

O'REILLY®

Artificial Intelligence Now

Current Perspectives from O'Reilly Media



O'Reilly Media Inc.

Learn from experts.
Find the answers you need.



Sign up for a **10-day free trial** to get **unlimited access** to all of the content on Safari, including Learning Paths, interactive tutorials, and curated playlists that draw from thousands of ebooks and training videos on a wide range of topics, including data, design, DevOps, management, business—and much more.

Start your free trial at:

oreilly.com/safari

(No credit card required.)

O'REILLY®

Safari



AI is moving fast. Don't fall behind.

Early adopters of applied AI have a unique opportunity to invent new business models, reshape industries, and build the impossible.

Put AI to work — right now.



theaiconf.com

Artificial
Intelligence
CONFERENCE

PRESENTED BY



Artificial Intelligence Now

*Current Perspectives from
O'Reilly Media*

O'Reilly Media, Inc.

Artificial Intelligence Now

by O'Reilly Media, Inc.

Copyright © 2017 O'Reilly Media, Inc. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com/safari>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Editor: Tim McGovern

Production Editor: Melanie Yarbrough

Proofreader: Jasmine Kwityn

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Rebecca Demarest

February 2017: First Edition

Revision History for the First Edition

2017-02-01: First Release

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Artificial Intelligence Now*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-491-97762-0

[LSI]

Table of Contents

Introduction.....	ix
-------------------	----

Part I. The AI Landscape

1. The State of Machine Intelligence 3.0.....	3
Ready Player World	4
Why Even Bot-Her?	5
On to 11111000001	6
Peter Pan's Never-Never Land	7
Inspirational Machine Intelligence	8
Looking Forward	9
2. The Four Dynamic Forces Shaping AI.....	11
Abundance and Scarcity of Ingredients	11
Forces Driving Abundance and Scarcity of Ingredients	15
Possible Scenarios for the Future of AI	17
Broadening the Discussion	20

Part II. Technology

3. To Supervise or Not to Supervise in AI?.....	25
4. Compressed Representations in the Age of Big Data.....	29
Deep Neural Networks and Intelligent Mobile Applications	29

Succinct: Search and Point Queries on Compressed Data	
Over Apache Spark	31
Related Resources	32
5. Compressing and Regularizing Deep Neural Networks.	33
Current Training Methods Are Inadequate	33
Deep Compression	34
DSD Training	36
Generating Image Descriptions	39
Advantages of Sparsity	40
6. Reinforcement Learning Explained.	41
Q-Learning: A Commonly Used Reinforcement Learning	
Method	43
Common Techniques of Reinforcement Learning	45
What Is Reinforcement Learning Good For?	47
Recent Applications	47
Getting Started with Reinforcement Learning	48
7. Hello, TensorFlow!.	49
Names and Execution in Python and TensorFlow	50
The Simplest TensorFlow Graph	51
The Simplest TensorFlow Neuron	54
See Your Graph in TensorBoard	55
Making the Neuron Learn	56
Flowing Onward	59
8. Dive into TensorFlow with Linux.	61
Collecting Training Images	62
Training the Model	63
Build the Classifier	64
Test the Classifier	64
9. A Poet Does TensorFlow.	69
10. Complex Neural Networks Made Easy by Chainer.	77
Chainer Basics	78
Chainer's Design: Define-by-Run	79
Implementing Complex Neural Networks	83
Stochastically Changing Neural Networks	85
Conclusion	86

11. Building Intelligent Applications with Deep Learning and TensorFlow.....	87
Deep Learning at Google	87
TensorFlow Makes Deep Learning More Accessible	88
Synchronous and Asynchronous Methods for Training Deep Neural Networks	88
Related Resources	89

Part III. Homebuilt Autonomous Systems

12. How to Build a Robot That “Sees” with \$100 and TensorFlow. . .	93
Building My Robot	93
Programming My Robot	98
Final Thoughts	99
13. How to Build an Autonomous, Voice-Controlled, Face-Recognizing Drone for \$200.....	101
Choosing a Prebuilt Drone	101
Programming My Drone	102
Architecture	103
Getting Started	103
Flying from the Command Line	104
Flying from a Web Page	104
Streaming Video from the Drone	105
Running Face Recognition on the Drone Images	106
Running Speech Recognition to Drive the Drone	107
Autonomous Search Paths	108
Conclusion	109

Part IV. Natural Language

14. Three Three Tips for Getting Started with NLU.....	113
Examples of Natural Language Understanding	114
Begin Using NLU—Here’s Why and How	115
Judging the Accuracy of an Algorithm	116
15. Training and Serving NLP Models Using Spark.....	119
Constructing Predictive Models with Spark	120
The Process of Building a Machine Learning Product	120

Operationalization	122
Spark's Role	123
Fitting It into Our Existing Platform with IdiML	129
Faster, Flexible Performant Systems	131
16. Capturing Semantic Meanings Using Deep Learning.	133
Word2Vec	135
Coding an Example	137
Training the Model	138
fastText	139
Evaluating Embeddings: Analogies	140
Results	141

Part V. Use Cases

17. Bot Thots.	145
Text Isn't the Final Form	145
Discovery Hasn't Been Solved Yet	146
Platforms, Services, Commercial Incentives, and Transparency	147
How Important Is Flawless Natural Language Processing?	148
What Should We Call Them?	149
18. Infographic: The Bot Platforms Ecosystem.	151
19. Creating Autonomous Vehicle Systems.	155
An Introduction to Autonomous Driving Technologies	156
Autonomous Driving Algorithms	157
The Client System	163
Cloud Platform	167
Just the Beginning	169

Part VI. Integrating Human and Machine Intelligence

20. Building Human-Assisted AI Applications.	173
Orchestra: A Platform for Building Human-Assisted AI Applications	173
Bots and Data Flow Programming for Human-in-the-Loop Projects	174

Related Resources	175
21. Using AI to Build a Comprehensive Database of Knowledge. . .	177
Building the Largest Structured Database of Knowledge	178
Knowledge Component of an AI System	178
Leveraging Open Source Projects: WebKit and Gigablast	179
Related Resources	180

Introduction

The phrase “artificial intelligence” has a way of retreating into the future: as things that were once in the realm of imagination and fiction become reality, they lose their wonder and become “machine translation,” “real-time traffic updates,” “self-driving cars,” and more. But the past 12 months have seen a true explosion in the capacities as well as adoption of AI technologies. While the flavor of these developments has not pointed to the “general AI” of science fiction, it has come much closer to offering generalized AI *tools*—these tools are being deployed to solve specific problems. But now they solve them more powerfully than the complex, rule-based tools that preceded them. More importantly, they are flexible enough to be deployed in many contexts. This means that more applications and industries are ripe for transformation with AI technologies.

This book, drawing from the best posts on the [O’Reilly AI blog](#), brings you a summary of the current state of AI technologies and applications, as well as a selection of useful guides to getting started with deep learning and AI technologies.

Part I covers the overall landscape of AI, focusing on the platforms, businesses, and business models are shaping the growth of AI. We then turn to the technologies underlying AI, particularly deep learning, in **Part II**. **Part III** brings us some “hobbyist” applications: intelligent robots. Even if you don’t build them, they are an incredible illustration of the low cost of entry into computer vision and autonomous operation. **Part IV** also focuses on one application: natural language. **Part V** takes us into commercial use cases: bots and autonomous vehicles. And finally, **Part VI** discusses a few of the interplays

between human and machine intelligence, leaving you with some big issues to ponder in the coming year.

The AI Landscape

Shivon Zilis and James Cham start us on our tour of the AI landscape, with their most recent survey of the state of machine intelligence. One strong theme: the emergence of platforms and reusable tools, the beginnings of a canonical AI “stack.” Beau Cronin then picks up the question of what’s coming by looking at the forces shaping AI: data, compute resources, algorithms, and talent. He picks apart the (market) forces that may help balance these requirements and makes a few predictions.

The State of Machine Intelligence 3.0

Shivon Zilis and James Cham

Almost a year ago, we published our now-annual **landscape** of machine intelligence companies, and goodness have we seen a lot of activity since then. This year's landscape has *a third more companies* than our first one did two years ago, and it feels even more futile to try to be comprehensive, since this just scratches the surface of all of the activity out there.

As has been the case for the last couple of years, our fund still obsesses over “problem first” machine intelligence—we’ve invested in 35 machine intelligence companies solving 35 meaningful problems in areas from security to recruiting to software development. (Our fund focuses on the future of work, so there are some machine intelligence domains where we invest more than others.)

At the same time, the hype around machine intelligence methods continues to grow: the words “deep learning” now equally represent a series of meaningful breakthroughs (wonderful) but also a hyped phrase like “big data” (not so good!). We care about whether a founder uses the right method to solve a problem, not the fanciest one. We favor those who apply technology thoughtfully.

What’s the biggest change in the last year? We are getting inbound inquiries from a different mix of people. For **v1.0**, we heard almost exclusively from founders and academics. Then came a healthy mix of investors, both private and public. Now overwhelmingly we have

heard from existing companies trying to figure out how to transform their businesses using machine intelligence.

For the first time, a “one stop shop” of the machine intelligence stack is coming into view—even if it’s a year or two off from being neatly formalized. The maturing of that stack might explain why more established companies are more focused on building legitimate machine intelligence capabilities. Anyone who has their wits about them is still going to be making initial build-and-buy decisions, so we figured an early attempt at laying out these technologies is better than no attempt (see [Figure 1-1](#)).

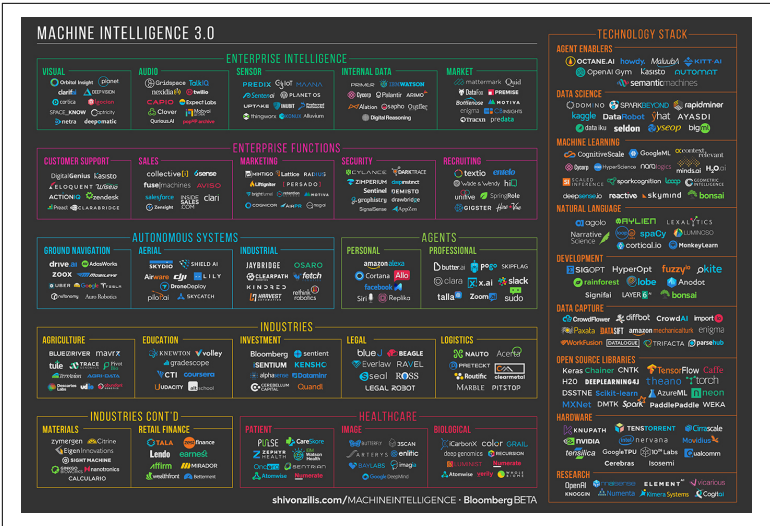


Figure 1-1. Image courtesy of Shivon Zilis and James Cham, designed by Heidi Skinner (a larger version can be found on [Shivon Zilis’ website](#))

Ready Player World

Many of the most impressive looking feats we’ve seen have been in the gaming world, from DeepMind beating Atari classics and the world’s best at Go, to the [OpenAI Gym](#), which allows anyone to train intelligent agents across an array of gaming environments.

The gaming world offers a perfect place to start machine intelligence work (e.g., constrained environments, explicit rewards, easy-to-compare results, looks impressive)—especially for reinforcement learning. And it is much easier to have a self-driving car agent go a

trillion miles in a simulated environment than on actual roads. Now we're seeing the techniques used to conquer the gaming world moving to the real world. A newsworthy example of game-tested technology entering the real world was when DeepMind used neural networks to make Google's data centers more efficient. This begs questions: What else in the world looks like a game? Or *what else in the world can we reconfigure to make it look more like a game?*

Early attempts are intriguing. Developers are dodging meter maids (brilliant—a modern day *Paper Boy*), categorizing cucumbers, sorting trash, and recreating the memories of loved ones as conversational bots. Otto's self-driving trucks delivering beer on their first commercial ride even seems like a bonus level from *Grand Theft Auto*. We're excited to see what new creative applications come in the next year.

Why Even Bot-Her?

Ah, the great chatbot explosion of 2016, for better or worse—we liken it to the mobile app explosion we saw with the launch of iOS and Android. The dominant platforms (in the machine intelligence case, Facebook, Slack, Kik) race to get developers to build on their platforms. That means we'll get some excellent bots but also many terrible ones—the joys of public experimentation.

The danger here, unlike the mobile app explosion (where we lacked expectations for what these widgets could actually do), is that *we assume anything with a conversation interface will converse with us at near-human level. Most do not.* This is going to lead to disillusionment over the course of the next year but it will clean itself up fairly quickly thereafter.

When our fund looks at this emerging field, we divide each technology into two components: the conversational interface itself and the “agent” behind the scenes that's learning from data and transacting on a user's behalf. While you certainly can't drop the ball on the interface, we spend almost all our time thinking about that behind-the-scenes agent and whether it is actually solving a meaningful problem.

We get a lot of questions about whether there will be “one bot to rule them all.” To be honest, as with many areas at our fund, we disagree on this. We certainly believe there will not be one *agent* to rule them

all, even if there is one interface to rule them all. For the time being, bots will be idiot savants: stellar for very specific applications.

We've **written a bit about this**, and the framework we use to think about how agents will evolve is a CEO and her support staff. Many Fortune 500 CEOs employ a scheduler, handler, a research team, a copy editor, a speechwriter, a personal shopper, a driver, and a professional coach. Each of these people performs a dramatically different function and has access to very different data to do their job. The bot/agent ecosystem will have a similar separation of responsibilities with very clear winners, and they will divide fairly cleanly along these lines. (Note that some CEOs have a chief of staff who coordinates among all these functions, so perhaps we will see examples of "one interface to rule them all.")

You can also see, in our landscape, some of the corporate functions machine intelligence will reinvent (most often in interfaces other than conversational bots).

On to 11111000001

Successful use of machine intelligence at a large organization is surprisingly binary, like flipping a stubborn light switch. It's hard to do, but once machine intelligence is enabled, an organization sees everything through the lens of its potential. Organizations like Google, Facebook, Apple, Microsoft, Amazon, Uber, and Bloomberg (our sole investor) bet heavily on machine intelligence and have its capabilities pervasive throughout all of their products.

Other companies are struggling to figure out what to do, as many boardrooms did on "what to do about the internet" in 1997. Why is this so difficult for companies to wrap their heads around? *Machine intelligence is different from traditional software*. Unlike with big data, where you could buy a new capability, machine intelligence depends on deeper organizational and process changes. Companies need to decide whether they will trust machine intelligence analysis for one-off decisions or if they will embed often-inscrutable machine intelligence models in core processes. Teams need to figure out how to test newfound capabilities, and applications need to change so they offer more than a system of record; they also need to coach employees and learn from the data they enter.

Unlike traditional hard-coded software, machine intelligence gives only probabilistic outputs. We want to ask machine intelligence to make subjective decisions based on imperfect information (eerily like what we trust our colleagues to do?). As a result, this new machine intelligence software will make mistakes, just like we do, and we'll need to be thoughtful about when to trust it and when not to.

The idea of this new machine trust is daunting and makes machine intelligence harder to adopt than traditional software. We've had a few people tell us that the biggest predictor of whether a company will successfully adopt machine intelligence is whether it has a C-suite executive with an advanced math degree. These executives understand it isn't magic—it is just (hard) math.

Machine intelligence business models are going to be different from licensed and subscription software, but we don't know how. Unlike traditional software, we still lack frameworks for management to decide where to deploy machine intelligence. Economists like Ajay Agrawal, Joshua Gans, and Avi Goldfarb have taken the **first steps** toward helping managers understand the economics of machine intelligence and predict where it will be most effective. But there is still a lot of work to be done.

In the next few years, the danger here isn't what we see in dystopian sci-fi movies. The real danger of machine intelligence is that *executives will make bad decisions about what machine intelligence capabilities to build*.

Peter Pan's Never-Never Land

We've been wondering about the path to grow into a large machine intelligence company. Unsurprisingly, there have been many machine intelligence acquisitions (Nervana by Intel, Magic Pony by Twitter, Turi by Apple, Metamind by Salesforce, Otto by Uber, Cruise by GM, SalesPredict by Ebay, Viv by Samsung). Many of these happened fairly early in a company's life and at quite a high price. Why is that?

Established companies struggle to understand machine intelligence technology, so it's painful to sell to them, and the market for buyers who can use this technology in a self-service way is small. Then, if you do understand how this technology can supercharge your orga-

nization, you realize it's so valuable that you want to hoard it. Businesses are saying to machine intelligence companies, "forget you selling this technology to others, I'm going to buy the whole thing."

This absence of a market today makes it difficult for a machine intelligence startup, especially horizontal technology providers, to "grow up"—hence the Peter Pans. *Companies we see successfully entering a long-term trajectory can package their technology as a new problem-specific application for enterprise or simply transform an industry themselves as a new entrant* (love this). We flagged a few of the industry categories where we believe startups might "go the distance" in this year's landscape.

Inspirational Machine Intelligence

Once we do figure it out, machine intelligence can solve much more interesting problems than traditional software can. We're thrilled to see so many smart people applying machine intelligence for good.

Established players like **Conservation Metrics** and **Vulcan Conservation** have been using deep learning to protect endangered animal species; the ever-inspiring team at **Thorn** is constantly coming up with creative algorithmic techniques to protect our children from online exploitation. The philanthropic arms of the tech titans joined in, enabling nonprofits with free storage, compute, and even developer time. Google partnered with nonprofits to found **Global Fishing Watch** to detect illegal fishing activity using satellite data in near real time, satellite intelligence startup **Orbital Insight** (in which we are investors) partnered with **Global Forest Watch** to detect illegal logging and other causes of global forest degradation. Startups are getting into the action, too. The **Creative Destruction Lab** machine intelligence accelerator (with whom we work closely) has companies working on problems like earlier **disease detection** and **injury prevention**. One area where we have seen some activity but would love to see more is machine intelligence to **assist the elderly**.

In talking to many people using machine intelligence for good, they all cite the critical role of open source technologies. In the last year, we've seen the launch of **OpenAI**, which offers everyone access to world-class research and environments, and better and better releases of TensorFlow and Keras. Nonprofits are always trying to do more with less, and machine intelligence has allowed them to extend the scope of their missions without extending their budgets. Algo-

rithms allow nonprofits to inexpensively scale what would not be affordable to do with people.

We also saw growth in universities and corporate think tanks, where new centers like USC's Center for AI in Society, Berkeley's Center for Human Compatible AI, and the multiple-corporation Partnership on AI study the ways in which machine intelligence can help humanity. The White House even got into the act: after a series of workshops around the US, it published a 48-page report outlining recommendations for applying machine intelligence to safely and fairly address broad social problems.

On a lighter note, we've also heard whispers of more artisanal versions of machine intelligence. Folks are doing things like using computer vision algorithms to help them choose the best cocoa beans for high-grade chocolate, write poetry, cook steaks, and generate musicals.

Curious minds want to know. If you're working on a unique or important application of machine intelligence, we'd love to hear from you.

Looking Forward

We see all this activity only continuing to accelerate. The world will give us more open sourced and commercially available machine intelligence building blocks; there will be more data; there will be more people interested in learning these methods; and there will always be problems worth solving. We still need ways of explaining the difference between machine intelligence and traditional software, and we're working on that. The value of code is different from data, but what about the value of the model that code improves based on that data?

Once we understand machine intelligence deeply, we might look back on the era of traditional software and think it was just a prologue to what's happening now. We look forward to seeing what the next year brings.

Thank You

A massive thank you to the Bloomberg Beta team, David Klein, Adam Gibson, Ajay Agrawal, Alexandra Suich, Angela Tranyens,

Anthony Goldblum, Avi Goldfarb, Beau Cronin, Ben Lorica, Chris Nicholson, Doug Fulop, Dror Berman, Dylan Tweney, Gary Kazantsev, Gideon Mann, Gordon Ritter, Jack Clark, John Lilly, Jon Lehr, Joshua Gans, Matt Turck, Matthew Granade, Mickey Graham, Nick Adams, Roger Magoulas, Sean Gourley, Shruti Gandhi, Steve Jurvetson, Vijay Sundaram, Zavain Dar, and for the help and fascinating conversations that led to this year's report!

Landscape designed by **Heidi Skinner**.

Disclosure: Bloomberg Beta is an investor in Alation, Arimo, Aviso, Brightfunnel, Context Relevant, Deep Genomics, Diffbot, Digital Genius, Domino Data Labs, Drawbridge, Gigster, Gradescope, Graphistry, Gridspace, Howdy, Kaggle, Kindred.ai, Mavrx, Motiva, Popup Archive, Primer, Sapho, Shield.AI, Textio, and Tule.

Shivon Zilis and James Cham

Shivon Zilis is a partner and founding member of Bloomberg Beta, an early-stage VC firm that invests in startups making work better, with a focus on machine intelligence. She's particularly fascinated by intelligence tools and industry applications. Like any good Canadian, she spends her spare time playing hockey and snowboarding. She holds a degree in economics and philosophy from Yale.

James Cham is a partner at Bloomberg Beta based in Palo Alto. He invests in data-centric and machine learning-related companies. He was a principal at Trinity Ventures and vice president at Bessemer Venture Partners, where he worked with investments like Dropcam, Twilio, and LifeLock. He's a former software developer and management consultant. He has an MBA from the Massachusetts Institute of Technology and was an undergraduate in computer science at Harvard College.

The Four Dynamic Forces Shaping AI

Beau Cronin

There are four basic ingredients for making AI: *data*, *compute resources* (i.e., hardware), *algorithms* (i.e., software), and the *talent* to put it all together. In this era of deep learning ascendancy, it has become conventional wisdom that data is the most differentiating and defensible of these resources; companies like Google and Facebook spend billions to develop and provide consumer services, largely in order to amass information about their users and the world they inhabit. While the original strategic motivation behind these services was to monetize that data via ad targeting, both of these companies—and others desperate to follow their lead—now view the creation of AI as an equally important justification for their massive collection efforts.

Abundance and Scarcity of Ingredients

While all four pieces are necessary to build modern AI systems, what we'll call their “scarcity” varies widely. Scarcity is driven in large part by the balance of supply and demand: either a tight supply of a limited resource or a heavy need for it can render it more scarce. When it comes to the ingredients that go into AI, these supply and demand levels can be influenced by a wide range of forces—not just technical changes, but also social, political, and economic shifts.

Fictional depictions can help to draw out the form and implications of technological change more clearly. So, before turning to our present condition, I want to briefly explore one of my favorite sci-fi treatments of AI, from David Marusek's tragically under-appreciated novel *Counting Heads* (A Tor Book). Marusek paints a 22nd-century future where an AI's intelligence, and its value, scales directly with the amount of "neural paste" it runs on—and that stuff isn't cheap. Given this hardware (wetware?) expense, the most consequential intelligences, known as mentars, are sponsored by—and allied with—only the most powerful entities: government agencies, worker guilds, and corporations owned by the super-rich "affs" who really run the world. In this scenario, access to a powerful mentar is both a signifier and a source of power and influence.

Translating this world into our language of AI ingredients, in *Counting Heads* it is the hardware substrate that is far and away the scarcest resource. While training a new mentar takes time and skill, both the talent and data needed to do so are relatively easy to come by. And the algorithms are so commonplace as to be beneath mention.

With this fictional example in mind, let's take stock of the relative abundance and scarcity of these four ingredients in today's AI landscape:

- The algorithms and even the specific software libraries (e.g., [TensorFlow](#), [Torch](#), [Theano](#)) have become, by and large, a matter of public record—they are simply there for the taking on GitHub and the ArXiv.
- Massive compute resources (e.g., [Amazon AWS](#), [Google](#), [Microsoft Azure](#)) aren't without cost, but are nevertheless fully commoditized and easily accessible to any individual or organization with modest capital. A run-of-the-mill AWS instance, running about \$1 an hour, would have been at or near the top of the world supercomputer rankings in the early 1990s.
- The talent to build the most advanced systems is much harder to come by, however. There is a genuine shortage of individuals who are able to work fluently with the most effective methods, and even fewer who can advance the state of the art.
- Finally, the massive data sets necessary to train modern AIs are hardest of all to obtain, in some cases requiring a level of capital expenditure and market presence that only the largest organizations can muster. While data is a non-rival good, and therefore

could be shared widely after collection, in practice the largest and most valuable data sets are closely guarded—they form the true barriers to competition in today’s landscape.

You could summarize the current situation with **Figure 2-1**, by Beau Cronin.

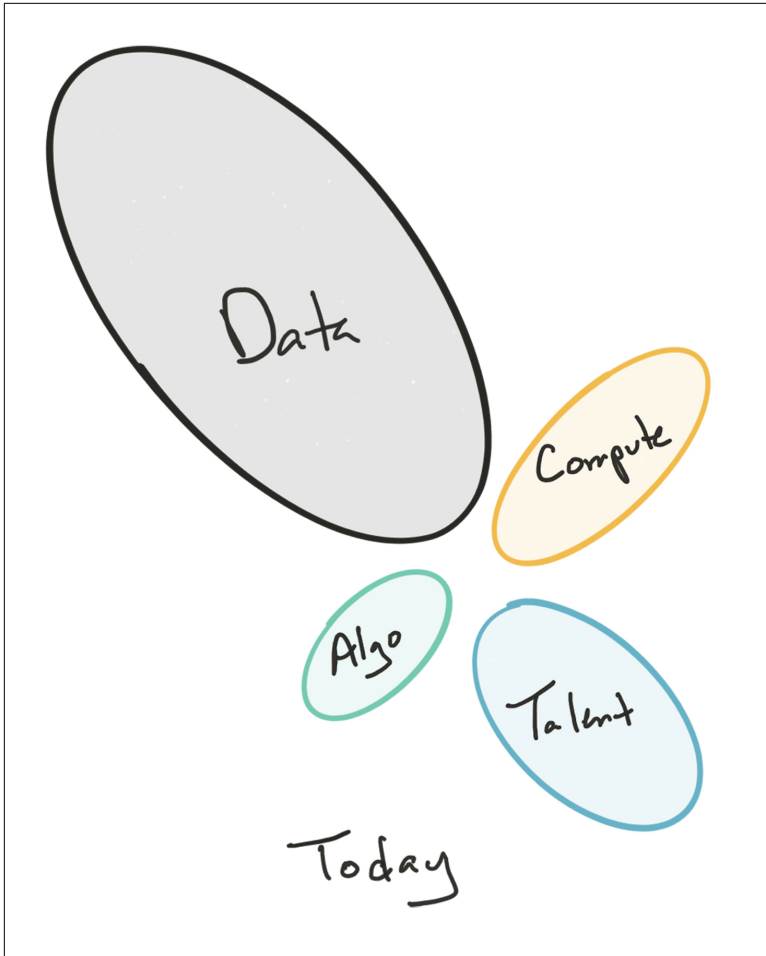


Figure 2-1. Current AI situation (image courtesy of Beau Cronin)

NOTE

(Note that bubble size here is scarcity, not importance —these resources are all necessary.)

