and put all your efforts behind it. The other is to focus on helping make it so no particular framework or technology gains control of the ecosystem. Getting all these sundry groups of people working together as a whole has the potential to fundamentally revolutionize this field.

The hardware is just now being figured out, but this is going to change dramatically in the next few years. The software is a bit rough around the edges right now, I will admit. But opportunity never comes wrapped up neatly in a package with a bow. More often than not, it looks like hard work. But a little bit of work today will leave you well positioned for whatever tomorrow brings.

Why now

Progress is the result of many, many people working together over the centuries, not isolated to any one place or time. By helping make machine learning more accessible, you are helping improve tools that will indirectly touch millions of other people's lives. This has the potential to allow progress on a scale never before seen in history.

Why you

You can wait for other people to bring you the future or help them build it. There's never been a better time to get started. The future is now! Come join us!