But this particular distribution of AI ingredient scarcity is not the only possible configuration, and it is in fact quite new (though [see this Edge.org article](#) for the argument that data sets are the fundamentally scarce resource). For example, the realization that very large and expensive-to-gather data sets are in fact crucial in creating the most valuable and impactful AI systems is usually [credited to Google](#), and is largely a phenomenon of the last 10–15 years. The demand for these data sets has therefore grown fantastically; their supply has increased as well, but not at the same rate.

The conventional wisdom was quite different around the turn of the century, when many of the smartest practitioners viewed the relative importance of the ingredients quite differently, leading to a different distribution of demand. People knew that data was essential, to be sure, but the scale at which it was needed simply wasn't appreciated. On the other hand, reliance on different (and often secret) algorithms for competitive differentiation was much more widespread—even if the *actual* superiority of these proprietary methods didn't live up to their *perceived* worth. We might caricature the scarcity distribution of this pre-big data AI era as in [Figure 2-2](#).

Debate the details if you will, but the overall balance was certainly quite different: this was a time when the most successful approach to natural language understanding, for example, was to construct parsers that included detailed grammar rules for the language in question, rather than today's practice of extracting statistical patterns from very large text corpora. And eliciting explicit knowledge from experts through in-person interviews and encoding the results into formal decision rules was a dominant learning method, rather than today's standard approach of extracting patterns automatically from many examples.

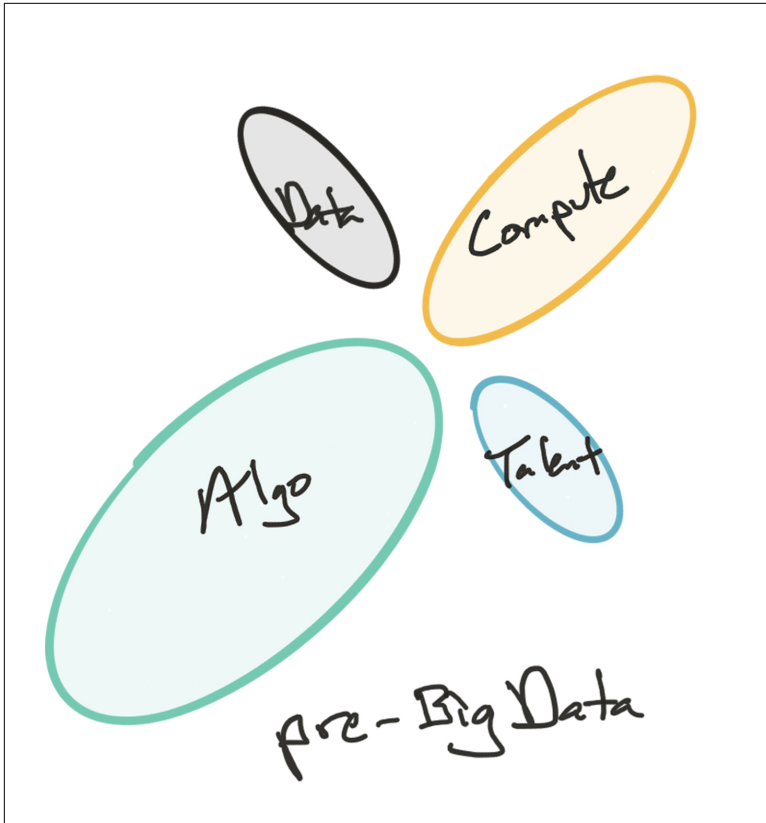


Figure 2-2. Pre-big data AI (image courtesy of Beau Cronin)

Forces Driving Abundance and Scarcity of Ingredients

This glimpse at recent history raises the question: will our current scarcity distribution hold, or instead shift again? Which immediately begs another question: what are the factors that affect the absolute and relative scarcity levels of each ingredient?

We don't have room here for a complete exploration of the many variables, but a brief tour should at least get the critical juices flowing.

New *technology advances* obviously drive change in AI. Perhaps a new kind of sensor will reach a price/performance tipping point that allows it to be deployed at massive scale, transforming our ability to

observe and track human behavior. New chip architectures or even circuit substrates may emerge that fundamentally alter the capital requirements for running the most effective AI methods—either pushing them up, so that the needed computing resources are no longer a cheap commodity, or down, so that highly capable intelligences are significantly cheaper to train and run. Or new algorithms could emerge that require *much* less data to train: it is widely noted that, unlike today’s machine learning methods, humans do not need thousands or millions of labeled examples to make new distinctions, but can instead generalize in very nuanced ways from just a handful of cases. In fact, such data-efficient methods are undergoing intense development in academic and corporate research labs today, and constitute one of the most active areas of the field. What would happen to the balance of scarcity if these algorithms proved their effectiveness and became widely available?

Technology is not developed in a vacuum, but rather constitutes just one major element in the larger socioeconomic system. Shifts may occur in the *political and public opinion* landscape around issues of privacy, economic inequality, government power, and virtual or physical security. These changes in public sentiment can themselves be motivated by technological advances, and can in turn feed back to influence the pace and direction of those technology developments. Such policy and political shifts might render data collection harder (or easier), which could influence which algorithm classes are the most valuable to invest in.

The *larger economic picture* can also change the context in which AI is developed. For example, the skills distribution of the workforce will inevitably shift over time. As the simplest example, the current scarcity of qualified AI architects and developers will surely drive more people to enter the field, increasing the supply of qualified people. But other shifts are possible, too: continued difficulty in finding the right workers could tip the balance toward algorithms that are easier to train for a class of workers with less rarefied skills, for example. Or a new technology may emerge that requires a new mix of skills to reach its full potential—say, more **coach or trainer** than today’s applied mathematicians and engineers.

Possible Scenarios for the Future of AI

With these and other possible factors in mind, it should be clear that today's status quo is at least subject to question—while it might endure for some time, it's far more likely to represent a temporary equilibrium. Looking forward, a large number of scenarios are possible; we only have room to describe a few of them here, but I hope that this piece sparks a bit of discussion about the full range of conceivable outcomes.

First, the baseline scenario is a straightforward extrapolation from our current situation: data, at large scale and of high quality, remains the key differentiator in constructing AIs. New hardware architectures that accelerate the most important learning algorithms may also come into play, but the role of these new substrates remains secondary. The net effect is that AI remains a capital-intensive and strong-get-stronger affair: while there is near-universal consumer access to no-cost, shared, cloud-based AIs, these systems are exclusively created by, and ultimately reflect the interests and priorities of, highly resourced organizations.

This is the future that Kevin Kelly foresees in his recent book, *The Inevitable* (Penguin Random House):

The bigger the network, the more attractive it is to new users, which makes it even bigger and thus more attractive, and so on. A cloud that serves AI will obey the same law. The more people who use an AI, the smarter it gets. The smarter it gets, the more people who use it. The more people who use it, the smarter it gets. And so on. Once a company enters this virtuous cycle, it tends to grow so big so fast that it overwhelms any upstart competitors. As a result, our AI future is likely to be ruled by an oligarchy of two or three large, general-purpose cloud-based commercial intelligences.

In a second scenario, developments in the technical or political sphere alter the fundamental dynamics of access to data, making training data sets accessible to a much broader range of actors. This in turn gradually loosens the near-monopoly on talent currently held by the leading AI companies, which the top recruits no longer in thrall to the few sources of good data. Assuming the necessary compute remains a cheap commodity, this is a more “open” world (Figure 2-3), one in which it's easier to imagine powerful AIs being developed that reflect the interests of a broader range of individuals and organizations.



Figure 2-3. Open world AI (image courtesy of Beau Cronin)

We can also consider more exotic scenarios, such as [Figure 2-4](#). Consider a scenario in which the human talent element becomes even *more* differentiating and scarce than today (i.e., a world where the most effective AI algorithms must be taught and trained, and where some people are much better at this than others). Data, compute power, and the “blank slate” software are all available off-the-shelf, but the most gifted AI teachers are very highly sought-after. This seems far-fetched in our current technosocial landscape, but I wouldn’t be surprised if, one way or another, the scarcity landscape of the year 2030 didn’t seem at least this outlandish.

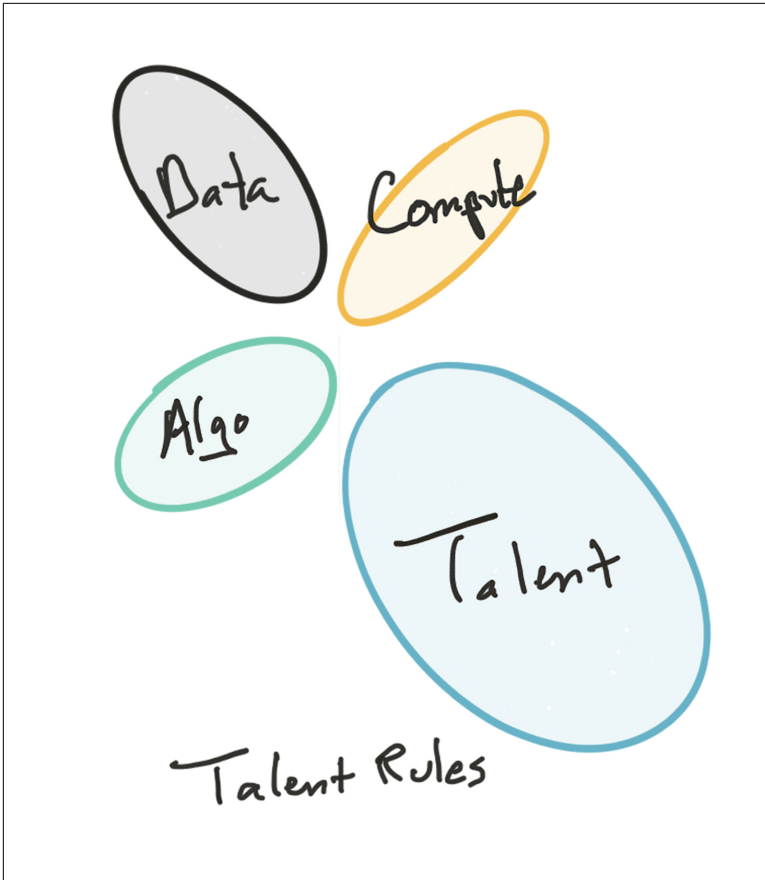


Figure 2-4. Exotic AI scenario (image courtesy of Beau Cronin)

Several early readers of this piece had their own pet scenarios in mind—including, I was glad to see, storylines in which the general public awoke to the value of personal data and demanded the right to control and profit from access to it—and I hope that they and others will share and debate them in a public forum. How do you see the driving forces playing out in the years to come, and what scenarios do those developments suggest?

Broadening the Discussion

Two tired frames currently dominate the popular discussion about the future of AI: Will robots and automation take all the jobs? And how long after that before a superintelligent AI kills or enslaves us all? These derpy discussions distract us from the issues around AI that are far more likely to impact us, for better and worse, in the next few decades. AI will certainly change our lives and livelihoods, but the ways in which this plays out will depend largely on which aspects of AI creation remain difficult versus easy, expensive versus affordable, exclusive versus accessible, and serious versus child's play.

Why does this ultimately matter? As Ben Lorica and Mike Loukides wrote in their report “[What Is Artificial Intelligence?](#)”:

If AI research becomes the sole province of the military, we will have excellent auto-targeting drones; if AI research becomes the sole province of national intelligence, we will have excellent systems for surreptitiously listening to and understanding conversations. Our imaginations will be limited about what else AI might do for us, and we will have trouble imagining AI applications other than murderous drones and the watchful ear of Big Brother. We may never develop intelligent medical systems or robotic nurses' aides.

This, in the end, is why it is important to debate and discuss these scenarios: our AI future is being written as we speak. While it seems unlikely today that, say, the military will dominate the field, it is entirely possible that large companies will. If this outcome is not the one you want, then the analysis here suggests that the key areas to examine are not algorithms or hardware, but rather the data and the talent. As [David Chapman](#) points out, “Given sufficient confidence that ‘deep learning would solve multi-billion-dollar problem X, if only we had the data,’ getting the resources to gather the data shouldn’t be difficult.” This even in the face of very large up-front capital outlay.

Today, the center of the AI universe has shifted from academic labs to applied research labs at large technology companies. Whether that remains the case 5, 10, or 20 years from now depends in large part on whether it becomes possible for that activity to happen elsewhere. Will it ever be possible for the next great AI to be born in a garage?

*I'd like to thank **Peter Schwartz**, who I had the honor and pleasure of working with for much of 2015. Those who are familiar with his groundbreaking scenario-planning framework will see his methodological fingerprints all over this piece, though of course I reserve all idiocies for myself. Thanks also to Miles Brundage, James Cham, David Chapman, and Dan Marthaler for insightful feedback; I probably should have taken more of their advice.*

Beau Cronin

Beau cofounded two startups based on probabilistic inference, the second of which was acquired by Salesforce in 2012. He now works as Head of Data at 21 Inc. He received his PhD in computational neuroscience from MIT in 2008.

Technology

We now take a deep dive into the technical underpinnings of artificial intelligence. The first three articles in this section discuss issues of broad AI significance before we shift our focus to deep learning. To kick things off, Mike Loukides discusses the balance of supervised and unsupervised learning—in both the human and machine contexts. Surprisingly, although we think of humans as the paradigm unsupervised learners, he points out that as a species, most of our learning is quite supervised. Junling Hu then gives an overview of reinforcement learning, with tips for implementation and a few signal examples. Ben Lorica then dives into compressed representations of deep learning models, in both mobile and distributed computing environments, and Song Han picks up that theme with a deep dive into compressing and regularizing deep neural networks. Next, we shift to deep learning with tutorials for Tensorflow from Aaron Schumacher and Justin Francis, and Mike Loukides’ account of learning “Tensorflow for poets”—as a poet! An alternative to Tensorflow that offers a “define by run” approach to deep learning is Chainer, the subject of Shohei Hido’s contribution. Finally, some thoughts from Rajat Monga on deep learning, both at Google and in any business model, round out the section.

To Supervise or Not to Supervise in AI?

Mike Loukides

One of the truisms of modern AI is that the next big step is to move from supervised to unsupervised learning. In the last few years, we've made tremendous progress in supervised learning: photo classification, speech recognition, even playing Go (which represents a partial, but only partial, transition to unsupervised learning). Unsupervised learning is still an unsolved problem. As **Yann LeCun** says, "We need to solve the unsupervised learning problem before we can even think of getting to true AI."

I only partially agree. Although AI and human intelligence aren't the same, LeCun appears to be assuming that unsupervised learning is central to human learning. I don't think that's true, or at least, it isn't true in the superficial sense. Unsupervised learning is critical to us, at a few very important stages in our lives. But if you look carefully at how humans learn, you see surprisingly little unsupervised learning.

It's possible that the the first few steps in language learning are unsupervised, though it would be hard to argue that point rigorously. It's clear, though, that once a baby has made the first few steps—once it's uttered its first ma-ma-ma and da-da-da—the learning process takes place in the context of constant support from parents, from siblings, even from other babies. There's constant feedback: praise for new words, attempts to communicate, and even preschool teachers saying, "Use your words." Our folktales recognize the same process.

There are many stories about humans raised by wolves or other animals. In none of those stories can the human, upon re-entering civilization, converse with other humans. This suggests that unsupervised learning may get the learning process started initially, but once the process has been started, it's heavily supervised.

Unsupervised learning may be involved in object permanence, but endless games of “peekaboo” should certainly be considered training. I can imagine a toddler learning some rudiments of counting and addition on his or her own, but I can't imagine a child developing any sort of higher mathematics without a teacher.

If we look at games like Chess and Go, experts don't achieve expertise without long hours of practice and training. Lee Sedol and Garry Kasparov didn't become experts on their own: it takes a tremendous investment in training, lessons, and directed study to become a contender even in a local tournament. Even at the highest professional levels, champions have coaches and advisors to direct their learning.

If the essence of general intelligence isn't unsupervised learning, and if unsupervised learning is a prerequisite for general intelligence, but not the substance, what should we be looking for? Here are some suggestions.

Humans are good at thinking by analogy and relationship. We learn something, then apply that knowledge in a completely different area. In AI, that's called “transfer learning”; I haven't seen many examples of it, but I suspect it's extremely important. What does picture classification tell us about natural language processing? What does fluid dynamics tell us about electronics? Taking an idea from one domain and applying it to another is perhaps the most powerful way by which humans learn.

I haven't seen much AI research on narrative, aside from projects to **create simple news stories from event logs**. I suspect that researchers undervalue the importance of narrative, possibly because our ability to create narrative has led to many false conclusions. But if we're anything, we're not the “rational animal” but the “storytelling animal,” and our most important ability is pulling disparate facts together into a coherent narrative. It's certainly true that our narratives are frequently wrong when they are based on a small number of events: a quintessentially human example of “overfitting.”

But that doesn't diminish their importance as a key tool for comprehending our world.

Humans are good at learning based on small numbers of examples. As one [redditor](#) says, "You don't show a kid 10,000 pictures of cars and houses for him or her to recognize them." But it's a mistake to think that tagging and supervision aren't happening. A toddler may learn the difference between cars and houses with a half dozen or so examples, but only with an adult saying, "That's a car and that's a house" (perhaps while reading a picture book). The difference is that humans do the tagging without noticing it, and the toddler shifts context from a 2D picture book to the real world without straining. Again, our ability to learn based on a small number of examples is both a strength and a weakness: we're plagued by overfitting and "truths" that are no more than prejudices. But our ability to learn based on a relatively small number of examples is important. Lee Sedol has probably played tens of thousands of Go games, but he certainly hasn't played millions.

I'm not arguing that unsupervised learning is unimportant. We may discover that unsupervised learning is an important prerequisite to other forms of learning, that unsupervised learning starts the learning process. It may be a necessary step in evolving from narrow AI to general AI. By sorting inputs into unlabeled categories, unsupervised learning might help to reduce the need for labeled data and greatly speed the learning process. But the biggest project facing AI isn't making the learning process faster and more efficient. It's moving from machines that solve one problem very well (such as playing Go or generating [imitation Rembrandts](#)) to machines that are flexible and can solve many unrelated problems well, even problems they've never seen before. If we really want general intelligence, we need to think more about transferring ideas from one domain to another, working with analogies and relationships, creating narratives, and discovering the implicit tagging and training that humans engage in constantly.

Mike Loukides

Mike Loukides is Vice President of Content Strategy for O'Reilly Media, Inc. He's edited many highly regarded books on technical subjects that don't involve Windows programming. He's particularly interested in programming languages, Unix and what passes for

Unix these days, and system and network administration. Mike is the author of *System Performance Tuning* and a coauthor of *Unix Power Tools* (O'Reilly). Most recently, he's been fooling around with data and data analysis and languages like R, Mathematica, Octave, and thinking about how to make books social.

Compressed Representations in the Age of Big Data

Ben Lorica

When developing intelligent, real-time applications, one often has access to a data platform that can wade through and unlock patterns in massive data sets. The backend infrastructure for such applications often relies on distributed, fault-tolerant, scaleout technologies designed to handle large data sets. But, there are situations when compressed representations are useful and even necessary. The rise of mobile computing and sensors (IoT) will lead to devices and software that push computation from the cloud toward the edge. In addition, in-memory computation tends to be much faster, and thus, many popular (distributed) systems operate on data sets that can be cached.

To drive home this point, let me highlight two recent examples that illustrate the importance of efficient compressed representations: one from *mobile* computing, the other from a popular *distributed* computing framework.

Deep Neural Networks and Intelligent Mobile Applications

In a **recent presentation**, Song Han, of the **Concurrent VLSI Architecture** (CVA) group at Stanford University, outlined an initiative to help optimize deep neural networks for mobile devices. Deep learning has produced impressive results across a range of applications in

computer vision, speech, and machine translation. Meanwhile the growing popularity of mobile computing platforms means many mobile applications will need to have capabilities in these areas. The challenge is that deep learning models tend to be too large to fit into mobile applications (these applications are downloaded and often need to be updated frequently). Relying on cloud-based solutions is an option, but *network delay* and *privacy* can be an issue in certain applications and domains.

One solution is to significantly reduce the size of deep learning models. CVA researchers recently proposed a **general scheme for compressing deep neural networks** in three steps (illustrated in Figure 4-1):

- Prune the unimportant connections.
- Quantize the network and enforce weight sharing.
- Apply **Huffman encoding**.

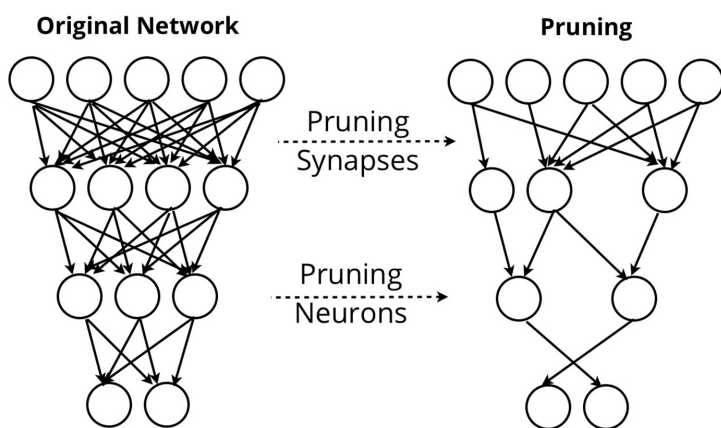


Figure 4-1. Sample diagram comparing compression schemes on neural network sizes (image courtesy of Ben Lorica)

Initial experiments showed their compression scheme reduced neural network sizes by 35 to 50 times, and the resulting compressed models were able to match the accuracy of the corresponding original models. CVA researchers also designed an accompanying energy-efficient **ASIC** accelerator for running compressed deep neural networks, hinting at next-generation software and hardware designed specifically for intelligent mobile applications.

Succinct: Search and Point Queries on Compressed Data Over Apache Spark

Succinct is a “compressed” data store that enables a wide range of point queries (search, count, range, random access) directly on a compressed representation of input data. Succinct uses a compression technique that empirically achieves compression close to that of **gzip**, and supports the above queries without storing secondary indexes, without data scans, and without data decompression. Succinct does not store the input file, just the compressed representation. By letting users query compressed data directly, **Succinct** combines *low latency* and *low storage* (Figure 4-2).

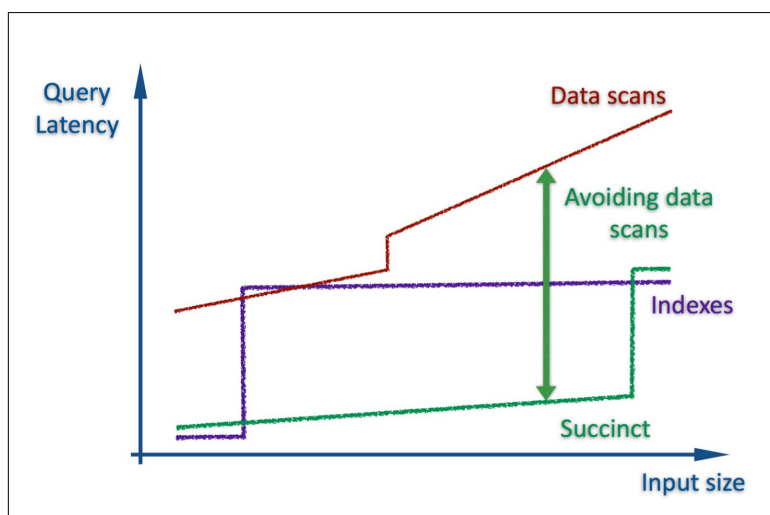


Figure 4-2. Qualitative comparison of data scans, indexes, and Succinct. Since it stores and operates on compressed representations, Succinct can keep data in-memory for much larger-sized input files.

Source: *Rachit Agarwal*, used with permission.

While this **AMPLab** project had been around as a **research initiative**, **Succinct became available on Apache Spark** late last year. This means Spark users can leverage Succinct against flat files and immediately execute *search queries* (including regex queries directly on compressed RDDs), compute counts, and do **range queries**. Moreover, **abstractions have been built on top** of Succinct’s basic flat (unstructured) file interface, allowing Spark to be used as a document or key-value store; and a DataFrames API currently exposes

search, count, range, and random access queries. Having these new capabilities on top of Apache Spark simplifies the software stack needed to build many interesting data applications.

Early **comparisons with Elasticsearch have been promising** and, most importantly for its users, Succinct is an active project. The team behind it plans many enhancements in future releases, including Succinct Graphs (for queries on compressed graphs), support for SQL on compressed data, and further improvements in preprocessing/compression (currently at 4 gigabytes per hour, per core). They are also working on a research project called Succinct Encryption (for queries on compressed *and* encrypted data).

Related Resources

- **Big Data: Efficient Collection and Processing**, Anna Gilbert's Strata + Hadoop World presentation on **compressed sensing**
- **Doing the Impossible (Almost)**, Ted Dunning's Strata + Hadoop World presentation on **t-digest** and approximation algorithms

Ben Lorica

Ben Lorica is the Chief Data Scientist and Director of Content Strategy for Data at O'Reilly Media, Inc. He has applied business intelligence, data mining, machine learning, and statistical analysis in a variety of settings including direct marketing, consumer and market research, targeted advertising, text mining, and financial engineering. His background includes stints with an investment management company, internet startups, and financial services.

Compressing and Regularizing Deep Neural Networks

Song Han

Deep neural networks have evolved to be the state-of-the-art technique for machine learning tasks ranging from computer vision and speech recognition to natural language processing. However, deep learning algorithms are both computationally intensive and memory intensive, making them difficult to deploy on embedded systems with limited hardware resources.

To address this limitation, *deep compression* significantly reduces the computation and storage required by neural networks. For example, for a convolutional neural network with fully connected layers, such as Alexnet and VGGnet, it can reduce the model size by 35×-49×. Even for fully convolutional neural networks such as GoogleNet and SqueezeNet, deep compression can still reduce the model size by 10×. *Both scenarios results in no loss of prediction accuracy.*

Current Training Methods Are Inadequate

Compression without losing accuracy means there's significant redundancy in the trained model, which shows the inadequacy of current training methods. To address this, I've worked with Jeff Pool of NVIDIA, Sharan Narang of Baidu, and Peter Vajda of Facebook to develop the *dense-sparse-dense (DSD) training*, a novel training method that first regularizes the model through sparsity-constrained optimization, and improves the prediction accuracy by

recovering and retraining on pruned weights. At test time, the final model produced by DSD training still has the same architecture and dimension as the original dense model, and DSD training doesn't incur any inference overhead. We experimented with DSD training on mainstream CNN, RNN, and LSTMs for image classification, image caption, and speech recognition and found substantial performance improvements.

In this article, we first introduce deep compression, and then introduce dense-sparse-dense training.

Deep Compression

The first step of deep compression is *synaptic pruning*. The human brain has the process of pruning inherently. Many—possibly a great majority—of the synapses we're born with are **pruned away from infancy to adulthood**.

Does a similar rule apply to artificial neural networks? The answer is yes. In **early work**, network pruning proved to be a valid way to reduce the network complexity and overfitting. This method works on modern neural networks as well. We start by learning the connectivity via normal network training. Next, we prune the small-weight connections: all connections with weights below a threshold are removed from the network. Finally, we retrain the network to learn the final weights for the remaining sparse connections. Pruning reduced the number of parameters by 9× and 13× for AlexNet and the VGG-16 model (see **Figure 5-1**).

The next step of deep compression is *weight sharing*. We found neural networks have really high tolerance to low precision: aggressive approximation of the weight values does not hurt the prediction accuracy. As shown in **Figure 5-2**, the blue weights are originally 2.09, 2.12, 1.92 and 1.87; by letting four of them share the same value, which is 2.00, the accuracy of the network can still be recovered. Thus we can save very few weights, call it “codebook,” and let many other weights share the same weight, storing only the index to the codebook.

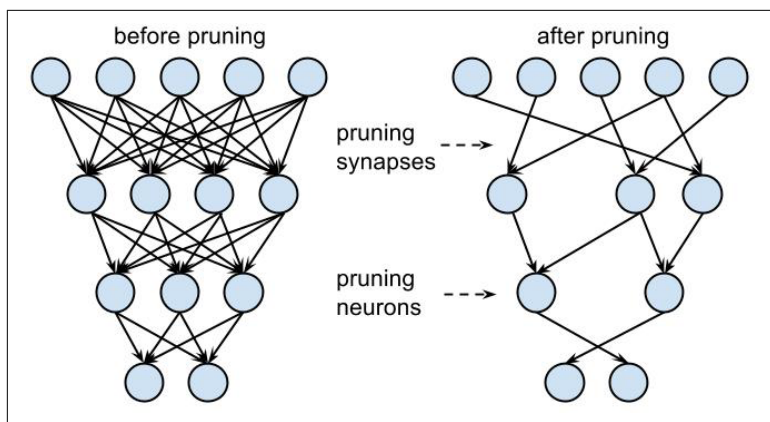


Figure 5-1. Pruning a neural network (all images courtesy of Song Han)

The index could be represented with very few bits. For example, in [Figure 5-2](#), there are four colors; thus, only two bits are needed to represent a weight, as opposed to 32 bits originally. The codebook, on the other side, occupies negligible storage. Our experiments found this kind of weight-sharing technique is better than linear quantization, with respect to the compression ratio and accuracy trade-off.

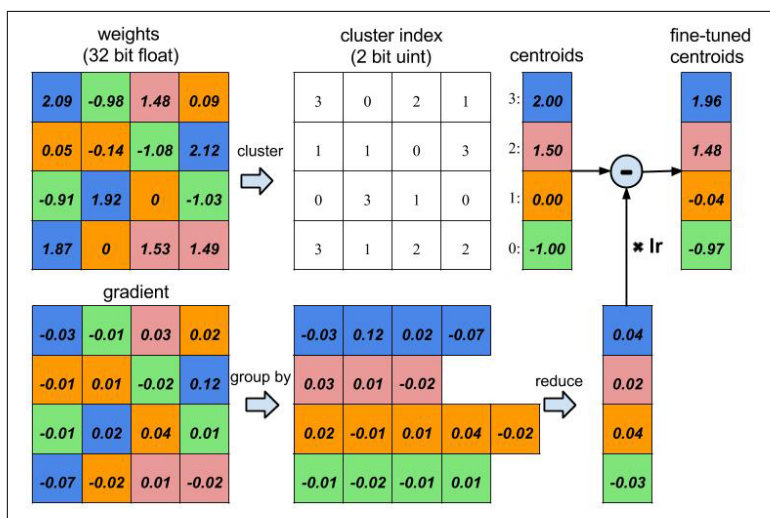


Figure 5-2. Training a weight-sharing neural network

Figure 5-3 shows the overall result of deep compression. Lenet-300-100 and Lenet-5 are evaluated on MNIST data set, while AlexNet, VGGNet, GoogleNet, and SqueezeNet are evaluated on ImageNet data set. The compression ratio ranges from 10× to 49×—even for those fully convolutional neural networks like GoogleNet and SqueezeNet, deep compression can still compress it by an order of magnitude. We highlight SqueezeNet, which has 50× fewer parameters than AlexNet but has the same accuracy, and can still be compressed by 10×, making it only 470 KB. This makes it easy to fit in on-chip SRAM, which is both faster and more energy efficient to access than DRAM.

We have tried other compression methods such as low-rank approximation based methods, but the compression ratio isn't as high. A complete discussion can be found in my research group's [paper on deep compression](#) (see Figure 5-3).

Network	Original Size	Compressed Size	Compression Ratio	Original Accuracy	Compressed Accuracy
Lenet-300-100	1070KB	27KB	40x	98.36%	98.42%
Lenet-5	1720KB	44Kb	39x	99.20%	99.26%
AlexNet	240MB	6.9MB	35x	80.27%	80.30%
VGGNet	550MB	11.3MB	49x	88.68%	89.09%
GoogleNet	28MB	2.8MB	10x	88.90%	88.92%
SqueezeNet	4.8MB	0.47MB	10x	80.32%	80.35%

Figure 5-3. Results of deep compression

DSD Training

The fact that deep neural networks can be aggressively pruned and compressed means that our current training method has some limitation: it can not fully exploit the full capacity of the dense model to find the best local minima, yet a pruned, sparse model that has many fewer synapses can achieve the same accuracy. This brings a question: can we achieve better accuracy by recovering those weights and learn them again?

Let's make an analogy to training for track racing in the Olympics. The coach will first train a runner on high-altitude mountains, where there are a lot of constraints: low oxygen, cold weather, etc. The result is that when the runner returns to the plateau area again, his/her speed is increased. Similar for neural networks, given the

heavily constrained sparse training, the network performs as well as the dense model; once you release the constraint, the model can work better.

Theoretically, the following factors contribute to the effectiveness of DSD training:

Escape Saddle Point

One of the most profound difficulties of optimizing deep networks is the proliferation of **saddle points**. DSD training overcomes saddle points by a pruning and re-densing framework. Pruning the converged model perturbs the learning dynamics and allows the network to jump away from saddle points, which gives the network a chance to converge at a better local or global minimum. This idea is also similar to *simulated annealing*. While Simulated Annealing randomly jumps with decreasing probability on the search graph, DSD deterministically deviates from the converged solution achieved in the first dense training phase by removing the small weights and enforcing a sparsity support.

Regularized and Sparse Training

The sparsity regularization in the sparse training step moves the optimization to a lower-dimensional space where the loss surface is smoother and tends to be more robust to noise. More numerical experiments verified that both sparse training and the final DSD reduce the variance and lead to lower error.

Robust re-initialization

Weight initialization plays a big role in deep learning. Conventional training has only one chance of initialization. DSD gives the optimization a second (or more) chance during the training process to re-initialize from more robust sparse training solutions. We re-dense the network from the sparse solution, which can be seen as a zero initialization for pruned weights. Other initialization methods are also worth trying.

Break Symmetry

The permutation symmetry of the hidden units makes the weights symmetrical, thus prone to co-adaptation in training. In DSD, pruning the weights breaks the symmetry of the hidden units associated with the weights, and the weights are asymmetrical in the final dense phase.

We examined several mainstream CNN, RNN, and LSTM architectures on image classification, image caption, and speech recognition data sets and found that this dense-sparse-dense training flow gives significant accuracy improvement. Our DSD training employs a three-step process: dense, sparse, dense; each step is illustrated in Figure 5-4.

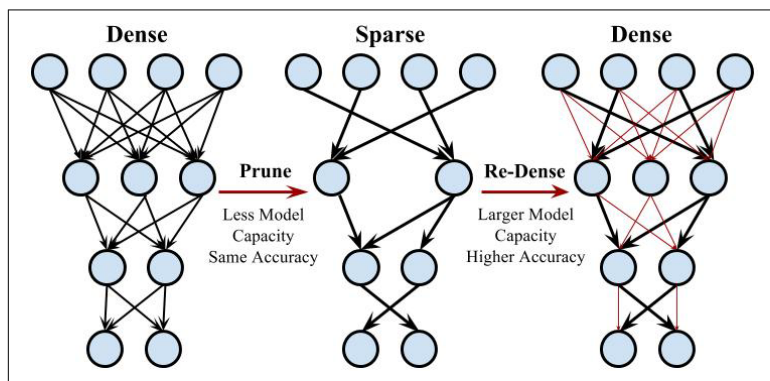


Figure 5-4. Dense-sparse-dense training flow

1. **Initial dense training:** The first D-step learns the connectivity via normal network training on the dense network. Unlike conventional training, however, the goal of this D-step is not to learn the final values of the weights; rather, we are learning which connections are important.
2. **Sparse training:** The S-step prunes the low-weight connections and retrain the sparse network. We applied the same sparsity to all the layers in our experiments, thus there's a *single* hyperparameter: the sparsity. For each layer we sort the parameters, the smallest $N \times \text{sparsity}$ parameters are removed from the network, converting a dense network into a sparse network. We found that a sparsity ratio of 50%-70% works very well. Then, we retrain the sparse network, which can fully recover the model accuracy under the sparsity constraint.
3. **Final dense training:** The final D-step recovers the pruned connections, making the network dense again. These previously pruned connections are initialized to zero and retrained. Restoring the pruned connections increases the dimensionality of the network, and more parameters make it easier for the net-

work to slide down the saddle point to arrive at a better local minima.

We applied DSD training to different kinds of neural networks on data sets from different domains. We found that DSD training improved the accuracy for all these networks compared to neural networks that were not trained with DSD. The neural networks are chosen from CNN, RNN, and LSTMs; the data sets are chosen from image classification, speech recognition, and caption generation. The results are shown in [Figure 5-5](#). DSD models are available to download at [DSD Model Zoo](#).

Baseline	Top-1 error	Top-5 error	DSD	Top-1 error	Top-5 error
AlexNet	42.78%	19.73%	AlexNet_DSD	41.48%	18.71%
VGG16	31.50%	11.32%	VGG16_DSD	27.19%	8.67%
GoogleNet	31.14%	10.96%	GoogleNet_DSD	30.02%	10.34%
SqueezeNet	42.39%	19.32%	SqueezeNet_DSD	38.24%	16.53%
ResNet18	30.43%	10.76%	ResNet18_DSD	29.17%	10.13%
ResNet50	24.01%	7.02%	ResNet50_DSD	22.89%	6.47%

Figure 5-5. DSD training improves prediction accuracy

Generating Image Descriptions

We visualized the effect of DSD training on an image caption task (see [Figure 5-6](#)). We applied DSD to [NeuralTalk](#), an LSTM for generating image descriptions. The baseline model fails to describe images 1, 4, and 5. For example, in the first image, the baseline model mistakes the girl for a boy, and mistakes the girl’s hair for a rock wall; the sparse model can tell that it’s a girl in the image, and the DSD model can further identify the swing.

In the the second image, DSD training can tell the player is trying to make a shot, rather than the baseline, which just says he’s playing with a ball. It’s interesting to notice that the sparse model sometimes works better than the DSD model. In the last image, the sparse model correctly captured the mud puddle, while the DSD model only captured the forest from the background. The good performance of DSD training generalizes beyond these examples, and more image caption results generated by DSD training are provided in the appendix of this [paper](#).

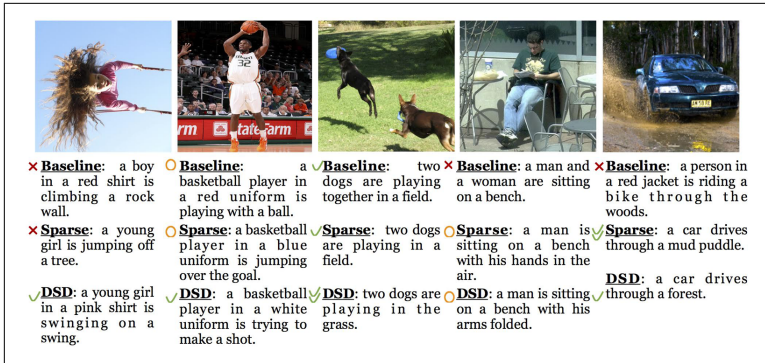


Figure 5-6. Visualization of DSD training improves the performance of image captioning

Advantages of Sparsity

Deep compression, for compressing deep neural networks for smaller model size, and DSD training for regularizing neural networks, are both techniques that utilize sparsity and achieve a smaller size or higher prediction accuracy. Apart from model size and prediction accuracy, we looked at two other dimensions that take advantage of sparsity: speed and energy efficiency, which are beyond the scope of this article. Readers can refer to [EIE](#) for further references.

Song Han

Song Han is a fifth year PhD student with [Professor Bill Dally](#) at [Stanford University](#). His research focuses on energy-efficient deep learning, at the intersection between machine learning and computer architecture. Song proposed [deep compression](#) that can compress state-of-the art CNNs by 10×–49× and compressed [SqueezeNet](#) to only 470 KB, which fits fully in on-chip SRAM. He proposed a [DSD](#) training flow that improved that accuracy of a wide range of neural networks. He designed [EIE: Efficient Inference Engine](#), a hardware architecture that does inference directly on the compressed sparse neural network model, which is 13× faster and 3,000× energy efficient than GPU. His work has been covered by [The Next Platform](#), [TechEmergence](#), [Embedded Vision](#), and [O'Reilly](#). His work received the [Best Paper Award](#) in ICLR'16.

Reinforcement Learning Explained

Junling Hu

A robot takes a big step forward, then falls. The next time, it takes a smaller step and is able to hold its balance. The robot tries variations like this many times; eventually, it learns the right size of steps to take and walks steadily. It has succeeded.

What we see here is called reinforcement learning. It directly connects a robot's action with an outcome, without the robot having to learn a complex relationship between its action and results. The robot learns how to walk based on reward (staying on balance) and punishment (falling). This feedback is considered "reinforcement" for doing or not doing an action.

Another example of reinforcement learning can be found when playing the game Go. If the computer player puts down its white piece at a location, then gets surrounded by the black pieces and loses that space, it is punished for taking such a move. After being beaten a few times, the computer player will avoid putting the white piece in that location when black pieces are around.

Reinforcement learning, in a simplistic definition, is learning best actions based on reward or punishment.

There are three basic concepts in reinforcement learning: state, action, and reward. The state describes the current situation. For a robot that is learning to walk, the state is the position of its two legs. For a Go program, the state is the positions of all the pieces on the board.

Action is what an agent can do in each state. Given the state, or positions of its two legs, a robot can take steps within a certain distance. There are typically finite (or a fixed range of) actions an agent can take. For example, a robot stride can only be, say, 0.01 meter to 1 meter. The Go program can only put down its piece in one of 19 x 19 (that is 361) positions.

When a robot takes an action in a state, it receives a reward. Here the term “reward” is an abstract concept that describes feedback from the environment. A reward can be positive or negative. When the reward is positive, it is corresponding to our normal meaning of reward. When the reward is negative, it is corresponding to what we usually call “punishment.”

Each of these concepts seems simple and straightforward: once we know the state, we choose an action that (hopefully) leads to positive reward. But the reality is more complex.

Consider this example: a robot learns to go through a maze. When the robot takes one step to the right, it reaches an open location; but when it takes one step to the left, it also reaches an open location. After going left for three steps, the robot hits a wall. Looking back, taking the left step at location 1 is a bad idea (bad action). How would the robot use the reward at each location (state) to learn how to get through the maze (which is the ultimate goal)?

“Real” reinforcement learning, or the version used as a machine learning method today, concerns itself with the long-term reward, not just the immediate reward.

The long-term reward is learned when an agent interacts with an environment through many trials and errors. The robot that is running through the maze remembers every wall it hits. In the end, it remembers the previous actions that lead to dead ends. It also remembers the path (that is, a sequence of actions) that leads it successfully through the maze. The essential goal of reinforcement learning is learning a sequence of actions that lead to a long-term reward. An agent learns that sequence by interacting with the environment and observing the rewards in every state.

How does an agent know what the desired long-term payoff should be? The secret lies in a Q-table (or Q function). It’s a lookup table for rewards associated with every state-action pair. Each cell in this table records a value called a Q-value. It is a representation of the

long-term reward an agent would receive when taking this action at this particular state, followed by taking the best path possible afterward.

How does the agent learn about this long-term Q-value reward? It turns out, the agent does not need to solve a complex mathematical function. There is a simple procedure to learn all the Q-values called Q-learning. Reinforcement learning is essentially learning about Q-values while taking actions.

Q-Learning: A Commonly Used Reinforcement Learning Method

Q-learning is the most commonly used reinforcement learning method, where Q stands for the long-term value of an action. Q-learning is about learning Q-values through observations.

The procedure for Q-learning is:

1. In the beginning, the agent initializes Q-values to 0 for every state-action pair. More precisely, $Q(s,a) = 0$ for all states s and actions a . This is essentially saying we have no information on long-term reward for each state-action pair.
2. After the agent starts learning, it takes an action a in state s and receives reward r . It also observes that the state has changed to a new state s' . The agent will update $Q(s,a)$ with this formula:

$$Q(s,a) = (1 - \text{learning_rate})Q(s,a) + \text{learning_rate} (r + \text{discount_rate} \max_a Q(s',a))$$

The learning rate is a number between 0 and 1. It is a weight given to the new information versus the old information. The new long-term reward is the current reward, r , plus all future rewards in the next state, s' , and later states, assuming this agent always takes its best actions in the future. The future rewards are discounted by a discount rate between 0 and 1, meaning future rewards are not as valuable as the reward now.

In this updating method, Q carries memory from the past and takes into account all future steps. Note that we use the maximized Q-value for the new state, assuming we always follow the optimal path afterward. As the agent visits all the states and tries different actions,

it eventually learns the optimal Q-values for all possible state-action pairs. Then it can derive the action in every state that is optimal for the long term.

A simple example can be seen with the maze robot in [Figure 6-1](#).

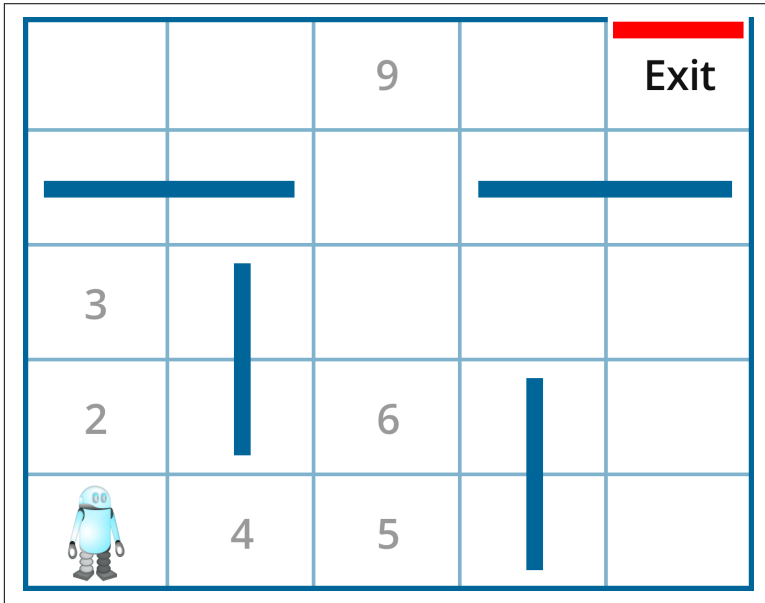


Figure 6-1. Maze robot

The robot starts from the lower-left corner of the maze. Each location (state) is indicated by a number. There are four action choices (left, right, up, down), but in certain states, action choices are limited. For example, in state 1 (initial state), the robot has only two action choices: up or right. In state 4, it has three action choices: left, right, or up. When the robot hits a wall, it receives reward -1. When it reaches an open location, it receives reward 0. When it reaches the exit, it receives reward 100. However, note that this one-time reward is very different from Q-values. In fact, we have the following, where the learning rate is 0.2 and the discount rate is 0.9:

$$Q(4, \text{left}) = 0.8 \times 0 + 0.2 (0 + 0.9 Q(1, \text{right}))$$

and

$$Q(4, \text{right}) = 0.8 \times 0 + 0.2 (0 + 0.9 Q(5, \text{up}))$$

The best action in state 1 is right, and the best action in state 5 is up. $Q(1, \text{right})$ and $Q(5, \text{up})$ have different values because it takes more steps from state 1 than state 5 to reach the exit. Since we discount future rewards, we discount the added steps to reach the goal. Thus $Q(5, \text{up})$ has a higher value than $Q(1, \text{right})$. For this reason, $Q(4, \text{right})$ has a higher value than $Q(4, \text{left})$. Thus, the best action in state 4 is going right.

Q-learning requires the agent try many times to visit all possible state-action pairs. Only then does the agent have a complete picture of the world. Q-values represent the optimal values when taking the best sequence of actions. This sequence of actions is also called “policy.”

A fundamental question we face is: is it possible for an agent to learn all the Q-values when it explores the actions possible in a given environment? In other words, is such learning feasible? The answer is yes if we assume the world responds to actions. In other words, the state changes based on an action. This assumption is called the *Markov Decision Process (MDP)*. It assumes that the next state depends on the previous state and the action taken. Based on this assumption, all possible states can eventually be visited, and the long-term value (Q-value) of every state-action pair can be determined.

Imagine that we live in a random universe where our actions have no impact on what happens next: reinforcement learning (Q-learning) would break down. After many times of trying, we'd have to throw in the towel. Fortunately, our universe is much more predictable. When a Go player puts down a piece on the board, the board position is clear in the next round. Our agent interacts with the environment and shapes it through its actions. The exact impact of our agent's action on the state is typically straightforward. The new state is immediately observable. The robot can tell where it ends up.

Common Techniques of Reinforcement Learning

The essential technique of reinforcement learning is exploration versus exploitation. An agent learns about the value of $Q(s, a)$ in state s for every action a . Since the agent needs to get a high reward, it

can choose the action that leads to the highest reward based on current information (exploitation), or keep trying new actions, hoping it brings even higher reward (exploration). When an agent is learning online (in real time), the balance of these two strategies is very important. That's because learning in real time means the agent has to maintain its own survival (when exploring a cave or fighting in a combat) versus finding the best move. It is less of a concern when an agent is learning offline (meaning not in real time). In machine learning terms, offline learning means an agent processes information without interacting with the world. In such cases, the price to pay for failing (like hitting a wall, or being defeated in a game) is little when it can experiment with (explore) many different actions without worrying about the consequences.

The performance of Q-learning depends on visiting all state-action pairs in order to learn the correct Q-values. This can be easily achieved with a small number of states. In the real world, however, the number of states can be very large, particularly when there are multiple agents in the system. For example, in a maze game, a robot has at most 1,000 states (locations); this grows to 1,000,000 when it is in a game against another robot, where the state represents the joint location of two robots (1,000 x 1,000).

When the state space is large, it is not efficient to wait until we visit all state-actions pairs. A faster way to learn is called the **Monte Carlo method**. In statistics, the Monte Carlo method derives an average through repeated sampling. In reinforcement learning, the Monte Carlo method is used to derive Q-values after repeatedly seeing the same state-action pair. It sets the Q-value, $Q(s,a)$, as the average reward after many visits to the same state-action pair (s, a) . This method removes the need for using a learning rate or a discount rate. It depends only on large numbers of simulations. Due to its simplicity, this method has become very popular. It has been used by AlphaGo after playing many games against itself to learn about the best moves.

Another way to reduce the number of states is by using a neural network, where the inputs are states and outputs are actions, or Q-values associated with each action. A deep neural network has the power to dramatically simplify the representation of states through hidden layers. In **this Nature paper on deep reinforcement learning used with Atari games**, the whole game board is mapped by a convolutional neural network to determine Q-values.

What Is Reinforcement Learning Good For?

Reinforcement learning is good for situations where information about the world is very limited: there is no given map of the world. We have to learn our actions by interacting with the environment: trial and error is required. For example, a Go program cannot calculate all possible future states, which could be 10^{170} , while the universe is only 10^{17} seconds old. This means even if the computer can compute one billion (10^9) possible game boards (states) in a second, it will take longer than the age of the universe to finish that calculation.

Since we cannot enumerate all possible situations (and optimize our actions accordingly), we have to learn through the process of taking actions. Once an action is taken, we can immediately observe the results and change our action the next time.

Recent Applications

Reinforcement learning historically was mostly applied to robot control and simple board games, such as backgammon. Recently, it achieved a lot of momentum by combining with deep learning, which simplifies the states (when the number of states is very large). Current applications of reinforcement learning include:

Playing the board game Go

The most successful example is AlphaGo, a computer program that won against the second best human player in the world. **AlphaGo uses reinforcement learning** to learn about its next move based on current board position. The board position is simplified through a convolutional neural network, which then produces actions as outputs.

Computer games

Most recently, playing Atari Games.

Robot control

Robots can learn to walk, run, dance, fly, play ping-pong or stack Legos with reinforcement learning.

Online advertising

A computer program can use reinforcement learning to select an ad to show a user at the right time, or in the right format.

Dialogue generation

A conversational agent selects a sentence to say based on a forward-looking, long-term reward. This makes the dialogue more engaging and longer lasting. For example, instead of saying, “I am 16” in response to the question, “How old are you?” the program can say, “I am 16. Why are you asking?”

Getting Started with Reinforcement Learning

OpenAI provides a reinforcement learning benchmarking toolkit called **OpenAI Gym**. It has sample code and can help a beginner to get started. The **CartPole problem** is a fun problem to start with, where many learners have submitted their code and documented their approach.

The success of AlphaGo and its use of reinforcement learning has prompted interest in this method. Combined with deep learning, reinforcement learning has become a powerful tool for many applications. The time of reinforcement learning has come!

Junling Hu

Junling Hu is a leading expert in artificial intelligence and data science, and chair for the AI Frontiers Conference. She was director of data mining at Samsung, where she led an end-to-end implementation of data mining solutions for large-scale and real-time data products. Prior to Samsung, Dr. Hu led a data science team at PayPal and eBay, building predictive models in marketing, sales, and customer churn prediction. Before joining eBay, Dr. Hu managed a data mining group at Bosch research. Dr. Hu was an assistant professor at the University of Rochester from 2000 to 2003.

Dr. Hu has more than 1,000 scholarly citations on her papers. She is a recipient of the Sloan Fellowship for Distinguished Women in Engineering, and a recipient of the prestigious CAREER award from the National Science Foundation (NSF). She served twice on NSF review panels for funding proposals from US institutes. Dr. Hu received her PhD in computer science from the University of Michigan in Ann Arbor in 1999, with her focus area in AI, particularly in reinforcement learning.

Hello, TensorFlow!

Aaron Schumacher

The **TensorFlow** project is bigger than you might realize. The fact that it's a library for deep learning and its connection to Google have helped TensorFlow attract a lot of attention. But beyond the hype, there are unique elements to the project that are worthy of closer inspection:

- The core library is suited to a broad family of machine learning techniques, not “just” deep learning.
- Linear algebra and other internals are prominently exposed.
- In addition to the core machine learning functionality, TensorFlow also includes its own logging system, its own interactive log visualizer, and even its own heavily engineered serving architecture.
- The execution model for TensorFlow differs from Python's scikit-learn, or most tools in R.

Cool stuff, but—especially for someone hoping to explore machine learning for the first time—TensorFlow can be a lot to take in.

How does TensorFlow work? Let's break it down so we can see and understand every moving part. We'll explore the data flow **graph** that defines the computations your data will undergo, how to train models with **gradient descent** using TensorFlow, and how **TensorBoard** can visualize your TensorFlow work. The examples here won't solve industrial machine learning problems, but they'll help you understand the components underlying everything built with TensorFlow, including whatever you build next!

Names and Execution in Python and TensorFlow

The way TensorFlow manages computation is not totally different from the way Python usually does. With both, it's important to remember, to paraphrase **Hadley Wickham**, that an object has no name (see **Figure 7-1**). In order to see the similarities (and differences) between how Python and TensorFlow work, let's look at how they refer to objects and handle evaluation.

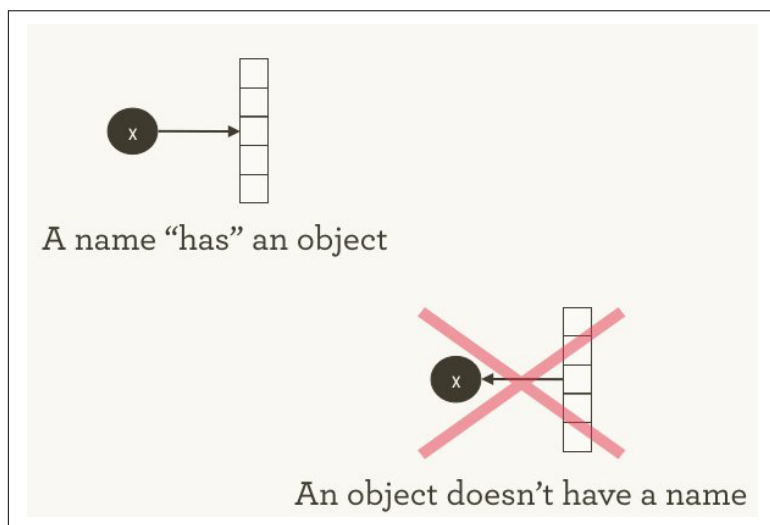


Figure 7-1. Names “have” objects, rather than the reverse (image courtesy of Hadley Wickham, used with permission)

The variable names in Python code aren't what they represent; they're just pointing at objects. So, when you say in Python that `foo = []` and `bar = foo`, it isn't just that `foo` equals `bar`; `foo` *is* `bar`, in the sense that they both point at the same list object:

```
>>> foo = []
>>> bar = foo
>>> foo == bar
## True
>>> foo is bar
## True
```

You can also see that `id(foo)` and `id(bar)` are the same. This identity, especially with **mutable** data structures like lists, can lead to surprising bugs when it's misunderstood.

Internally, Python manages all your objects and keeps track of your variable names and which objects they refer to. The TensorFlow graph represents another layer of this kind of management; as we'll see, Python names will refer to objects that connect to more granular and managed TensorFlow graph operations.

When you enter a Python expression, for example at an interactive interpreter or Read-Evaluate-Print Loop (REPL), whatever is read is almost always evaluated right away. Python is eager to do what you tell it. So, if I tell Python to `foo.append(bar)`, it appends right away, even if I never use `foo` again.

A lazier alternative would be to just remember that I said `foo.append(bar)`, and if I ever evaluate `foo` at some point in the future, Python could do the append then. This would be closer to how TensorFlow behaves, where defining relationships is entirely separate from evaluating what the results are.

TensorFlow separates the definition of computations from their execution even further by having them happen in separate places: a graph defines the operations, but the operations only happen within a session. Graphs and sessions are created independently. A graph is like a blueprint, and a session is like a construction site.

Back to our plain Python example, recall that `foo` and `bar` refer to the same list. By appending `bar` into `foo`, we've put a list inside itself. You could think of this structure as a graph with one node, pointing to itself. Nesting lists is one way to represent a graph structure like a TensorFlow computation graph:

```
>>> foo.append(bar)
>>> foo
## [[...]]
```

Real TensorFlow graphs will be more interesting than this!

The Simplest TensorFlow Graph

To start getting our hands dirty, let's create the simplest TensorFlow graph we can, from the ground up. TensorFlow is admirably easier to **install** than some other frameworks. The examples here work with either Python 2.7 or 3.3+, and the TensorFlow version used is 0.8:

```
>>> import tensorflow as tf
```

At this point TensorFlow has already started managing a lot of state for us. There's already an implicit default graph, for example. **Internally**, the default graph lives in the `_default_graph_stack`, but we don't have access to that directly. We use `tf.get_default_graph()`:

```
>>> graph = tf.get_default_graph()
```

The nodes of the TensorFlow graph are called “operations,” or “ops.” We can see what operations are in the graph with `graph.get_operations()`:

```
>>> graph.get_operations()
## []
```

Currently, there isn't anything in the graph. We'll need to put everything we want TensorFlow to compute into that graph. Let's start with a simple constant input value of one:

```
>>> input_value = tf.constant(1.0)
```

That constant now lives as a node, an operation, in the graph. The Python variable name `input_value` refers indirectly to that operation, but we can also find the operation in the default graph:

```
>>> operations = graph.get_operations()
>>> operations
## [<tensorflow.python.framework.ops.Operation at 0x1185005d0>]
>>> operations[0].node_def
## name: "Const"
## op: "Const"
## attr {
##   key: "dtype"
##   value {
##     type: DT_FLOAT
##   }
## }
## attr {
##   key: "value"
##   value {
##     tensor {
##       dtype: DT_FLOAT
##       tensor_shape {
##       }
##       float_val: 1.0
##     }
##   }
## }
```

TensorFlow uses protocol buffers internally. (**Protocol buffers** are sort of like a Google-strength **JSON**.) Printing the `node_def` for the

constant operation above shows what's in TensorFlow's protocol buffer representation for the number one.

People new to TensorFlow sometimes wonder why there's all this fuss about making “TensorFlow versions” of things. Why can't we just use a normal Python variable without also defining a TensorFlow object? [One of the TensorFlow tutorials](#) has an explanation:

To do efficient numerical computing in Python, we typically use libraries like NumPy that do expensive operations such as matrix multiplication outside Python, using highly efficient code implemented in another language. Unfortunately, there can still be a lot of overhead from switching back to Python every operation. This overhead is especially bad if you want to run computations on GPUs or in a distributed manner, where there can be a high cost to transferring data.

TensorFlow also does its heavy lifting outside Python, but it takes things a step further to avoid this overhead. Instead of running a single expensive operation independently from Python, TensorFlow lets us describe a graph of interacting operations that run entirely outside Python. This approach is similar to that used in Theano or Torch.

TensorFlow can do a lot of great things, but it can only work with what's been explicitly given to it. This is true even for a single constant.

If we inspect our `input_value`, we see it is a constant 32-bit float tensor of no dimension: just one number:

```
>>> input_value
## <tf.Tensor 'Const:0' shape=() dtype=float32>
```

Note that this *doesn't* tell us what that number *is*. To evaluate `input_value` and get a numerical value out, we need to create a “session” where graph operations can be evaluated and then explicitly ask to evaluate or “run” `input_value`. (The session picks up the default graph by default.)

```
>>> sess = tf.Session()
>>> sess.run(input_value)
## 1.0
```

It may feel a little strange to “run” a constant. But it isn't so different from evaluating an expression as usual in Python; it's just that TensorFlow is managing its own space of things—the computational graph—and it has its own method of evaluation.

The Simplest TensorFlow Neuron

Now that we have a session with a simple graph, let's build a neuron with just one parameter, or weight. Often, even simple neurons also have a bias term and a nonidentity activation function, but we'll leave these out.

The neuron's weight isn't going to be constant; we expect it to change in order to learn based on the “true” input and output we use for training. The weight will be a TensorFlow **variable**. We'll give that variable a starting value of 0.8:

```
>>> weight = tf.Variable(0.8)
```

You might expect that adding a variable would add one operation to the graph, but in fact that one line adds four operations. We can check all the operation names:

```
>>> for op in graph.get_operations(): print(op.name)
## Const
## Variable/initial_value
## Variable
## Variable/Assign
## Variable/read
```

We won't want to follow every operation individually for long, but it will be nice to see at least one that feels like a real computation:

```
>>> output_value = weight * input_value
```

Now there are six operations in the graph, and the last one is that multiplication:

```
>>> op = graph.get_operations()[-1]
>>> op.name
## 'mul'
>>> for op_input in op.inputs: print(op_input)
## Tensor("Variable/read:0", shape=(), dtype=float32)
## Tensor("Const:0", shape=(), dtype=float32)
```

This shows how the multiplication operation tracks where its inputs come from: they come from other operations in the graph. To understand a whole graph, following references this way quickly becomes tedious for humans. **TensorBoard graph visualization** is designed to help.

How do we find out what the product is? We have to “run” the `output_value` operation. But that operation depends on a variable: `weight`. We told TensorFlow that the initial value of `weight` should

be 0.8, but the value hasn't yet been set in the current session. The `tf.initialize_all_variables()` function generates an operation which will initialize all our variables (in this case just one). Then we can run that operation:

```
>>> init = tf.initialize_all_variables()
>>> sess.run(init)
```

The result of `tf.initialize_all_variables()` will include initializers for all the variables *currently in the graph*. So if you add more variables, you'll want to use `tf.initialize_all_variables()` again; a stale `init` wouldn't include the new variables.

Now we're ready to run the `output_value` operation:

```
>>> sess.run(output_value)
## 0.80000001
```

Recall that's $0.8 * 1.0$ with 32-bit floats, and 32-bit floats **have a hard time** with 0.8; 0.80000001 is as close as they can get.

See Your Graph in TensorBoard

Up to this point, the graph has been simple, but it would already be nice to see it represented in a diagram. We'll use TensorBoard to generate that diagram. TensorBoard reads the name field that is stored inside each operation (quite distinct from Python variable names). We can use these TensorFlow names and switch to more conventional Python variable names. Using `tf.mul` here is equivalent to our earlier use of just `*` for multiplication, but it lets us set the name for the operation:

```
>>> x = tf.constant(1.0, name='input')
>>> w = tf.Variable(0.8, name='weight')
>>> y = tf.mul(w, x, name='output')
```

TensorBoard works by looking at a directory of output created from TensorFlow sessions. We can write this output with a `SummaryWriter`, and if we do nothing aside from creating one with a graph, it will just write out that graph.

The first argument when creating the `SummaryWriter` is an output directory name, which will be created if it doesn't exist.

```
>>> summary_writer = tf.train.SummaryWriter('log_simple_graph',
sess.graph)
```

Now, at the command line, we can start up TensorBoard:

```
$ tensorboard --logdir=log_simple_graph
```

TensorBoard runs as a local web app, on port 6006. (“6006” is “goog” upside-down.) If you go in a browser to `localhost:6006/#graphs` you should see a diagram of the graph you created in TensorFlow, which looks something like [Figure 7-2](#).



Figure 7-2. A TensorBoard visualization of the simplest TensorFlow neuron

Making the Neuron Learn

Now that we’ve built our neuron, how does it learn? We set up an input value of 1.0. Let’s say the correct output value is zero. That is, we have a very simple “training set” of just one example with one feature, which has the value one, and one label, which is zero. We want the neuron to learn the function taking one to zero.

Currently, the system takes the input one and returns 0.8, which is not correct. We need a way to measure how wrong the system is. We’ll call that measure of wrongness the “loss” and give our system the goal of minimizing the loss. If the loss can be negative, then minimizing it could be silly, so let’s make the loss the square of the difference between the current output and the desired output:

```
>>> y_ = tf.constant(0.0)
>>> loss = (y - y_)**2
```

So far, nothing in the graph does any learning. For that, we need an optimizer. We’ll use a gradient descent optimizer so that we can update the weight based on the derivative of the loss. The optimizer takes a learning rate to moderate the size of the updates, which we’ll set at 0.025:

```
>>> optim = tf.train.GradientDescentOptimizer(learning_rate=
0.025)
```

The optimizer is remarkably clever. It can automatically work out and apply the appropriate gradients through a whole network, carrying out the backward step for learning.

Let's see what the gradient looks like for our simple example:

```
>>> grads_and_vars = optim.compute_gradients(loss)
>>> sess.run(tf.initialize_all_variables())
>>> sess.run(grads_and_vars[1][0])
## 1.6
```

Why is the value of the gradient 1.6? Our loss is error squared, and the derivative of that is two times the error. Currently the system says 0.8 instead of 0, so the error is 0.8, and two times 0.8 is 1.6. It's working!

For more complex systems, it will be very nice indeed that TensorFlow calculates and then applies these gradients for us automatically.

Let's apply the gradient, finishing the backpropagation:

```
>>> sess.run(optim.apply_gradients(grads_and_vars))
>>> sess.run(w)
## 0.75999999 # about 0.76
```

The weight decreased by 0.04 because the optimizer subtracted the gradient times the learning rate, $1.6 * 0.025$, pushing the weight in the right direction.

Instead of hand-holding the optimizer like this, we can make one operation that calculates and applies the gradients: the `train_step`:

```
>>> train_step = tf.train.GradientDescentOptimizer(0.025).
minimize(loss)
>>> for i in range(100):
>>>     sess.run(train_step)
>>>
>>> sess.run(y)
## 0.0044996012
```

Running the training step many times, the weight and the output value are now very close to zero. The neuron has learned!

Training Diagnostics in TensorBoard

We may be interested in what's happening during training. Say we want to follow what our system is predicting at every training step. We could print from inside the training loop:

```
>>> sess.run(tf.initialize_all_variables())
>>> for i in range(100):
>>>     print('before step {}, y is {}'.format(i, sess.run(y)))
>>>     sess.run(train_step)
>>>
## before step 0, y is 0.800000011921
## before step 1, y is 0.759999990463
## ...
## before step 98, y is 0.00524811353534
## before step 99, y is 0.00498570781201
```

This works, but there are some problems. It's hard to understand a list of numbers. A plot would be better. And even with only one value to monitor, there's too much output to read. We're likely to want to monitor many things. It would be nice to record everything in some organized way.

Luckily, the same system that we used earlier to visualize the graph also has just the mechanisms we need.

We instrument the computation graph by adding operations that summarize its state. Here, we'll create an operation that reports the current value of *y*, the neuron's current output:

```
>>> summary_y = tf.scalar_summary('output', y)
```

When you run a summary operation, it returns a string of protocol buffer text that can be written to a log directory with a `SummaryWriter`:

```
>>> summary_writer = tf.train.SummaryWriter('log_simple_stats')
>>> sess.run(tf.initialize_all_variables())
>>> for i in range(100):
>>>     summary_str = sess.run(summary_y)
>>>     summary_writer.add_summary(summary_str, i)
>>>     sess.run(train_step)
>>>
```

Now after running `tensorboard --logdir=log_simple_stats`, you get an interactive plot at `localhost:6006/#events` (Figure 7-3).

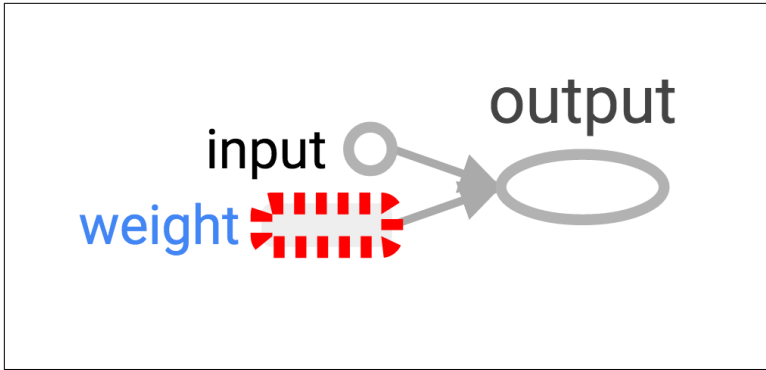


Figure 7-3. A TensorBoard visualization of a neuron's output against its training iteration number

Flowing Onward

Here's a final version of the code. It's fairly minimal, with every part showing useful (and understandable) TensorFlow functionality:

```
import tensorflow as tf

x = tf.constant(1.0, name='input')
w = tf.Variable(0.8, name='weight')
y = tf.mul(w, x, name='output')
y_ = tf.constant(0.0, name='correct_value')
loss = tf.pow(y - y_, 2, name='loss')
train_step = tf.train.GradientDescentOptimizer(0.025).minimize(
    loss)

for value in [x, w, y, y_, loss]:
    tf.scalar_summary(value.op.name, value)

summaries = tf.merge_all_summaries()

sess = tf.Session()
summary_writer = tf.train.SummaryWriter('log_simple_stats',
    sess.graph)

sess.run(tf.initialize_all_variables())
for i in range(100):
    summary_writer.add_summary(sess.run(summaries), i)
    sess.run(train_step)
```

The example we just ran through is even simpler than the ones that inspired it in Michael Nielsen's *Neural Networks and Deep Learning* (Determination Press). For myself, seeing details like these helps

with understanding and building more complex systems that use and extend from simple building blocks. Part of the beauty of TensorFlow is how flexibly you can build complex systems from simpler components.

If you want to continue experimenting with TensorFlow, it might be fun to start making more interesting neurons, perhaps with different **activation functions**. You could train with more interesting data. You could add more neurons. You could add more layers. You could dive into more complex **prebuilt models**, or spend more time with TensorFlow's own **tutorials** and **how-to guides**. Go for it!

Aaron Schumacher

Aaron Schumacher is a data scientist and software engineer for Deep Learning Analytics. He has taught with Python and R for General Assembly and the Metis data science bootcamp. Aaron has also worked with data at Booz Allen Hamilton, New York University, and the New York City Department of Education. Aaron's career-best breakdancing result was advancing to the semifinals of the R16 Korea 2009 individual footwork battle. He is honored to now be the least significant contributor to TensorFlow 0.9.

Dive into TensorFlow with Linux

Justin Francis

For the last eight months, I have spent a lot of time trying to absorb as much as I can about machine learning. I am constantly amazed at the variety of people I meet on online MOOCs in this small but quickly growing community, from quantum researchers at [Fermilab](#) to Tesla-driving Silicon Valley CEOs. Lately, I have been putting a lot of my focus into the open source software TensorFlow, and this tutorial is the result of that.

I feel like a lot of machine learning tutorials are geared toward Mac. One major advantage of using Linux is it's free and it supports using TensorFlow with your GPU. The accelerated parallel computing power of GPUs is one of the reasons for such major advancements in machine learning. You don't need cutting-edge technology to build a fast image classifier; my computer and graphic card cost me less than \$400 USD.

In this tutorial, I am going to walk you through how I learned to train my own image classifier on Ubuntu with a GPU. This tutorial is very similar to Pete Warden's "[TensorFlow for Poets](#)", but I did things a little differently. I am going to assume that you have TensorFlow and [Bazel](#) installed and have [cloned](#) the latest TensorFlow release in your home directory. If you have not yet done that, you can follow a tutorial on my [blog](#). If you don't have a compatible GPU, you can still follow this tutorial; it will just take longer.

The overall process is extremely simple and can be broken into four main steps:

1. Collecting the training images
2. Training a graph/model using TensorFlow and the Inception model
3. Building the script that will classify new images using your graph
4. Testing the script by classifying new images

I decided to train my classifier on five birds of prey. Using birds of prey wasn't a random decision. For two years, I worked for **The Raptors**, an educational center and wildlife management organization in Duncan, British Columbia, and I've had a long-standing passion for these powerful and mythical creatures. As the ultimate challenge, I pitted my classifier against **Cornell Ornithology Lab's Merlin ID tool**. Since writing this article, the lab has updated its site with this note: "Merlin Photo ID is temporarily down for maintenance and upgrades. ... The Cornell Lab of Ornithology and Visipedia are collaborating to develop computer vision technology to identify birds in photos." Without a doubt, I suspect that they are upgrading their (now unavailable) Merlin system to a modern machine learning classifier.

Collecting Training Images

I began by gathering about 100 photos of each bird from The Raptors' **Facebook** page and from web searches. I was able to build a set of photos with a good mix of the birds in many different environments and positions. It is ideal for an image classifier to have at least 100 photos in varying scenarios and backgrounds in order for it to generalize well. There are also ways to **distort** existing photos to make more training examples, but this can slow your training process to a crawl. Keep in mind that we don't need thousands of examples to train our model, because TensorFlow will retrain a new model using trained feature detectors from the previously trained **Inception model**.

As an additional experiment, about 10% of the pictures I used were of juvenile birds of each species. I was curious if the classifier could find the similarities between a juvenile bird and a mature one.

Once I had the right number and types of images, I created a folder in my TensorFlow directory:

```
$ cd ~/tensorflow

$ mkdir tf_files && cd tf_files && mkdir bird_photos &&
cd bird_photos

$ mkdir baldeagle goldeneagle peregrine saker vulture
```

Figure 8-1 is what my directory looked like.

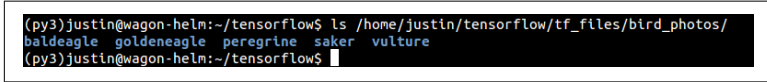
A terminal window showing the command 'ls /home/justin/tensorflow/tf_files/bird_photos/' and its output: 'baldeagle goldeneagle peregrine saker vulture'. The prompt is '(py3)justin@wagon-helm:~/tensorflow\$'.

Figure 8-1. TensorFlow directory (Image courtesy of Justin Francis)

I then moved the images into their appropriate folders. The script accepted PNG, JPG, GIF & TIF images, but I noticed in order to prevent an error, I had to rename one file that had a lot of symbols in its name.

Training the Model

I then trained a new model and associated labels using a built-in Python script from the TensorFlow git clone. The original graph we are retraining took Google researchers two weeks to build on a desktop with eight NVidia Tesla K40s:

```
$ cd ~/tensorflow

$ python tensorflow/examples/image_retraining/retrain.py \
--bottleneck_dir=tf_files/bottlenecks \
--model_dir=tf_files/inception \
--output_graph=tf_files/retrained_graph.pb \
--output_labels=tf_files/retrained_labels.txt \
--image_dir tf_files/bird_photos
```

Since I had TensorFlow installed with GPU support, training my model took less than 10 minutes. If I had done this on my old Xeon CPU, it probably would have taken me all day! Figure 8-2 shows the results.

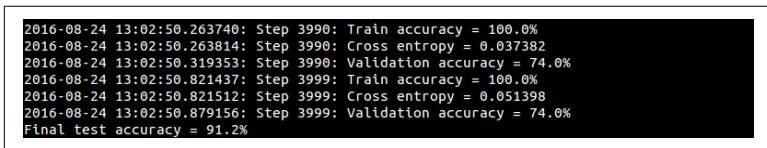
A terminal window showing the output of the retraining script. It displays training progress for steps 3990 and 3999, including train accuracy, cross entropy, and validation accuracy. The final test accuracy is 91.2%. The prompt is '(py3)justin@wagon-helm:~/tensorflow\$'.

Figure 8-2. TensorFlow classifier results 1 (image courtesy of Justin Francis)

My TensorFlow model got a final test accuracy of 91.2%. I found this very impressive given the wide variety of photos I had used.

Build the Classifier

Up to this point, I took the Inception graph and retrained it using my photos. Next, I built my image classifier from the TensorFlow git clone using Bazel. (Don't close the terminal or you will need to rebuild.)

```
$ bazel build tensorflow/examples/label_image/label_image
```

Test the Classifier

Now the fun part—testing the classifier against a new set of images. To make things easy, I put my test photos in my *tf_files* folder:

```
$ bazel-bin/tensorflow/examples/label_image/label_image \  
  --graph=tf_files/retrained_graph.pb \  
  --labels=tf_files/retrained_labels.txt \  
  --output_layer=final_result \  
  --image=tf_files/bird.jpg #This is the test image
```

First, I tried to classify a picture that I could immediately identify as a saker falcon (**Figure 8-3**).



Figure 8-3. Mature saker falcon (image courtesy of [DickDaniels on Wikimedia Commons](#))

TensorFlow agreed (see [Figure 8-4](#))!

```
I tensorflow/examples/label_image/main.cc:204] saker (4): 0.996915
I tensorflow/examples/label_image/main.cc:204] peregrine (2): 0.00152883
I tensorflow/examples/label_image/main.cc:204] goldeneagle (3): 0.000925998
I tensorflow/examples/label_image/main.cc:204] baldeagle (1): 0.000627987
I tensorflow/examples/label_image/main.cc:204] vulture (0): 1.95542e-06
(py3)justin@wagon-helm:~/tensorflow$
```

Figure 8-4. TensorFlow classifier results 2 (image courtesy of [Justin Francis](#))

Next, I tried a trickier juvenile peregrine falcon (left in [Figure 8-5](#)) that I had not used in my training data. A juvenile peregrine has front plumage similar to a saker but will develop a more striped belly, yellow beak, and a paler lower neck as it matures.



Figure 8-5. Left: juvenile peregrine falcon (*Spinus Nature Photography on Wikimedia Commons*); right: peregrine falcon (*Norbert Fischer on Wikimedia Commons*)

To my amazement, the classifier detected the juvenile peregrine with quite high accuracy (Figure 8-6).

```
I tensorflow/examples/label_image/main.cc:204] peregrine (2): 0.619089
I tensorflow/examples/label_image/main.cc:204] saker (4): 0.380429
I tensorflow/examples/label_image/main.cc:204] baldeagle (1): 0.000418734
I tensorflow/examples/label_image/main.cc:204] goldeneagle (3): 6.29243e-05
I tensorflow/examples/label_image/main.cc:204] vulture (0): 5.57738e-07
(py3)justin@wagon-helm:~/tensorflow$
```

Figure 8-6. TensorFlow classifier results 3 (image courtesy of Justin Francis)

For my last example, I used a bird that humans often misclassify: the juvenile bald eagle. People often mistake it for a golden eagle because it does not have solid white plumage on its head and tail. I trained my classifier with about 10% photos of juvenile eagles (Figure 8-7).



Figure 8-7. Newly fledged juvenile bald eagle (*KetaDesign on Wikimedia Commons*)

Seems like my classifier is not exceeding ornithologist-level intelligence yet (Figure 8-8).

```
I tensorflow/examples/label_image/main.cc:204] goldeneagle (3): 0.769338
I tensorflow/examples/label_image/main.cc:204] vulture (0): 0.223886
I tensorflow/examples/label_image/main.cc:204] baldeagle (1): 0.00397391
I tensorflow/examples/label_image/main.cc:204] saker (4): 0.00225172
I tensorflow/examples/label_image/main.cc:204] peregrine (2): 0.000550929
(py3)justin@wagon-helm:~/tensorflow$
```

Figure 8-8. TensorFlow classifier results 4 (image courtesy of Justin Francis)

I was especially surprised that “vulture” was its second guess and not bald eagle. This may be due to the fact that many of my vulture pictures were taken from a similar view.

What about Merlin? Its first choice was spot on with a very reasonable second choice (Figure 8-9).

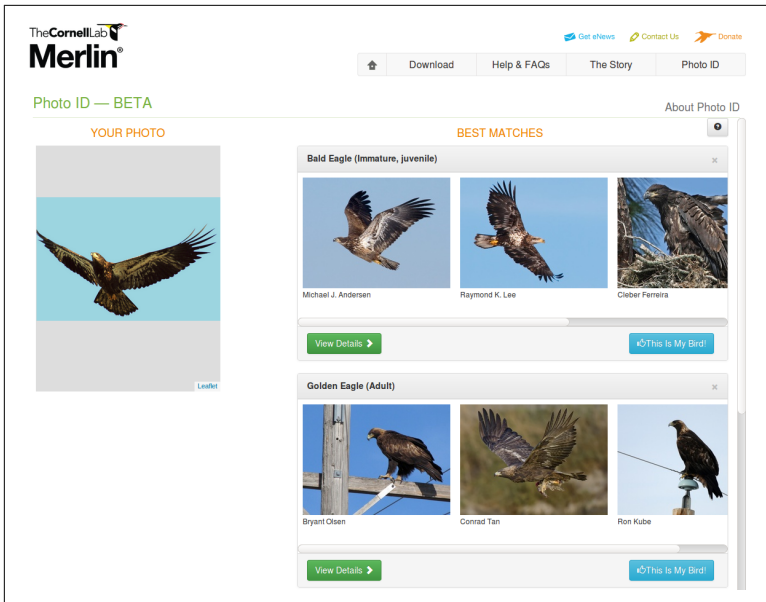


Figure 8-9. Cornell Ornithology Lab's Merlin ID tool (image courtesy of Justin Francis)

This is the only photo that Merlin beat me on. But with enough training data, I'm fully confident my classifier could learn to distinguish adult bald eagles, juvenile bald eagles, and golden eagles. I, of course, would need a separate juvenile bald eagle folder with additional images.

I had a lot of fun training and experimenting with this classifier, and I hope this post helps you make your own classifier, too. The possibilities for researchers and enthusiasts are endless. Feel free to tag me [@wagonhelm](#) or [#TensorFlow](#) on Twitter and show off what you created.

Justin Francis

Justin lives on the west coast of Canada and works on a small farm focused on permaculture ethics and design. In the past, he was the founder and educator at a nonprofit community cooperative bicycle shop. For the last two years, he lived on a sailboat exploring and experiencing the Georgia Strait full-time but is now primarily focused on studying machine learning.

A Poet Does TensorFlow

Mike Loukides

After reading Pete Warden’s excellent “[TensorFlow for poets](#)”, I was impressed at how easy it seemed to build a working deep learning classifier. It was so simple that I had to try it myself.

I have a lot of photos around, mostly of birds and butterflies. So, I decided to build a simple butterfly classifier. I chose butterflies because I didn’t have as many photos to work with, and because they were already fairly well sorted. I didn’t want to spend hours sorting a thousand or so bird pictures. According to Pete, that’s the most laborious, time-consuming part of the process: getting your training data sorted.

Sorting was relatively easy: while I thought I’d need a database, or some CSV file tagging the photos by filename, I only needed a simple directory structure: a top-level directory named `butterflies`, with a directory underneath for each kind of butterfly I was classifying. Here’s what I ended up with:

```
# ls ../tf_files/butterflies/  
Painted Lady black swallowtail monarch tiger swallowtail
```

Only four kinds of butterflies? Unfortunately, yes. While you don’t need thousands of images, you do need at least a dozen or so in each directory. If you don’t, you’ll get divide-by-zero errors that will make you pull your hair out. Pete’s code randomly separates the images you provide into a training set and a validation set. If either of those sets ends up empty, you’re in trouble. (Pete, thanks for your help understanding this!) I ended up with a classifier that only knew

about four kinds of butterflies because I had to throw out all the species where I only had six or seven (or one or two) photos. I may try adding some additional species back in later; I think I can find a few more.

I'll skip the setup (see [Pete's article for that](#)). By using VirtualBox and Docker, he eliminates pain and agony of building and installing the software, particularly if you're using OS X. If you run into strange errors, try going back to the git steps and rebuilding. TensorFlow (TF) won't survive Docker disconnecting from the VM, so if that happens (for example, if you restart the VM), you'll need to rebuild.

Here's what I did to create the classifier; it's straight from Pete's article, except for the name of the image directory. You can ignore the "No files found" for the top-level directory (butterflies), but if you see this message for any of the subdirectories, you're in trouble:

```
# bazel-bin/tensorflow/examples/image_retraining/retrain \
> --bottleneck_dir=/tf_files/bottlenecks \
> --model_dir=/tf_files/inception \
> --output_graph=/tf_files/retrained_graph.pb \
> --output_labels=/tf_files/retrained_labels.txt \
> --image_dir /tf_files/butterflies
Looking for images in 'butterflies'
No files found
Looking for images in 'black swallowtail'
Looking for images in 'monarch'
Looking for images in 'Painted Lady'
Looking for images in 'tiger swallowtail'
100 bottleneck files created.
2016-03-14 01:46:08.962029: Step 0: Train accuracy = 31.0%
2016-03-14 01:46:08.962241: Step 0: Cross entropy = 1.311761
2016-03-14 01:46:09.137622: Step 0: Validation accuracy = 18.0%
... (Lots of output deleted...)
Final test accuracy = 100.0%
Converted 2 variables to const ops.
```

And here's what happens when you actually do some classifying. Here's the image I'm trying to classify: an eastern tiger swallowtail ([Figure 9-1](#)).

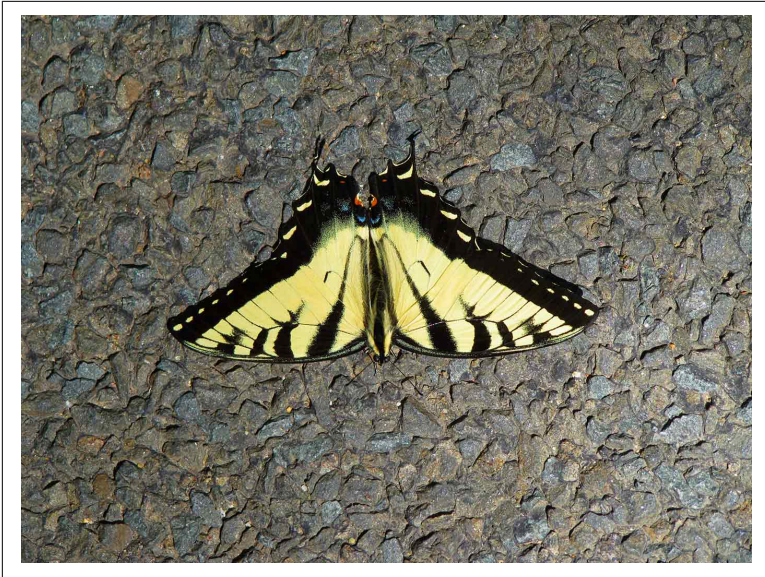


Figure 9-1. Eastern tiger swallowtail

And here's the result:

```
# bazel build tensorflow/examples/label_image:label_image && \
> bazel-bin/tensorflow/examples/label_image/label_image \
> --graph=tf_files/retrained_graph.pb \
> --labels=tf_files/retrained_labels.txt \
> --output_layer=final_result \
> --image=tf_files/sample/IMG_5941-e.jpg
(Lots of output)
INFO: Elapsed time: 532.630s, Critical Path: 515.99s
I tensorflow/examples/label_image/main.cc:206] tiger swallowtail
(1): 0.999395
I tensorflow/examples/label_image/main.cc:206] black swallowtail
(2): 0.000338286
I tensorflow/examples/label_image/main.cc:206] monarch (0):
0.000144585
I tensorflow/examples/label_image/main.cc:206] painted lady (3):
0.000121789
```

There's a 99.9% chance that picture was a tiger swallowtail. Not bad. Was I just lucky, or did it really work? [Figure 9-2](#) shows another image, this time a trickier photo of a pair of monarchs.

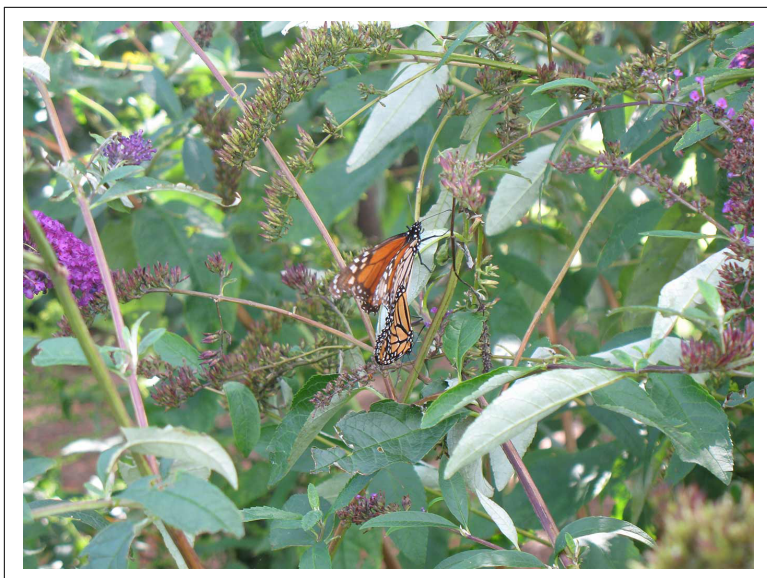


Figure 9-2. Pair of monarchs

```
# bazel build tensorflow/examples/label_image:label_image && \
bazel-bin/tensorflow/examples/label_image/label_image \
--graph=/tf_files/retrained_graph.pb \
--labels=/tf_files/retrained_labels.txt \
--output_layer=final_result \
--image=/tf_files/sample/MKL_2055.JPG
(Not quite as much output)
INFO: Elapsed time: 16.717s, Critical Path: 11.43s
I tensorflow/examples/label_image/main.cc:206] monarch (0):
0.875138
I tensorflow/examples/label_image/main.cc:206] painted lady (3):
0.117698
I tensorflow/examples/label_image/main.cc:206] tiger swallowtail
(1): 0.0054633
I tensorflow/examples/label_image/main.cc:206] black swallowtail
(2): 0.00170112
```

TF isn't as confident, but it still thinks the image is a monarch with a probability of about 87%.

I was surprised that TF worked so well. First, I thought that a successful classifier would need to be trained on thousands of photos, and I only had a hundred or so. You'd need thousands (or millions) if you're building an app for Google or Facebook, but I had at most a couple dozen in each of the four categories. That proved to be enough for my purposes. Second, the monarch is tricky; the butter-

flies are at a bad angle, and one is blurry because it was moving. I don't know why I didn't delete this image after shooting it, but it made a nice test case.

Pete pointed out that, if you don't have many images, you can **improve the accuracy** by using the `--random_crop`, `--random_scale`, and `--random_brightness` options. These make the classifier run much slower. In effect, they're creating more images by distorting the images you've provided.

Deep learning isn't magic, and playing with it will get you thinking about its limits. TensorFlow doesn't know anything about what it's classifying; it's just trying to find similar images. If you ask TensorFlow to classify something, classify it will, whether or not that classification makes any sense. It doesn't know the meaning of "I don't know." When I gave the butterfly classifier a skipper (Figure 9-3), one of a large and confusing family of small butterflies that doesn't look remotely like anything in the training set, TF classified it as a black swallowtail with 80% confidence.



Figure 9-3. Skipper butterfly

Of all the butterflies in the training set, the black swallowtail is probably the least similar (black swallowtails are, well, black). If I gave my classifier a snapshot of someone walking down the street, it

would helpfully determine the set of butterfly photos to which the photo was most similar. Can that be fixed? More training images, and more categories, would make classification more accurate, but wouldn't deal with the "don't know" problem. A larger training set might help identify a skipper (with enough photos, it could possibly even identify the type of skipper), but not a photo that's completely unexpected. Setting some sort of lower bound for confidence might help. For example, returning "don't know" if the highest confidence is under 50% might be useful for building commercial applications. But that leaves behind a lot of nuance: "I don't know, but it might be..." Pete suggests that you can solve the "don't know" problem by adding a random category that consists of miscellaneous photos unrelated to the photos in the "known" categories; this trick doesn't sound like it should work, but it's surprisingly effective.

Since TensorFlow doesn't really know anything about butterflies, or flowers, or birds, it might not be classifying based on what you think. It's easy to think that TF is comparing butterflies, but it's really just trying to find similar pictures. I don't have many pictures of swallowtails sitting still on the pavement (I suspect this one was dying). But I have many pictures of butterflies feeding on those purple flowers. Maybe TF identified the monarch correctly for the wrong reason. Maybe TF classified the skipper as a black swallowtail because it was also sitting on a purple flower, like several of the swallowtails.

Likewise, TensorFlow has no built-in sense of scale, nor should it. Photography is really good at destroying information about size, unless you're really careful about context. But for a human trying to identify something, size matters. Swallowtails and monarchs are huge as butterflies go (tiger swallowtails are the largest butterflies in North America). There are many butterflies that are tiny, and many situations in which it's important to know whether you're looking at a butterfly with a wingspan of 2 centimeters or 3, or 15. A skipper is much smaller than any swallowtail, but my skipper was cropped so that it filled most of the image, and thus looked like a giant among butterflies. I doubt that there's any way for a deep learning system to recover information about scale, aside from a very close analysis of the context.

How does TensorFlow deal with objects that look completely different from different angles? Many butterflies look completely different top and bottom (dorsal and ventral, if you know the [lingo](#)): for

example, the **painted lady**. If you're not used to thinking about butterflies, you're seeing the bottom when the wings are folded and pointing up; you're seeing the top when the wings are open and out to the sides. Can TF deal with this? Given enough images, I suspect it could; it would be an interesting experiment. Obviously, there would be no problem if you built your training set with "painted lady, dorsal" and "painted lady, ventral" as separate categories.

Finally, a thought about photography. The problem with butterflies (or birds, for that matter) is that you need to take dozens of pictures to get one good one. The animals won't stay still for you. I save a lot of my pictures, but not all of them: I delete the images that aren't focused, where the subject is moving, where it's too small, or just doesn't "look nice." We're spoiled by *National Geographic*. For a classifier, I suspect that these bad shots are as useful as the good ones, and that human aesthetics make classification more difficult. Save everything? If you're planning on building a classifier, that's the way to go.

Playing with TF was fun; I certainly didn't build anything that could be used commercially, but I did get surprisingly good results with surprisingly little effort. Now, onto the birds...can I beat **Cornell Ornithology Lab's Merlin**?

Mike Loukides

Mike Loukides is Vice President of Content Strategy for O'Reilly Media, Inc. He's edited many highly regarded books on technical subjects that don't involve Windows programming. He's particularly interested in programming languages, Unix and what passes for Unix these days, and system and network administration. Mike is the author of *System Performance Tuning* and a coauthor of *Unix Power Tools* (O'Reilly). Most recently, he's been fooling around with data and data analysis, and languages like R, Mathematica, as well as Octave, and thinking about how to make books social.

Complex Neural Networks Made Easy by Chainer

Shohei Hido

Chainer is an open source framework designed for efficient research into and development of deep learning algorithms. In this post, we briefly introduce Chainer with a few examples and compare with other frameworks such as Caffe, Theano, Torch, and TensorFlow.

Most existing frameworks construct a computational graph in advance of training. This approach is fairly straightforward, especially for implementing fixed and layer-wise neural networks like convolutional neural networks.

However, state-of-the-art performance and new applications are now coming from more complex networks, such as recurrent or stochastic neural networks. Though existing frameworks can be used for these kinds of complex networks, it sometimes requires (dirty) hacks that can reduce development efficiency and maintainability of the code.

Chainer's approach is unique: building the computational graph on-the-fly *during* training.

This allows users to change the graph at each iteration or for each sample, depending on conditions. It is also easy to debug and refactor Chainer-based code with a standard debugger and profiler, since Chainer provides an imperative API in plain Python and NumPy. This gives much greater flexibility in the implementation of complex

neural networks, which leads in turn to faster iteration, and greater ability to quickly realize cutting-edge deep learning algorithms.

In the following sections, I describe how Chainer actually works and what kind of benefits users can get from it.

Chainer Basics

Chainer is a standalone deep learning framework based on Python.

Unlike other frameworks with a Python interface such as Theano and TensorFlow, Chainer provides imperative ways of declaring neural networks by supporting NumPy-compatible operations between arrays. Chainer also includes a GPU-based numerical computation library named CuPy:

```
>>> from chainer import Variable
>>> import numpy as np
```

A class `Variable` represents the unit of computation by wrapping `numpy.ndarray` in it (`.data`):

```
>>> x = Variable(np.asarray([[0, 2],[1, -3]]).astype
(np.float32))
>>> print(x.data)
[[ 0.  2.]
 [ 1. -3.]]
```

Users can define operations and functions (instances of `Function`) directly on `Variables`:

```
>>> y = x ** 2 - x + 1
>>> print(y.data)
[[ 1.  3.]
 [ 1. 13.]]
```

Since `Variables` remember what they are generated from, `Variable y` has the additive operation as its parent (`.creator`):

```
>>> print(y.creator)
<chainer.functions.math.basic_math.AddConstant at 0x7f939XXXXX>
```

This mechanism makes backward computation possible by tracking back the entire path from the final loss function to the input, which is memorized through the execution of forward computation—without defining the computational graph in advance.

Many numerical operations and activation functions are given in `chainer.functions`. Standard neural network operations such as

fully connected linear and convolutional layers are implemented in Chainer as an instance of `Link`. A `Link` can be thought of as a function together with its corresponding learnable parameters (such as weight and bias parameters, for example). It is also possible to create a `Link` that itself contains several other links. Such a container of links is called a `Chain`. This allows Chainer to support modeling a neural network as a hierarchy of links and chains. Chainer also supports state-of-the-art optimization methods, serialization, and CUDA-powered faster computations with CuPy:

```
>>> import chainer.functions as F
>>> import chainer.links as L
>>> from chainer import Chain, optimizers, serializers, cuda
>>> import cupy as cp
```

Chainer's Design: Define-by-Run

To train a neural network, three steps are needed: (1) build a computational graph from network definition, (2) input training data and compute the loss function, and (3) update the parameters using an optimizer and repeat until convergence.

Usually, DL frameworks complete step one in advance of step two. We call this approach *define-and-run* (Figure 10-1).

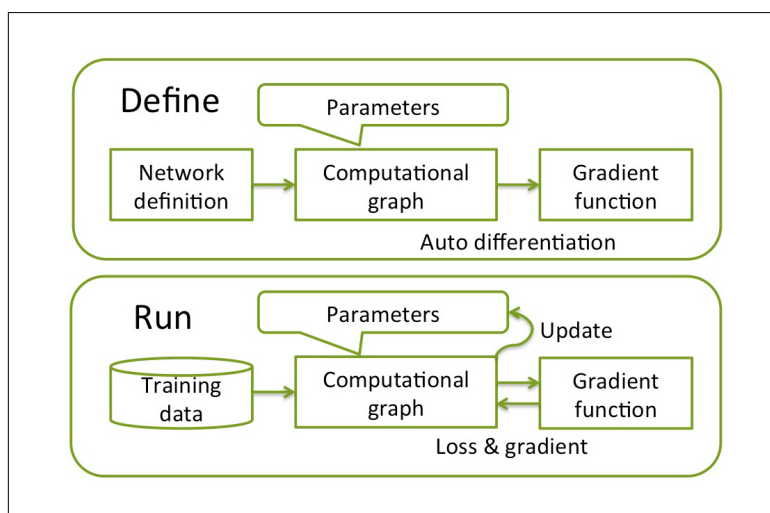


Figure 10-1. DL frameworks *define-and-run* (all images courtesy of Shohei Hido)

This is straightforward but not optimal for complex neural networks since the graph must be fixed before training. Therefore, when implementing recurrent neural networks, for example, users are forced to exploit special tricks (such as the `scan()` function in Theano) that make it harder to debug and maintain the code.

Instead, Chainer uses a unique approach called *define-by-run*, which combines steps one and two into a single step (Figure 10-2).

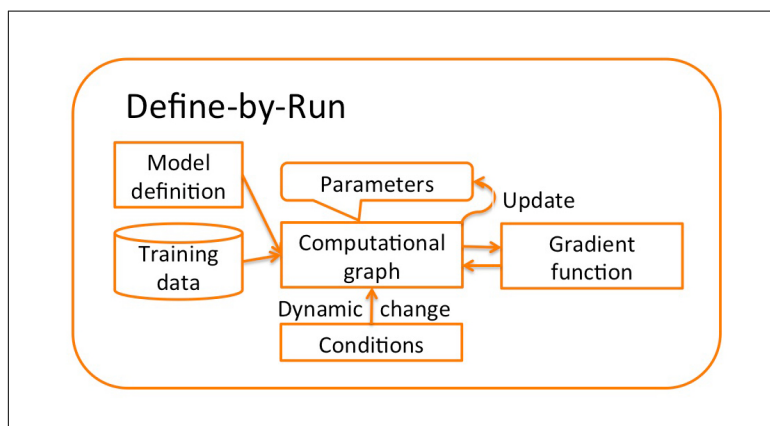


Figure 10-2. Chainer uses a unique approach called *define-by-run*

The computational graph is not given before training but obtained in the course of training. Since forward computation directly corresponds to the computational graph and backpropagation through it, any modifications to the computational graph can be done in the forward computation at each iteration and even for each sample.

As a simple example, let's see what happens using two-layer perceptron for MNIST digit classification (Figure 10-3).

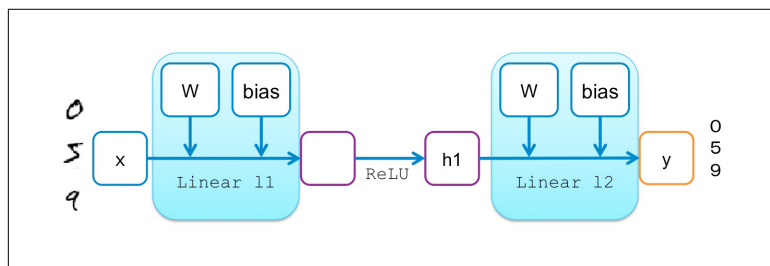


Figure 10-3. Two-layer perceptron for MNIST digit classification

The following code shows the implementation of two-layer perceptron in Chainer:

```
# 2-layer Multi-Layer Perceptron (MLP)
class MLP(Chain):

    def __init__(self):
        super(MLP, self).__init__(
            l1=L.Linear(784, 100),
            # From 784-dimensional input to hidden unit with
            # 100 nodes
            l2=L.Linear(100, 10),
            # From hidden unit with 100 nodes to output unit
            # with 10 nodes (10 classes)
        )

    # Forward computation
    def __call__(self, x):
        h1 = F.tanh(self.l1(x))
        # Forward from x to h1 through activation with
        # tanh function
        y = self.l2(h1)
        # Forward from h1 to y
        return y
```

In the constructor (`__init__`), we define two linear transformations from the input to hidden units, and hidden to output units, respectively. Note that no connection between these transformations is defined at this point, which means that the computation graph is not even generated, let alone fixed.

Instead, their relationship will be later given in the forward computation (`__call__`), by defining the activation function (`F.tanh`) between the layers. Once forward computation is finished for a minibatch on the MNIST training data set (784 dimensions), the computational graph in [Figure 10-4](#) can be obtained on the fly by backtracking from the final node (the output of the loss function) to the input (note that `SoftmaxCrossEntropy` is also introduced as the loss function).

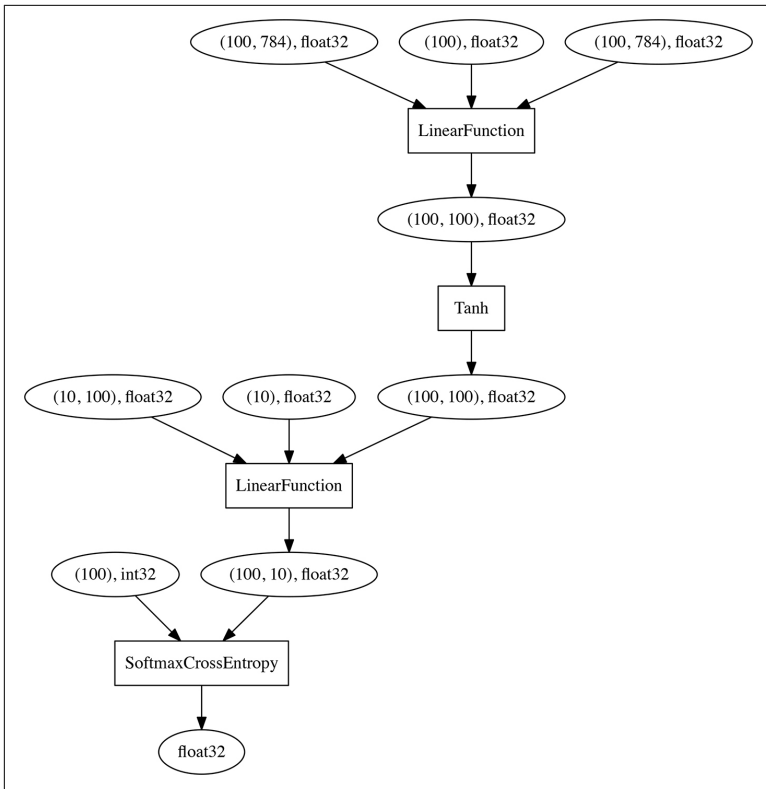


Figure 10-4. Computational graph

The point is that the network definition is simply represented in Python rather than a domain-specific language (DSL), so users can make changes to the network in each iteration (forward computation).

This imperative declaration of neural networks allows users to use standard Python syntax for branching, without studying any DSL. That can be beneficial compared to the symbolic approaches that TensorFlow and Theano utilize and also the text DSL that Caffe and CNTK rely on.

In addition, a standard debugger and profiler can be used to find the bugs, refactor the code, and also tune the hyper-parameters. On the other hand, although Torch and MXNet also allow users to employ imperative modeling of neural networks, they still use the define-and-run approach for building a computational graph object, so debugging requires special care.

Implementing Complex Neural Networks

Figure 10-4 is just an example of a simple and fixed neural network. Next, let's look at how complex neural networks can be implemented in Chainer.

A *recurrent neural network* is a type of neural network that takes sequence as input, so it is frequently used for tasks in natural language processing such as sequence-to-sequence translation and question answering systems. It updates the internal state depending not only on each tuple from the input sequence, but also on its previous state so it can take into account dependencies across the sequence of tuples.

Since the computational graph of a recurrent neural network contains directed edges between previous and current time steps, its construction and backpropagation are different from those for fixed neural networks, such as convolutional neural networks. In current practice, such cyclic computational graphs are unfolded into a directed acyclic graph each time for model update by a method called truncated backpropagation through time.

For this example, the target task is to predict the next word given a part of sentence. A successfully trained neural network is expected to generate syntactically correct words rather than random words, even if the entire sentence does not make sense to humans. The following example shows a simple recurrent neural network with one recurrent hidden unit:

```
# Definition of simple recurrent neural network
class SimpleRNN(Chain):

    def __init__(self, n_vocab, n_nodes):
        super(SimpleRNN, self).__init__(
            embed=L.EmbedID(n_vocab, n_nodes),
            # word embedding
            x2h=L.Linear(n_nodes, n_nodes),
            # the first linear layer
            h2h=L.Linear(n_nodes, n_nodes),
            # the second linear layer
            h2y=L.Linear(n_nodes, n_vocab),
            # the feed-forward output layer
        )
        self.h_internal=None # recurrent state

    def forward_one_step(self, x, h):
        x = F.tanh(self.embed(x))
```

```

if h is None: # branching in network
    h = F.tanh(self.x2h(x))
else:
    h = F.tanh(self.x2h(x) + self.h2h(h))
y = self.h2y(h)
return y, h

def __call__(self, x):
    # given the current word ID, predict the next word ID.
    y, h = self.forward_one_step(x, self.h_internal)
    self.h_internal = h # update internal state
    return y

```

Only the types and sizes of layers are defined in the constructor as well as on the multilayer perceptron. Given input word and current state as arguments, `forward_one_step()` method returns output word and new state. In the forward computation (`__call__`), `forward_one_step()` is called for each step and updates the hidden recurrent state with a new one.

By using the popular text data set **Penn Treebank** (PTB), we trained a model to predict the next word from probable vocabularies. Then the trained model is used to predict subsequent words using weighted sampling:

```

"If you build it," => "would a outlawed a out a tumor a colonial
a"
"If you build it, they" => " a passed a president a glad a
senate a billion"
"If you build it, they will" => " for a billing a jerome a
contracting a surgical a"
"If you build it, they will come" => "a interviewed a invites a
boren a illustrated a pinnacle"

```

This model has learned—and then produced—many repeated pairs of “a” and a noun or an adjective. Which means “a” is one of the most probable words, and a noun or adjective tend to follow “a.”

To humans, the results look almost the same, being syntactically wrong and meaningless, even when using different inputs. However, these are definitely inferred based on the real sentences in the data set by training the type of words and relationship between them.

Though this is inevitable due to the lack of expressiveness in the SimpleRNN model, the point here is that users can implement any kinds of recurrent neural networks just like SimpleRNN.

Just for comparison, by using off-the-shelf mode of recurrent neural network called *long short-term memory* (LSTM), the generated texts become more syntactically correct:

```
"If you build it," => "pension say computer ira <EOS> a week ago
the japanese"
"If you buildt it, they" => "were jointly expecting but too well
put the <unknown> to"
"If you build it, they will" => "see the <unknown> level that
would arrive in a relevant"
"If you build it, they will come" => "to teachers without an
mess <EOS> but he says store"
```

Since popular RNN components such as LSTM and *gated recurrent unit* (GRU) have already been implemented in most of the frameworks, users do not need to care about the underlying implementations. However, if you want to significantly modify them or make a completely new algorithm and components, the flexibility of Chainer makes a great difference compared to other frameworks.

Stochastically Changing Neural Networks

In the same way, it is very easy to implement stochastically changing neural networks with Chainer.

The following is mock code to implement Stochastic ResNet. In `__call__`, just flip a skewed coin with probability p , and change the forward path by having or not having unit f . This is done at each iteration for each minibatch, and the memorized computational graph is different each time but updated accordingly with backpropagation after computing the loss function:

```
# Mock code of Stochastic ResNet in Chainer
class StochasticResNet(Chain):

    def __init__(self, prob, size, **kwargs):
        super(StochasticResNet, self).__init__(size, **kwargs)
        self.p = prob # Survival probabilities

    def __call__(self, h):
        for i in range(self.size):
            b = numpy.random.binomial(1, self.p[i])
            c = self.f[i](h) + h if b == 1 else h
            h = F.relu(c)
        return h
```


Conclusion

Chainer also has many features to help users to realize neural networks for their tasks as easily and efficiently as possible.

CuPy is a NumPy-equivalent array backend for GPUs included in Chainer, which enables CPU/GPU-agnostic coding, just like NumPy-based operations. The training loop and data set handling can be abstracted by Trainer, which keeps users away from writing such routines every time, and allows them to focus on writing innovative algorithms. Though scalability and performance are not the main focus of Chainer, it is still competitive with other frameworks, as shown in the [public benchmark results](#), by making full use of NVIDIA's CUDA and cuDNN.

Chainer has been used in many academic papers not only for computer vision, but also speech processing, natural language processing, and robotics. Moreover, Chainer is gaining popularity in many industries since it is good for research and development of new products and services. [Toyota](#), [Panasonic](#), and [FANUC](#) are among the companies that use Chainer extensively and have shown some demonstrations, in partnership with the original Chainer development team at Preferred Networks.

Interested readers are encouraged to visit the [Chainer website](#) for further details. I hope Chainer will make a difference for cutting-edge research and real-world products based on deep learning!

Shohei Hido

Shohei Hido is the chief research officer of Preferred Networks, a spin-off company of Preferred Infrastructure, Inc., where he is currently responsible for Deep Intelligence in Motion, a software platform for using deep learning in IoT applications. Previously, Shohei was the leader of Preferred Infrastructure's Jubatus project, an open source software framework for real-time, streaming machine learning. He also worked at IBM Research in Tokyo for six years as a staff researcher in machine learning and its applications to industries. Shohei holds an MS in informatics from Kyoto University.

Building Intelligent Applications with Deep Learning and TensorFlow

Ben Lorica

This June, I spoke with [Rajat Monga](#), who serves as a director of engineering at Google and manages the [TensorFlow](#) engineering team. We talked about how he ended up [working on deep learning](#), the current state of TensorFlow, and the applications of deep learning to products at Google and other companies. Here are some highlights from [our conversation](#).

Deep Learning at Google

There's not going to be too many areas left that run without machine learning that you can program. The data is too much, there's just too much for humans to handle. ... Over the last few years, and this is something we've seen at Google, we've seen hundreds of products move to deep learning, and gain from that. In some cases, these are products that were actually applying machine learning that had been using traditional methods for a long time and had experts. For example, search, we had hundreds of signals in there, and then we applied deep learning. That was the last two years or so.

For somebody who is not familiar with deep learning, my suggestion would be to start from an example that is closest to your problem, and then try to adapt it to your problem. Start simple; don't go

to very complex things. There are many things you can do, even with simple models.

TensorFlow Makes Deep Learning More Accessible

At Google, I would say there are the machine learning researchers who are pushing machine learning research, then there are data scientists who are focusing on applying machine learning to their problems... We have a mix of people—some are people applying TensorFlow to their actual problems.

They don't always have a machine learning background. Some of them do, but a large number of them don't. They're usually developers who are good at writing software. They know maybe a little bit of math so they can pick it up, in some cases not that much at all, but who can take these libraries if there are examples. They start from those examples, maybe ask a few questions on our internal boards, and then go from there. In some cases they may have a new problem, they want some inputs on how to formulate that problem using deep learning, and we might guide them or point them to an example of how you might approach their problem. Largely, they've been able to take TensorFlow and do things on their own. Internally, we are definitely seeing these tools and techniques being used by people who have never done machine learning before.

Synchronous and Asynchronous Methods for Training Deep Neural Networks

When we started out back in 2011, everybody was using stochastic gradient descent. It's extremely efficient in what it does; but when you want to scale beyond 10 or 20 machines, it makes it hard to scale. So what do we do? At that time there were a couple of papers. One was on the **HOGWILD!** approach that people had done on a single machine... That was very interesting. We thought, can we make this work across the network, across many, many machines? We did some experiments and started tuning it, and it worked well. We were actually able to scale it to a large number of workers, hundreds of workers in some cases across thousands of machines, and that worked pretty well. Over time, we'd always had another question: is the asynchronous nature actually helping or making things worse? Finally last year, we started to experiment and try to understand what's happening; and as part of that, we realized if we could do synchronous well, it actually is better.

... With the asynchronous stuff, we had these workers and they would work completely independently of each other. They would just update things on the parameter server when they had gradients. They would send it back to the parameter server. It would update and then fetch the next set of parameters.

... From a systems perspective, it's nice, because it scales very, very well. It's okay if a few workers died—that's fine—all the others will continue to make progress. Now, with the synchronous approach, what we want to do is to send parameters out to all the workers, have them compute gradients, send those back, combine those together, and then apply them. Now, across many machines, you can do this; but the issue is if some of them start to slow down or fail, what happens then? That's always a tricky thing with the synchronous approach, and that's hard to scale. That's probably the biggest reason people hadn't pushed toward this earlier.

Related Resources

In my conversation with Rajat Monga, I alluded to these recent papers on asynchronous and synchronous methods for training deep neural networks:

- [“Revisiting Distributed Synchronous SGD”](#)
- [“Asynchrony begets Momentum, with an Application to Deep Learning”](#)
- [“Omnivore: An Optimizer for Multi-device Deep Learning on CPUs and GPUs”](#)

Ben Lorica

Ben Lorica is the Chief Data Scientist and Director of Content Strategy for Data at O'Reilly Media, Inc. He has applied business intelligence, data mining, machine learning, and statistical analysis in a variety of settings, including direct marketing, consumer and market research, target advertising, text mining, and financial engineering. His background includes stints with an investment management company, internet startups, and financial services.

Homebuilt Autonomous Systems

The availability of AI tools, libraries, cloud processing, and mobile computing is incredibly well illustrated in these two projects by Lukas Biewald: a rolling robot that recognizes a variety of objects, and a flying drone that recognizes people (with just a little help from the cloud).

How to Build a Robot That “Sees” with \$100 and TensorFlow

Lukas Biewald

Object recognition is one of the most exciting areas in machine learning right now. Computers have been able to recognize objects like faces or cats reliably for quite a while, but recognizing arbitrary objects within a larger image has been the holy grail of artificial intelligence. Maybe the real surprise is that human brains recognize objects so well. We effortlessly convert photons bouncing off objects at slightly different frequencies into a spectacularly rich set of information about the world around us. Machine learning still struggles with these simple tasks, but in the past few years, it's gotten much better.

Deep learning and a large public training data set called **ImageNet** have made an impressive amount of progress toward object recognition. **TensorFlow** is a well-known framework that makes it very easy to implement deep learning algorithms on a variety of architectures. TensorFlow is especially good at taking advantage of GPUs, which in turn are also very good at running deep learning algorithms.

Building My Robot

I wanted to build a robot that could recognize objects. Years of experience building computer programs and doing test-driven development have turned me into a menace working on physical projects. In the real world, testing your buggy device can burn down your

house, or at least fry your motor and force you to wait a couple of days for replacement parts to arrive.

The new third-generation **Raspberry Pi** is perfect for this kind of project. It costs \$36 on Amazon.com and has WiFi, a quad core CPU, and a gigabyte of RAM. A \$6 **microSD card** can load **Raspber-ian**, which is basically **Debian**. See **Figure 12-1** for an overview of how all the components worked together, and see **Figure 12-2** for a photo of the Pi.

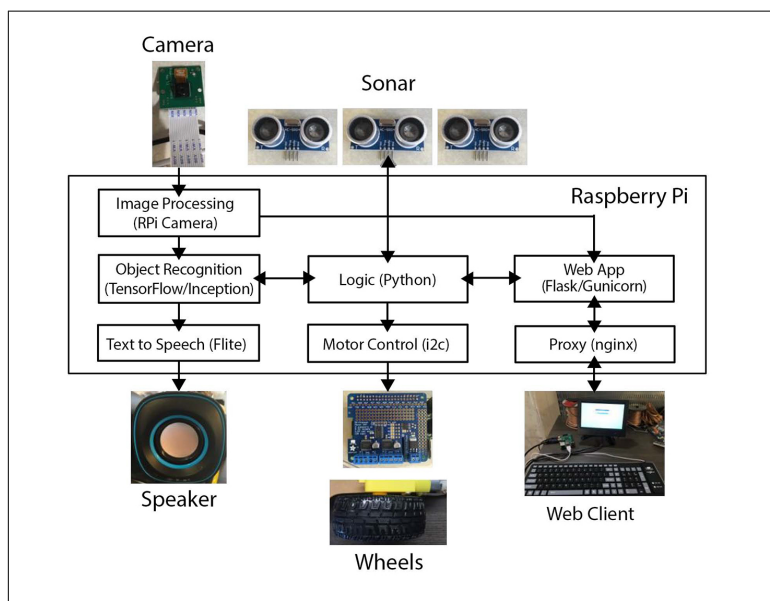


Figure 12-1. Architecture of the object-recognizing robot (image courtesy of Lukas Biewald)

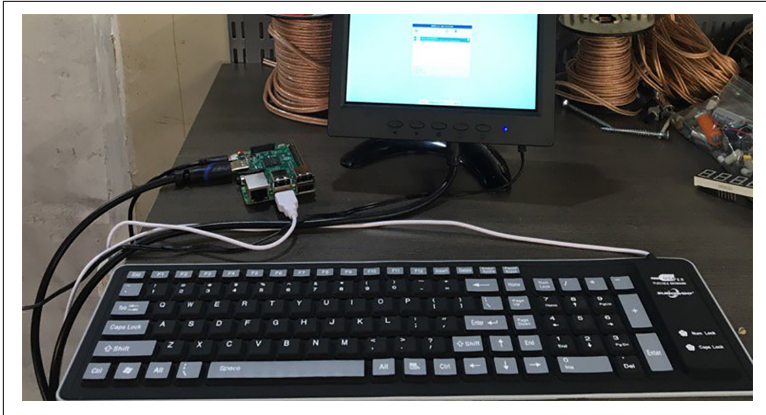


Figure 12-2. Raspberry Pi running in my garage (image courtesy of Lukas Biewald)

I love the **cheap robot chassis** that Sain Smart makes for around \$11. The chassis turns by spinning the wheels at different speeds, which works surprisingly well (see **Figure 12-3**).

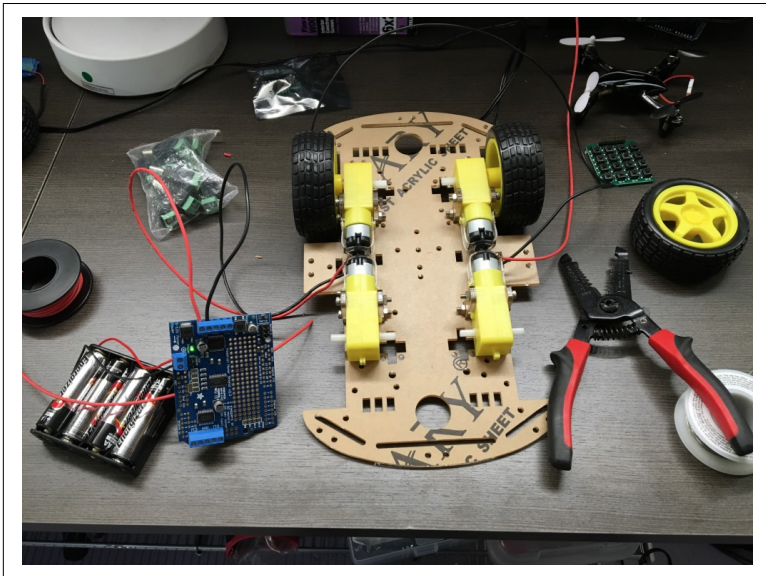


Figure 12-3. Robot chassis (image courtesy of Lukas Biewald)

The one place I spent more money when cheaper options were available is the **Adafruit motor hat** (see **Figure 12-4**). The DC motors run at a higher current than the Raspberry Pi can provide, so

a separate controller is necessary, and the Adafruit motor hat is super convenient. Using the motor hat required a tiny bit of soldering, but the hardware is extremely forgiving, and Adafruit provides a nice library and tutorial to control the motors over *i²C*. Initially, I used cheaper motor controllers, but I accidentally fried my Pi, so I decided to order a better quality replacement.

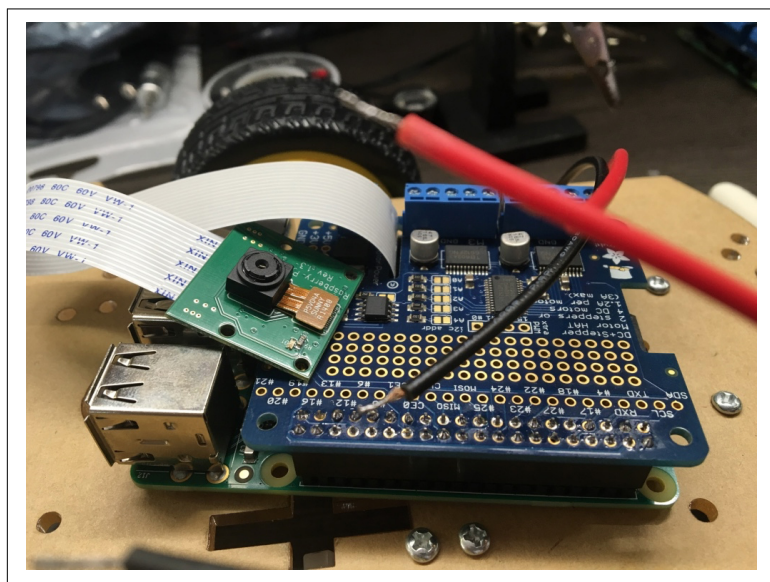


Figure 12-4. Raspberry Pi with motor hat and camera (image courtesy of Lukas Biewald)

A **\$15 camera** attaches right into the Raspberry Pi and provides a real-time video feed I can use to recognize objects. There are tons of awesome cameras available. I like the infrared cameras that offer night vision.

The Raspberry Pi needs about 2 amps of current, but 3 amps is safer with the speaker we’re going to plug into it. iPhone battery chargers work awesomely for this task. Small chargers don’t actually output enough amps and can cause problems, but the **Lumsing power bank** works great and costs \$18.

A couple of **4 sonar sensors** help the robot avoid crashing into things—you can buy five for \$11.

I added the cheapest USB speakers I could find, and used a bunch of zip ties, hot glue, and foam board to keep everything together. As an

added bonus, I cut up some of the packaging materials the electronics came with and drew on them to give the robots some personality. I should note here that I actually built *two* robots (see [Figure 12-5](#)) because I was experimenting with different chassis, cameras, sonar placement, software, and so forth, and ended up buying enough parts for two versions.

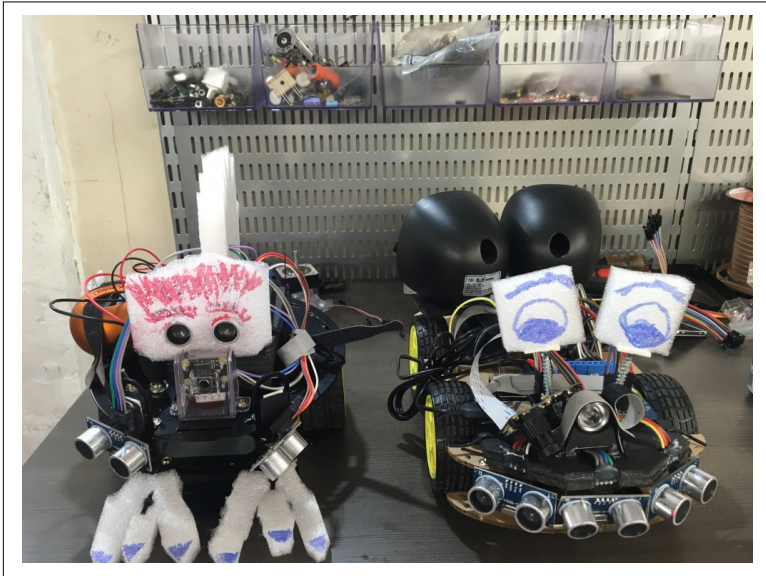


Figure 12-5. My 4WD robot (right) and his 2WD older sister (image courtesy of Lukas Biewald)

Once the robot is assembled, it's time to make it smart. There are a million [tutorials](#) for getting started with a Raspberry Pi online. If you've used Linux, everything should be very familiar.

For streaming the camera, the [RPi Cam Web interface](#) works great. It's super configurable and by default puts the latest image from the camera in a RAM disk at `/dev/shm/mjpeg/cam.jpg`.

If you want to stream the camera data to a web page (very useful for debugging), you can install [Nginx](#), an extremely fast open source webserver/proxy. I configured Nginx to pass requests for the camera image directly to the file location and everything else to my webserver:

```
http {  
    server {
```

```

        location / {
            proxy_pass http://unix:/home/pi/drive.sock;
        }
        location /cam.jpg {
            root /dev/shm/mjpeg;
        }
    }
}

```

I then built a simple Python webserver to spin the wheels of the robot based on keyboard commands that made for a nifty remote control car.

As a side note, it's fun to play with the sonar and the driving system to build a car that can maneuver around obstacles.

Programming My Robot

Finally, it's time to install TensorFlow. There are a couple of ways to do the installation, but TensorFlow actually comes with a makefile that lets you build it right on the system. The **steps** take a few hours and have quite a few dependencies, but they worked great for me.

TensorFlow comes with a prebuilt model called “inception” that performs object recognition. You can follow the **tutorial** to get it running.

To output the top five guesses, run `tensorflow/contrib/pi_examples/label_image/gen/bin/label_image` on an image from the camera. The model works surprisingly well on a wide range of inputs, but it's clearly missing an accurate “prior,” or a sense of what things it's likely to see, and there are quite a lot of objects missing from the training data. For example, it consistently recognizes my laptop, even at funny angles; but if I point it at my basket of loose wires, it consistently decides that it's looking at a toaster. If the camera is blocked, and it gets a dark or blurry image, it usually decides that it's looking at nematodes—clearly an artifact of the data it was trained on.

Finally, I connected the output to the **Flite** open source software package that does text to speech, so the robot can tell everyone what it's seeing (see **Figure 12-6**).

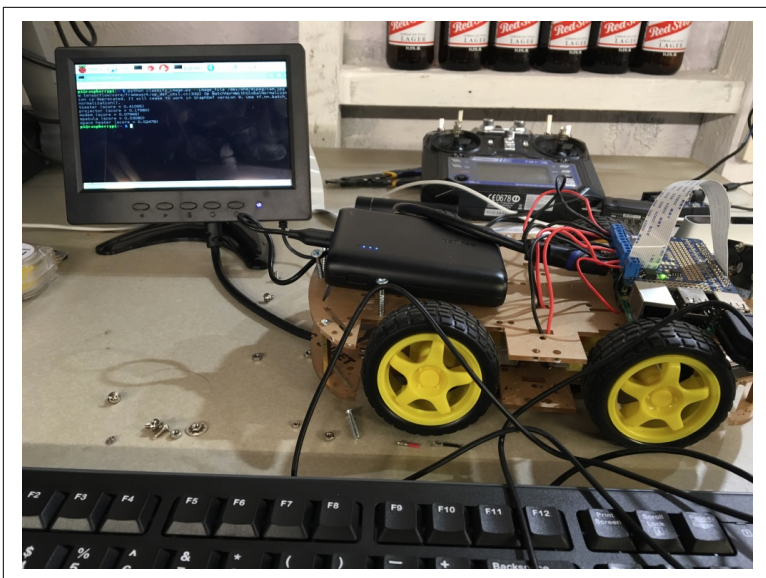


Figure 12-6. Robot plugged into my keyboard and monitor (image courtesy of Lukas Biewald)

Final Thoughts

From 2003 to 2005, I worked in the Stanford Robotics lab, where the robots cost hundreds of thousands of dollars and couldn't perform object recognition nearly as well as my robots. I'm excited to put this software on my drone and never have to look for my keys again.

I'd also like to acknowledge all the people that helped with this fun project. My neighbors, Chris Van Dyke and Shruti Gandhi, helped give the robot a friendly personality. My friend Ed McCullough dramatically improved the hardware design and taught me the value of hot glue and foam board. Pete Warden, who works at Google, helped get TensorFlow compiling properly on the Raspberry Pi and provided amazing customer support.

Lukas Biewald

Lukas Biewald is the founder of **CrowdFlower**. Founded in 2009, CrowdFlower is a data enrichment platform that taps into an on-demand workforce to help companies collect training data and do human-in-the-loop machine learning.

Following his graduation from Stanford University with a BS in mathematics and an MS in computer science, Lukas led the Search Relevance Team for Yahoo! Japan. He then worked as a senior data scientist at Powerset, which was acquired by Microsoft in 2008. Lukas was featured in *Inc.* magazine's 30 under 30 list.

Lukas is also an expert-level Go player.

How to Build an Autonomous, Voice-Controlled, Face-Recognizing Drone for \$200

Lukas Biewald

After building an **image-classifying robot**, the obvious next step was to make a version that can fly. I decided to construct an autonomous drone that could recognize faces and respond to voice commands.

Choosing a Prebuilt Drone

One of the hardest parts about hacking drones is getting started. I got my feet wet first by building a drone from parts, but like pretty much all of my DIY projects, building from scratch ended up costing me way more than buying a prebuilt version—and frankly, my homemade drone never flew quite right. It's definitely much easier and cheaper to buy than to build.

Most of the drone manufacturers claim to offer APIs, but there's not an obvious winner in terms of a hobbyist ecosystem. Most of the drones with usable-looking APIs cost more than \$1,000—a huge barrier to entry.

But after some research, I found the **Parrot AR Drone 2.0** (see **Figure 13-1**), which I think is a clear choice for a fun, low-end, hackable drone. You can buy one for \$200 new, but so many people buy drones and never end up using them that a secondhand drone is a good option and available widely on eBay for \$130 or less.



Figure 13-1. The drone collection in my garage; the Parrot AR drone I used is hanging on the far left (image courtesy of Lukas Biewald)

The Parrot AR drone doesn't fly quite as stably as the much more expensive (about \$550) new **Parrot Bebop 2 drone**, but the Parrot AR comes with an excellent node.js client library called **node-ar-drone** that is perfect for building onto.

Another advantage: the Parrot AR drone is very hard to break. While testing the autonomous code, I crashed it repeatedly into walls, furniture, house plants, and guests, and it still flies great.

The worst thing about hacking on drones compared to hacking on terrestrial robots is the short battery life. The batteries take hours to charge and then last for about 10 minutes of flying. I recommend buying two additional batteries and cycling through them while testing.

Programming My Drone

Javascript turns out to be a great language for controlling drones because it is so inherently event driven. And trust me, while flying a drone, there will be a lot of asynchronous events. Node isn't a language I've spent a lot of time with, but I walked away from this project super impressed with it. The last time I seriously programmed robots, I used C, where the threading and exception handling is painful enough that there is a tendency to avoid it. I hope someone builds Javascript wrappers for other drone platforms because the

language makes it easy and fun to deal with our indeterministic world.

Architecture

I decided to run the logic on my laptop and do the machine learning in the cloud. This setup led to lower latency than running a neural network directly on **Raspberry Pi** hardware, and I think this architecture makes sense for hobby drone projects at the moment.

Microsoft, Google, IBM, and Amazon all have fast, inexpensive cloud machine learning APIs. In the end, I used **Microsoft's Cognitive Services APIs** for this project because it's the only API that offers custom facial recognition.

See **Figure 13-2** for a diagram illustrating the architecture of the drone.

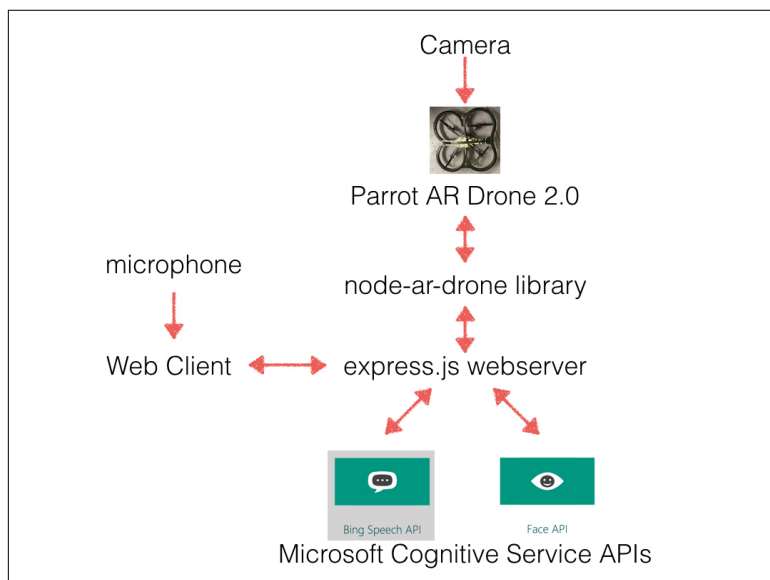


Figure 13-2. The smart drone architecture (image courtesy of Lukas Biewald)

Getting Started

By default, the Parrot AR Drone 2.0 serves a wireless network that clients connect to. This is incredibly annoying for hacking. Every

time you want to try something, you need to disconnect from your network and get on the drone's network. Luckily, there is a super useful project called [ardrone-wpa2](#) that has a script to hack your drone to join your own WiFi network.

It's fun to [Telnet](#) into your drone and poke around. The Parrot runs a stripped down version of Linux. When was the last time you connected to something with Telnet? Here's an example of how you would open a terminal and log into the drone's computer directly:

```
% script/connect "The Optics Lab" -p "particleorwave" -a
192.168.0.1 -d 192.168.7.43
% telnet 192.168.7.43
```

Flying from the Command Line

After installing the node library, it's fun to make a node.js REPL (Read-Evaluate-Print Loop) and steer your drone:

```
var arDrone = require('ar-drone');
var client = arDrone.createClient({ip: '192.168.7.43'});
client.createRepl();

drone> takeoff()
true

drone> client.animate('yawDance', 1.0)
```

If you are actually following along, by now you've definitely crashed your drone—at least a few times. I super-glued the safety hull back together about a thousand times before it disintegrated and I had to buy a new one. I hesitate to mention this, but the Parrot AR actually flies a lot better without the safety hull. This configuration makes the drone much more dangerous without the hull because when the drone bumps into something the propellers can snap, and it will leave marks in furniture.

Flying from a Web Page

It's satisfying and easy to build a web-based interface to the drone (see [Figure 13-3](#)). The express.js framework makes it simple to build a nice little web server:

```
var express = require('express');

app.get('/', function (req, res) {
  res.sendFile(path.join(__dirname + '/index.html'));
```

```
});

app.get('/land', function(req, res) {
  client.land();
});

app.get('/takeoff', function(req, res) {
  client.takeoff();
});

app.listen(3000, function () {
});
```

I set up a function to make **AJAX** requests using buttons:

```
<html>
<script language='javascript'>
function call(name) {
  var xhr = new XMLHttpRequest();
  xhr.open('GET', name, true);
  xhr.send();
}
</script>
<body>
<a onclick="call('takeoff');">Takeoff</a>
<a onclick="call('land');">Land</a>
</body>
</html>
```

Streaming Video from the Drone

I found the best way to send a feed from the drone's camera was to open up a connection and send a continuous stream of PNGs for my webserver to my website. My webserver continuously pulls PNGs from the drone's camera using the AR drone library:

```
var pngStream = client.getPngStream();

pngStream
  .on('error', console.log)
  .on('data', function(pngBuffer) {
    sendPng(pngBuffer);
  })

function sendPng(buffer) {
  res.write('--daboundary\nContent-Type: image/png\nContent-length: ' + buffer.length + '\n\n');
  res.write(buffer);
};
```

Running Face Recognition on the Drone Images

The **Azure Face API** is powerful and simple to use. You can upload pictures of your friends, and it will identify them. It will also guess age and gender, both functions of which I found to be surprisingly accurate. The latency is around 200 milliseconds, and it costs \$1.50 per 1,000 predictions, which feels completely reasonable for this application. The following code sends an image and does face recognition:

```
var oxford = require('project-oxford'),
    oxc = new oxford.Client(CLIENT_KEY);

loadFaces = function() {
  chris_url = "https://media.licdn.com/mpr/mpr/shrinknp_400_400/AAEAAQAAAAAAAAALyAAAAJGMyNmIzNWM0LTA5MTYtNDU0Mj05YjExLTgyMzVlMTZjYjEwYw.jpg";
  lukas_url = "https://media.licdn.com/mpr/mpr/shrinknp_400_400/p/3/000/058/147/34969d0.jpg";
  oxc.face.faceList.create('myFaces');
  oxc.face.faceList.addFace('myFaces', {url => chris_url,
    name=> 'Chris'});
  oxc.face.faceList.addFace('myFaces', {url => lukas_url,
    name=> 'Lukas'});
}

oxc.face.detect({
  path: 'camera.png',
  analyzesAge: true,
  analyzesGender: true
}).then(function (response) {
  if (response.length > 0) {
    drawFaces(response, filename)
  }
});
```

I used the excellent **ImageMagick** library to annotate the faces in my PNGs. There are a lot of possible extensions at this point—for example, there is an **emotion API** that can determine the emotion of faces.

Running Speech Recognition to Drive the Drone

The trickiest part about doing speech recognition was not the speech recognition itself, but streaming audio from a web page to my local server in the format **Microsoft's Speech API** wants, so that ends up being the bulk of the code. Once you've got the audio saved with one channel and the right sample frequency, the API works great and is extremely easy to use. It costs \$4 per 1,000 requests, so for hobby applications, it's basically free.

RecordRTC has a great library, and it's a good starting point for doing client-side web audio recording. On the client side, we can add code to save the audio file:

```
app.post('/audio', function(req, res) {
  var form = new formidable.IncomingForm();
  // specify that we want to allow the user to upload multiple
  // files in a single request
  form.multiples = true;
  form.uploadDir = path.join(__dirname, '/uploads');

  form.on('file', function(field, file) {
    filename = "audio.wav"
    fs.rename(file.path, path.join(form.uploadDir,
      filename));
  });

  // log any errors that occur
  form.on('error', function(err) {
    console.log('An error has occurred: \n' + err);
  });

  // once all the files have been uploaded, send a response to
  // the client
  form.on('end', function() {
    res.end('success');
  });

  // parse the incoming request containing the form data
  form.parse(req)

  speech.parseWav('uploads/audio.wav', function(text) {
    console.log(text);
    controlDrone(text);
  });
});
```

I used the **FFmpeg** utility to downsample the audio and combine it into one channel for uploading to Microsoft:

```
exports.parseWav = function(wavPath, callback) {  
  var cmd = 'ffmpeg -i ' + wavPath + ' -ar 8000 -ac 1 -y  
    tmp.wav';  
  
  exec(cmd, function(error, stdout, stderr) {  
    console.log(stderr); // command output is in stdout  
  });  
  
  postToOxford(callback);  
});
```

While we're at it, we might as well use **Microsoft's text-to-speech API** so the drone can talk back to us!

Autonomous Search Paths

I used the **ardrone-autonomy** library to map out autonomous search paths for my drone. After crashing my drone into the furniture and houseplants one too many times in my living room, my wife nicely suggested I move my project to my garage, where there is less to break—but there isn't much room to maneuver (see **Figure 13-3**).

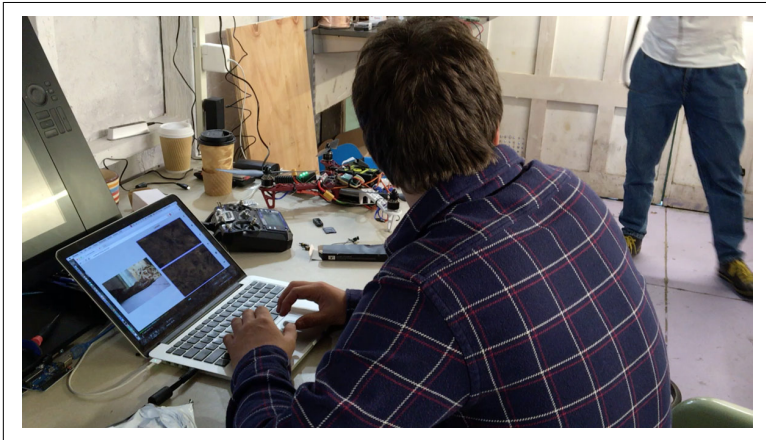


Figure 13-3. Flying the drone in my “lab” (image courtesy of Lukas Biewald)

When I get a bigger lab space, I'll work more on smart searching algorithms, but for now I'll just have my drone take off and rotate, looking for my friends and enemies:

```

var autonomy = require('ardrone-autonomy');
var mission = autonomy.createMission({ip: '10.0.1.3',
frameRate: 1, imageSize: '640:320'});

console.log("Here we go!")

mission.takeoff()
  .zero()    // Sets the current state as the reference
  .altitude(1)
  .taskSync(console.log("Checkpoint 1"))
  .go({x: 0, y: 0, z: 1, yaw: 90})
  .taskSync(console.log("Checkpoint 2"))
  .hover(1000)
  .go({x: 0, y: 0, z: 1, yaw: 180})
  .taskSync(console.log("Checkpoint 3"))
  .hover(1000)
  .go({x: 0, y: 0, z: 1, yaw: 270})
  .taskSync(console.log("Checkpoint 4"));
  .hover(1000)
  .go({x: 0, y: 0, z: 1, yaw: 0}
  .land()

```

Conclusion

Once everything is set up and you are controlling the drone through an API and getting the video feed, hacking on drones becomes incredibly fun. With all of the newly available image recognition technology, there are all kinds of possible uses, from surveying floorplans to painting the walls. The Parrot drone wasn't really designed to fly safely inside a small house like mine, but a more expensive drone might make this a totally realistic application. In the end, drones will become more stable, the price will come down, and the real-world applications will explode.

Microsoft's Cognitive Services cloud APIs are easy to use and amazingly cheap. At first, I was worried that the drone's unusually wide-angle camera might affect the face recognition and that the loud drone propeller might interfere with the speech recognition, but overall the performance was much better than I expected. The latency is less of an issue than I was expecting. Doing the computation in the Cloud on a live image feed seems like a strange architec-

ture at first, but it will probably be the way of the future for a lot of applications.

Lukas Biewald

Lukas Biewald is the founder of **CrowdFlower**. Founded in 2009, CrowdFlower is a data enrichment platform that taps into an on-demand workforce to help companies collect training data and do human-in-the-loop machine learning.

Following his graduation from Stanford University with a BS in mathematics and an MS in computer science, Lukas led the Search Relevance Team for Yahoo! Japan. He then worked as a senior data scientist at Powerset, which was acquired by Microsoft in 2008. Lukas was featured in *Inc.* magazine's 30 Under 30 list.

Lukas is also an expert-level Go player.

Natural Language

Natural language has long been a goal of AI research and development, and 2016 was a banner year for technologies that support the parsing, understanding, and generating of text. Alyona Medelyan takes you right into NLP with strategies for scoping and tackling your project. Michelle Casbon then discusses using Spark MLLib for NLP, and Lior Shkiller looks at vectorization models and architectures to capture semantics.

Three Three Tips for Getting Started with NLU

Alyona Medelyan

What makes a cartoon caption funny? As one algorithm found: a simple readable sentence, a negation, and a pronoun—but not “he” or “she.” **The algorithm** went on to pick the funniest captions for thousands of the *New Yorker*’s cartoons, and in most cases, it matched the intuition of its editors.

Algorithms are getting much better at understanding language, and we are becoming more aware of this through stories like that of **IBM Watson winning the Jeopardy quiz**. Thankfully, large corporations aren’t keeping the latest breakthroughs in natural language understanding (NLU) for themselves. Google released the **word2vec tool**, and Facebook followed by publishing its **speed-optimized deep learning modules**. Since language is at the core of many businesses today, it’s important to understand what NLU *is*, and how you can use it to meet some of your business goals. In this article, you will learn three key tips on how to get into this fascinating and useful field.

But first things first: what does “natural language understanding” actually mean? Whenever a computer understands *written* language—or in other words, derives the meaning of words, sentences, or text—we call it *natural language understanding*. When understanding *spoken* language, such as voice commands or recorded speech, a process called *automatic speech recognition* transforms it first into written words.

NLU is technically a sub-area of the broader area of natural language processing (NLP), which is a sub-area of artificial intelligence (AI). Many NLP tasks, such as part-of-speech or text categorization, do not always require *actual understanding* in order to perform accurately, but in some cases they might, which leads to confusion between these two terms. As a rule of thumb, an algorithm that builds a model that understands meaning falls under natural language understanding, not just natural language processing.

Examples of Natural Language Understanding

Let's look at some examples of what we mean by "understanding meaning," in a nonphilosophical way. For our first example, we'll look at *relation extraction*. The meaning of "London," for example, could be a multitude of relations, such as: "is a City," "is a British capital," "is the location of Buckingham Palace," "is the headquarters of HSBC." These are semantic relations, and NLU algorithms are pretty good at extracting such relations from unstructured text. For example, the **Open Information Extraction system** at the University of Washington extracted more than 500 million such relations from unstructured web pages by analyzing sentence structure. Another example is Microsoft's **ProBase**, which uses *syntactic patterns* ("is a," "such as") and resolves ambiguity through iteration and statistics. It then merges all these relations into knowledge taxonomy. Similarly, businesses can extract knowledge bases from web pages and documents relevant to their business.

Meaning can also be expressed through emotions. *Sentiment analysis* can determine emotions from text. For businesses, it's important to know the sentiment of their users and customers overall, and the sentiment attached to specific themes, such as areas of customer service or specific product features.

Another popular application of NLU is chat bots, also known as *dialogue agents*, which make our interaction with computers more human-like. At the most basic level, bots need to understand how to map our words into actions and use dialogue to clarify uncertainties. At the most sophisticated level, they should be able to hold a conversation about anything, which is true artificial intelligence. Anybody who has used Siri, Cortana, or Google Now while driving will attest that dialogue agents are already proving useful, and going beyond their current level of understanding would not necessarily improve

their function. Most other bots out there are nothing more than a natural language interface into an app that performs one specific task, such as shopping or meeting scheduling. Interestingly, this is already so technologically challenging that humans often **hide behind the scenes**. Keeping it simple is the key when it comes to building bots.

Begin Using NLU—Here's Why and How

The good news is that despite many challenges, **NLU is breaking out**. This means that whether you work with product reviews, receive user feedback, or interact with customers, you can start using NLU methods today. Here are three tips for how and why to get started:

You can choose the smartest algorithm out there without having to pay for it

Most algorithms are publicly available as open source. It's truly mind-bending that if you want, you can download and start using the same algorithms Google used to beat the world's **Go champion**, right now. Many machine learning toolkits come with an array of algorithms; which one is the best depends on what you are trying to predict and the amount of data available. While there may be some **general guidelines**, it's often best to **loop through them** to choose the right one.

Your deep knowledge of the business is more important than the best algorithm

When integrating NLU into a new product or service, it is more important to understand the specific business field—how it works and its priorities—than to have the best algorithm at hand. Consider this example: **an app that lets you query Salesforce in natural language**. Knowing which questions users are likely to ask and which data can be queried from Salesforce is more important than having the most accurate language parser. After all, it's more important to solve the right problem with an OK algorithm than the wrong problem with the best algorithm out there.

It's likely that you already have enough data to train the algorithms

Google may be the most prolific producer of successful NLU applications. The reason why its search, machine translation, and ad recommendation work so well is because Google has

access to huge data sets. For the rest of us, current algorithms like **word2vec** require significantly less data to return useful results. Indeed, companies have already started integrating such tools into their **workflows**. If your business has as a few thousand product reviews or user comments, you can probably make this data work for you using word2vec, or other language modeling methods available through tools like **Gensim**, **Torch**, and **TensorFlow**.

Judging the Accuracy of an Algorithm

These are all good reasons for giving natural language understanding a go, but how do you know if the *accuracy* of an algorithm will be sufficient? Consider the type of analysis it will need to perform and the breadth of the field. Analysis ranges from shallow, such as word-based statistics that ignore word order, to deep, which implies the use of ontologies and parsing. Deep learning, despite the name, does *not* imply a deep analysis, but it does make the traditional shallow approach deeper. *Field* stands for the application area, and *narrow* means a specialist domain or a specific task. *Broad* implies no restrictions on how varied language may be.

As a rule of thumb, try to classify your problem according to this grid:

Shallow analysis, narrow field

Sample problem: determining the level of frustration in support tickets of a specific company

Deep analysis, narrow field

Sample problem: scheduling appointments automatically through email

Shallow analysis, broad field

Sample problem: categorizing any social media comments into happy and sad

Deep analysis, broad field

Sample problem: a conversation about anything

If accuracy is paramount, go only for specific tasks that need shallow analysis. If accuracy is less important, or if you have access to people who can help where necessary, deepening the analysis or a broader field may work. In general, when accuracy is important, stay away from cases that require deep analysis of varied language—this is an area still under development in the field of AI.

Alyona Medelyan

Alyona Medelyan runs **Entopix**, a successful international NLP consultancy. Alyona has extensive experience working with customer feedback data, such as surveys, social media data, call center logs, and public forums. She is also CEO and cofounder of **The-matic**, a customer insight startup. Alyona's PhD was in keyword extraction, which led to the open source Maui toolkit, her role as chief research officer at Pingar, and subsequent work consulting on NLP for large multinationals.

Training and Serving NLP Models Using Spark

Michelle Casbon

NOTE

Author's Note

This article describes a framework we built to organize, construct, and serve predictive models. It was used by production systems in a variety of different industries, and while the larger system is no longer operational, the component that this article focuses on is open source and can be found on [GitHub](#).

Identifying critical information out of a sea of unstructured data or customizing real-time human interaction are a couple of examples of how clients utilize our technology at [Idibon](#)—a San Francisco startup focusing on [natural language processing](#) (NLP). The machine learning libraries in Spark ML and MLlib have enabled us to create an adaptive machine intelligence environment that analyzes text in any language, at a scale far surpassing the number of words per second in the Twitter firehose.

Our engineering team has built a platform that trains and serves thousands of NLP models that function in a distributed environment. This allows us to scale out quickly and provide thousands of predictions per second for many clients simultaneously. In this post, we'll explore the types of problems we're working to resolve, the processes we follow, and the technology stack we use. This should be

helpful for anyone looking to build out or improve their own NLP pipelines.

Constructing Predictive Models with Spark

Our clients are companies that need to automatically classify documents or extract information from them. This can take many diverse forms, including social media analytics, message categorization and routing of customer communications, newswire monitoring, risk scoring, and automating inefficient data entry processes. All of these tasks share a commonality: *the construction of predictive models trained on features extracted from raw text* (Figure 15-1). This process of creating NLP models represents a unique and challenging use case for the tools provided by Spark.

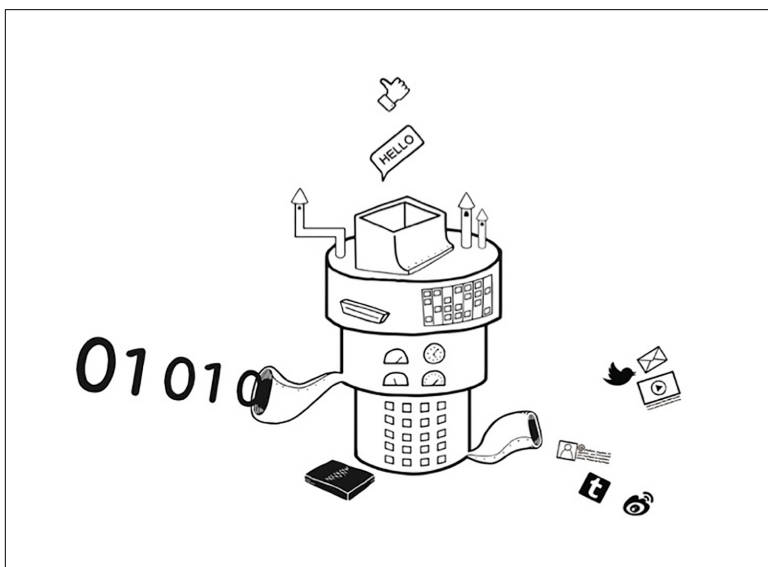


Figure 15-1. Creating NLP models (image courtesy of Idibon)

The Process of Building a Machine Learning Product

A machine learning product can be broken down into three conceptual pieces (see Figure 15-2): *the prediction* itself, *the models* that provide the prediction, and *the data set* used to train the models.

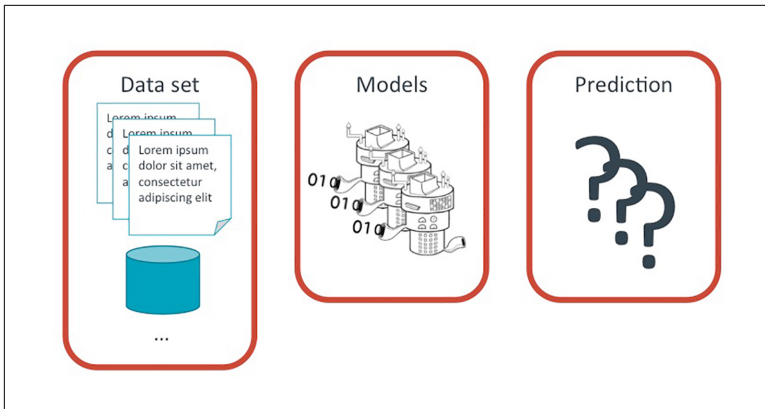


Figure 15-2. Building a machine-learning product (image courtesy of Michelle Casbon)

Prediction

In our experience, it's best to begin with business questions and use them to drive the selection of data sets, rather than having data sets themselves drive project goals. If you do begin with a data set, it's important to connect data exploration with critical business needs as quickly as possible. With the right questions in place, it becomes straightforward to choose useful classifications, which is what a prediction ultimately provides.

Data Set

Once the predictions are defined, it becomes fairly clear which data sets would be most useful. It is important to verify that the data you have access to can support the questions you are trying to answer.

Model Training

Having established the task at hand and the data to be used, it's time to worry about the models. In order to generate models that are accurate, we need training data, which is often generated by humans. These humans may be experts within a company or consulting firm, or in many cases, they may be part of a network of analysts.

Additionally, many tasks can be done efficiently and inexpensively by using a crowdsourcing platform like [CrowdFlower](#). We like the

platform because it categorizes workers based on specific areas of expertise, which is particularly useful for working with languages other than English.

All of these types of workers submit annotations for specific portions of the data set in order to generate training data. The training data is what you'll use to make predictions on new or remaining parts of the data set. Based on these predictions, you can make decisions about the *next* set of data to send to annotators. The point here is to make the best models with the fewest human judgements. You continue iterating between model training, evaluation, and annotation—getting higher accuracy with each iteration. We refer to this process as *adaptive learning*, which is a quick and cost-effective means of producing highly accurate predictions.

Operationalization

To support the adaptive learning process, we built a platform that *automates* as much as possible. Having components that *autoscale without our intervention* is key to supporting a real-time API with fluctuating client requests. A few of the tougher scalability challenges we've addressed include:

- Document storage
- Serving up thousands of *individual* predictions per second
- Support for continuous training, which involves automatically generating updated models whenever the set of training data or model parameters change
- Hyperparameter optimization for generating the most performant models

We do this by using a combination of components within the AWS stack, such as [Elastic Load Balancing](#), [Auto Scaling groups](#), [RDS](#), and [ElastiCache](#). There are also a number of metrics that we monitor within [New Relic](#) and [Datadog](#), which alert us before things go terribly awry.

[Figure 15-3](#) is a high-level diagram of the main tools in our infrastructure.

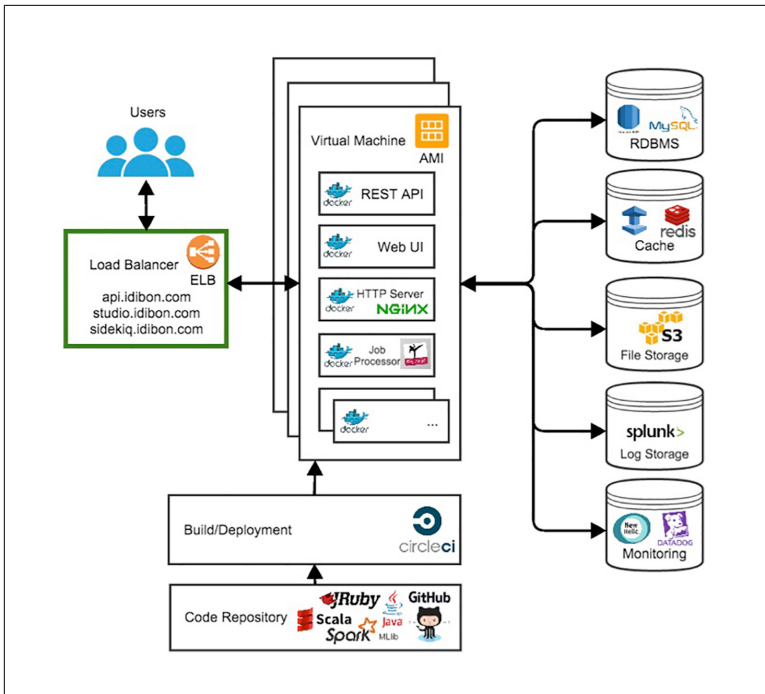


Figure 15-3. Main tools (image courtesy of Michelle Casbon)

Spark's Role

A core component of our machine learning capabilities is the optimization functionality within Spark ML and MLlib. Making use of these for NLP purposes involves the addition of a *persistence layer* that we refer to as IdiML. This allows us to utilize Spark for individual predictions, rather than its most common usage as a platform for processing large amounts of data all at once.

What Are We Using Spark For?

At a more detailed level, there are three main components of an NLP pipeline:

Feature extraction

Text is converted into a numerical format appropriate for statistical modeling.

Training

Models are generated based on the classifications provided for each feature vector.

Prediction

Trained models are used to generate a classification for new, unseen text.

A simple example of each component is described in [Figure 15-4](#).

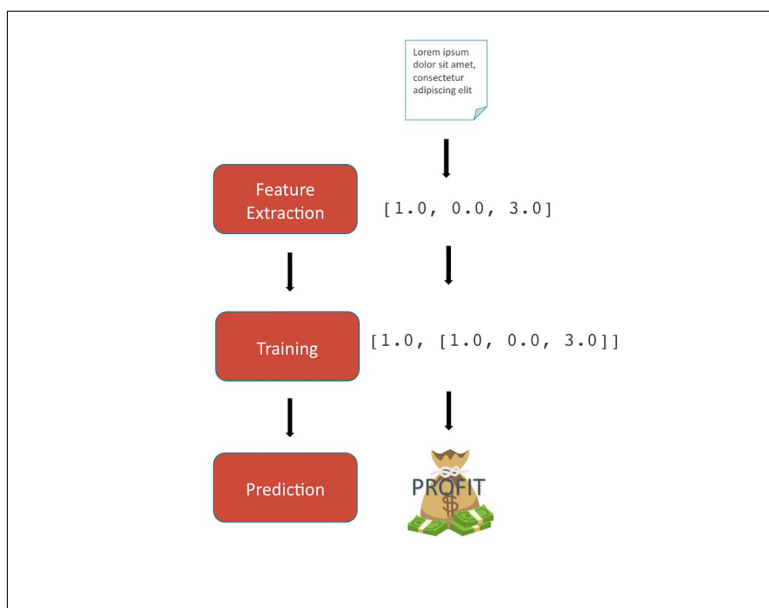


Figure 15-4. Core component of our machine learning capabilities (image courtesy of Michelle Casbon)

Feature Extraction

In the feature extraction phase, text-based data is transformed into numerical data in the form of a *feature vector*. This vector represents the unique characteristics of the text and can be generated by any sequence of mathematical transformations. Our system was built to easily accommodate additional feature types, such as features derived from deep learning; but for simplicity's sake, we'll consider a basic feature pipeline example ([Figure 15-5](#)):

Input

A single document, consisting of content and perhaps metadata.

Content extraction

Isolates the portion of the input that we're interested in, which is usually the content itself.

Tokenization

Separates the text into individual words. In English, a token is more or less a string of characters with whitespace or punctuation around them, but in other languages like Chinese or Japanese, you need to probabilistically determine what a “word” is.

Ngrams

Generates sets of word sequences of length n. Bigrams and trigrams are frequently used.

Feature lookup

Assigns an arbitrary numerical index value to each unique feature, resulting in a vector of integers. This feature index is stored for later use during prediction.

Output

A numerical feature vector in the form of Spark MLlib's Vector data type (`org.apache.spark.mllib.linalg.Vector`).

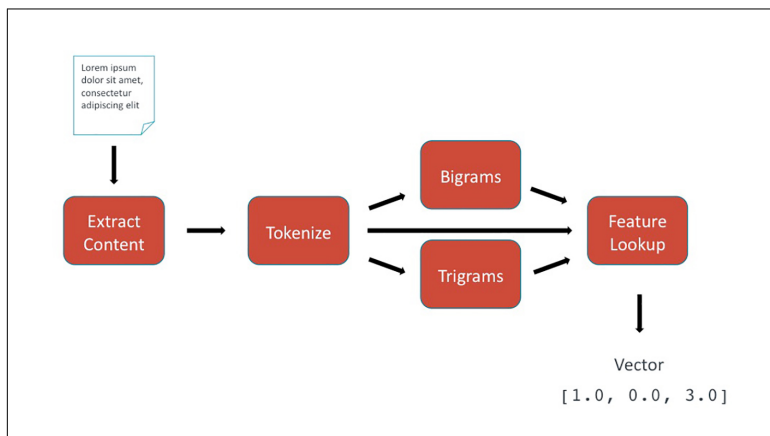


Figure 15-5. Feature extraction (image courtesy of Michelle Casbon)

Training

During the training phase (Figure 15-6), a classification is appended to the feature vector. In Spark, this is represented by the Labeled Point data type. In a binary classifier, the classification is either true or false (1.0 or 0.0):

1. Input: numerical feature Vectors.
2. A `LabeledPoint` is created, consisting of the feature vector and its classification. This classification was generated by a human earlier in the project lifecycle.
3. The set of `LabeledPoints` representing the full set of training data is sent to the `LogisticRegressionWithLBFGS` function in MLlib, which fits a model based on the given feature vectors and associated classifications.
4. Output: a `LogisticRegressionModel`.

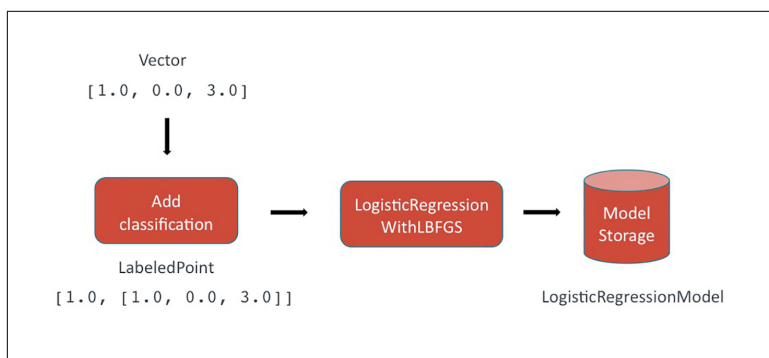


Figure 15-6. Training phase (image courtesy of Michelle Casbon)

Prediction

At prediction time, the models generated during training are used to provide a classification for the new piece of text. A *confidence interval* of 0-1 indicates the strength of the model's confidence in the prediction. The higher the confidence, the more certain the model is. The following components encompass the prediction process (Figure 15-7):

1. Input: unseen document in the same domain as the data used for training.
2. The same featurization pipeline is applied to the unseen text. The feature index generated during training is used here as a lookup. This results in a feature vector in the same feature space as the data used for training.
3. The trained model is retrieved.
4. The feature Vector is sent to the model, and a classification is returned as a prediction.

5. The classification is interpreted in the context of the specific model used, which is then returned to the user.
6. Output: a predicted classification for the unseen data and a corresponding confidence interval.

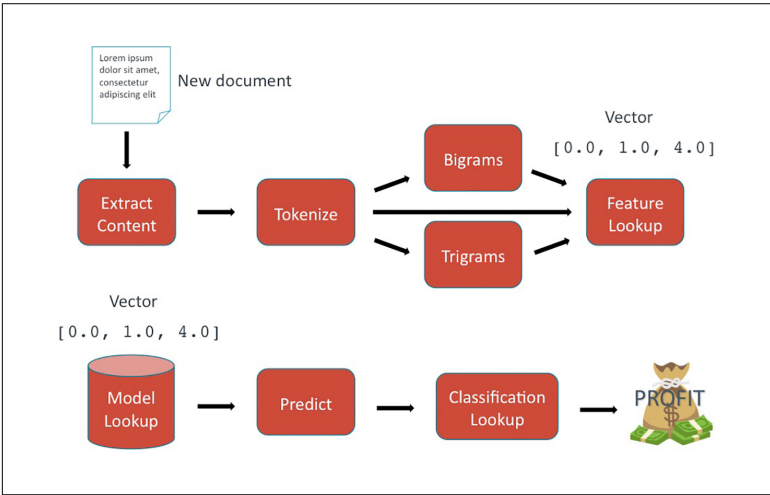


Figure 15-7. The prediction process (image courtesy of Michelle Casbon)

Prediction Data Types

In typical Spark ML applications, predictions are mainly generated using RDDs and DataFrames: the application loads document data into one column, and MLlib places the results of its prediction in another. Like all Spark applications, these prediction jobs may be distributed across a cluster of servers to efficiently process petabytes of data. However, our most demanding use case is exactly the opposite of big data: often, we must analyze a single, short piece of text and return results as quickly as possible, ideally within a millisecond.

Unsurprisingly, DataFrames are not optimized for this use case, and our initial DataFrame-based prototypes fell short of this requirement.

Fortunately for us, MLlib is implemented using an efficient linear algebra library, and all of the algorithms we planned to use included internal methods that generated predictions using single Vector objects without any added overhead. These methods looked perfect

for our use case, so we designed IdiML to be extremely efficient at converting single documents to single Vectors so that we could use Spark MLib's internal Vector-based prediction methods.

For a single prediction, we observed speed improvements of up to two orders of magnitude by working with Spark MLib's Vector type as opposed to RDDs. The speed differences between the two data types are most pronounced among smaller batch sizes. This makes sense considering that RDDs were designed for processing large amounts of data. In a real-time web server context such as ours, small batch sizes are by far the most common scenario. Since distributed processing is already built into our web server and load-balancer, the distributed components of core Spark are unnecessary for the small-data context of individual predictions. As we learned during the development of IdiML and have shown in [Figure 15-8](#), Spark MLib is an incredibly useful and performant machine learning library for low-latency and real-time applications. Even the worst-case IdiML performance is capable of performing sentiment analysis on every tweet written, in real time, from a mid-range consumer laptop ([Figure 15-9](#)).

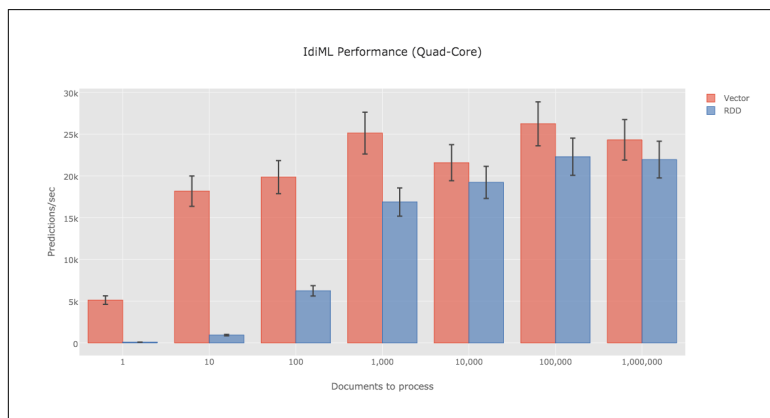


Figure 15-8. Measurements performed on a mid-2014 15-inch MacBook Pro Retina—the large disparity in single-document performance is due to the inability of the test to take advantage of the multiple cores (image courtesy of Michelle Casbon)

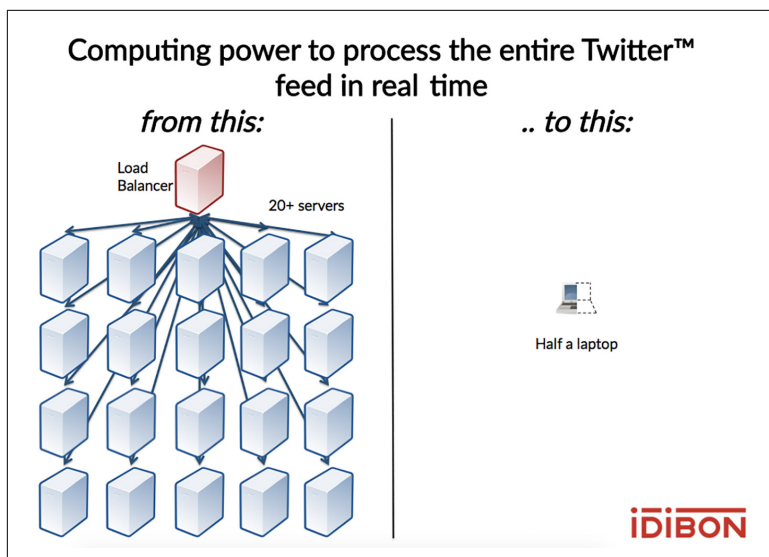


Figure 15-9. Processing power (image courtesy of Rob Munro)

Fitting It into Our Existing Platform with IdiML

In order to provide the most accurate models possible, we want to be able to support different types of machine learning libraries. Spark has a unique way of doing things, so we want to insulate our main code base from any idiosyncrasies. This is referred to as a *per-sistence layer* (IdiML) (Figure 15-10), which allows us to combine Spark functionality with NLP-specific code that we've written ourselves. For example, during hyperparameter tuning we can train models by combining components from both Spark and our own libraries. This allows us to automatically choose the implementation that performs best for each model, rather than having to decide on just one for all models.

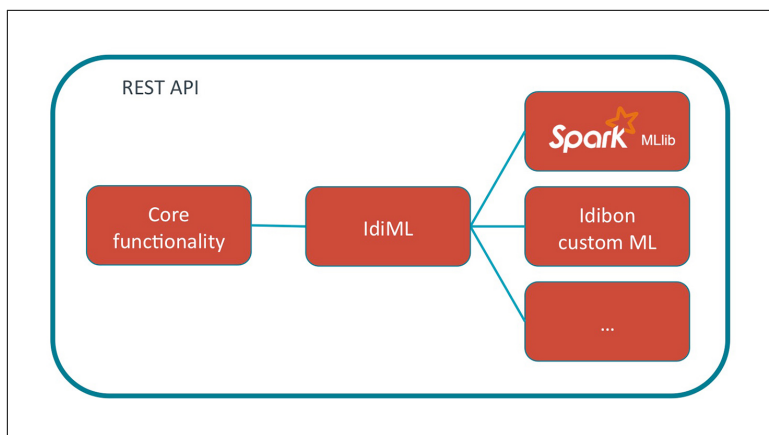


Figure 15-10. Persistence layer (image courtesy of Michelle Casbon)

Why a Persistence Layer?

The use of a persistence layer allows us to operationalize the training and serving of many thousands of models. Here's what IdiML provides us with:

A means of storing the parameters used during training.

This is necessary in order to return the corresponding prediction.

The ability to version control every part of the pipeline.

This enables us to support backward compatibility after making updates to the code base. Versioning also refers to the ability to recall and support previous iterations of models during a project's lifecycle.

The ability to automatically choose the best algorithm for each model.

During hyperparameter tuning, implementations from different machine learning libraries are used in various combinations and the results evaluated.

The ability to rapidly incorporate new NLP features

Standardizing the developer-facing components. This provides an insulation layer that makes it unnecessary for our feature engineers and data scientists to learn how to interact with a new tool.

The ability to deploy in any environment.

We are currently using Docker containers on EC2 instances, but our architecture means that we can also take advantage of the *burst capabilities* that services such as **Amazon Lambda** provide.

A single save and load framework

Based on generic `InputStreams` & `OutputStreams`, which frees us from the requirement of reading and writing to and from disk.

A logging abstraction in the form of `slf4j`

Insulates us from being tied to any particular framework.

Faster, Flexible Performant Systems

NLP differs from other forms of machine learning because it operates directly on human-generated data. This is often messier than machine-generated data, since language is inherently ambiguous, which results in highly variable interpretability—even among humans. Our goal is to automate as much of the NLP pipeline as possible so that resources are used more efficiently: machines help humans help machines help humans. To accomplish this across language barriers, we're using tools such as Spark to build performant systems that are faster and more flexible than ever before.

Michelle Casbon

Michelle Casbon was a senior data science engineer at Idibon, where she is contributing to the goal of bringing language technologies to all the world's languages. Her development experience spans a decade across various industries, including media, investment banking, healthcare, retail, and geospatial services. Michelle completed a master's at the University of Cambridge, focusing on NLP, speech recognition, speech synthesis, and machine translation. She loves working with open source technologies and has had a blast contributing to the Apache Spark project.

Capturing Semantic Meanings Using Deep Learning

Lior Shkiller

Word embedding is a technique that treats words as vectors whose relative similarities correlate with semantic similarity. This technique is one of the most successful applications of unsupervised learning. *Natural language processing (NLP)* systems traditionally encode words as strings, which are arbitrary and provide no useful information to the system regarding the relationships that may exist between different words. Word embedding is an alternative technique in NLP, whereby words or phrases from the vocabulary are mapped to vectors of real numbers in a low-dimensional space relative to the vocabulary size, and the similarities between the vectors correlate with the words' semantic similarity.

For example, let's take the words *woman*, *man*, *queen*, and *king*. We can get their vector representations and use basic algebraic operations to find semantic similarities. Measuring similarity between vectors is possible using measures such as cosine similarity. So, when we subtract the vector of the word *man* from the vector of the word *woman*, then its cosine distance would be close to the distance between the word *queen* minus the word *king* (see [Figure 16-1](#)):

$$W(\text{"woman"}) - W(\text{"man"}) \approx W(\text{"queen"}) - W(\text{"king"})$$

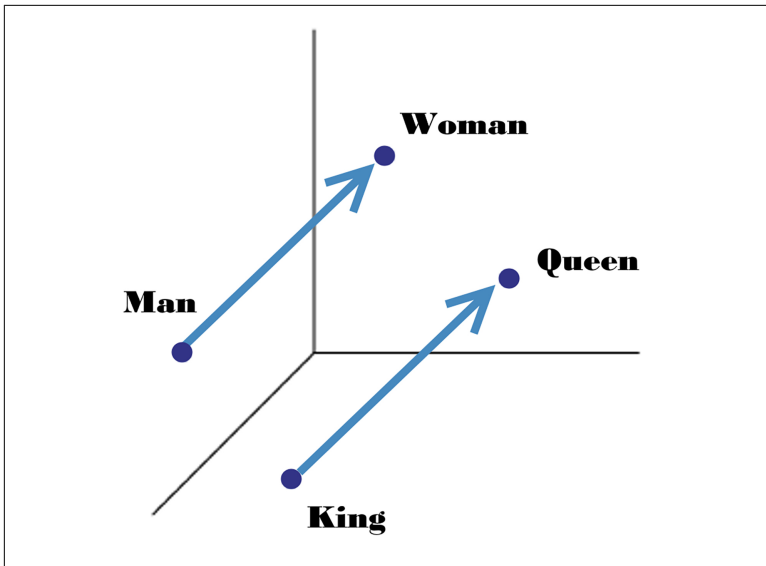


Figure 16-1. Gender vectors (image courtesy of Lior Shkiller)

Many different types of models were proposed for representing words as continuous vectors, including **latent semantic analysis (LSA)** and **latent Dirichlet allocation (LDA)**. The idea behind those methods is that words that are related will often appear in the same documents. For instance, *backpack*, *school*, *notebook*, and *teacher* are probably likely to appear together. But *school*, *tiger*, *apple*, and *basketball* would probably not appear together consistently. To represent words as vectors—using the assumption that similar words will occur in similar documents—LSA creates a matrix whereby the rows represent unique words and the columns represent each paragraph. Then, LSA applies **singular value decomposition (SVD)**, which is used to reduce the number of rows while preserving the similarity structure among columns. The problem is that those models become computationally very expensive on large data.

Instead of computing and storing large amounts of data, we can try to create a neural network model that will be able to learn the relationship between the words and do it efficiently.

Word2Vec

The most popular word embedding model is **word2vec**, created by **Mikolov et al.** in 2013. The model showed great results and improvements in efficiency. Mikolov et al. presented the negative-sampling approach as a more efficient way of deriving word embeddings. You can read more about it in **Goldberg and Levy’s “word2vec Explained”**.

The model can use either of two architectures to produce a distributed representation of words: **continuous bag-of-words (CBOW)** or **continuous skip-gram**.

We’ll look at both of these architectures next.

The CBOW Model

In the CBOW architecture, the model predicts the current word from a window of surrounding context words. Mikolov et al. thus use both the n words before and after the target word w to predict it.

A sequence of words is equivalent to a set of items. Therefore, it is also possible to replace the terms *word* and *item*, which allows for applying the same method for **collaborative filtering** and **recommender systems**. CBOW is several times faster to train than the skip-gram model and has slightly better accuracy for the words that appear frequently (see **Figure 16-2**).

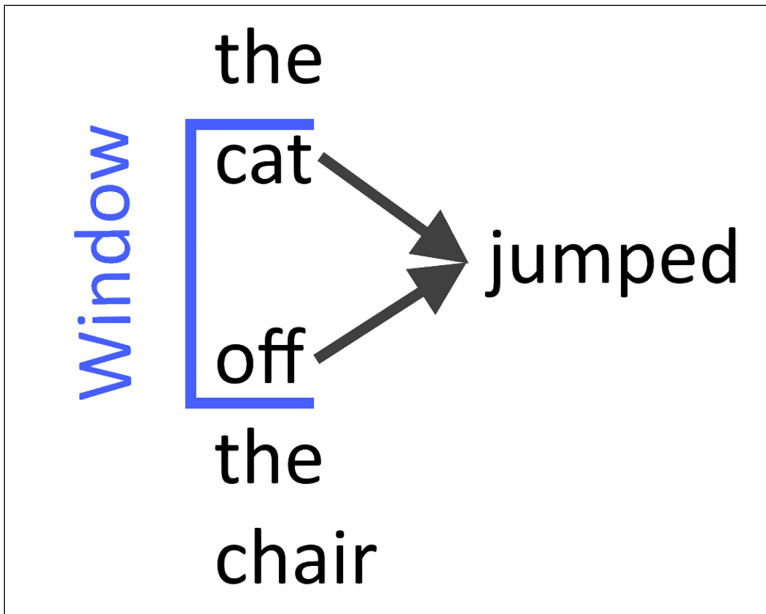


Figure 16-2. Predicting the word given its context (image courtesy of Lior Shkiller)

The Continuous Skip-Gram Model

In the skip-gram model, instead of using the surrounding words to predict the center word, it uses the center word to predict the surrounding words (see [Figure 16-3](#)). According to Mikolov et al. skip-gram works well with a small amount of the training data and does a good job of representing even rare words and phrases.

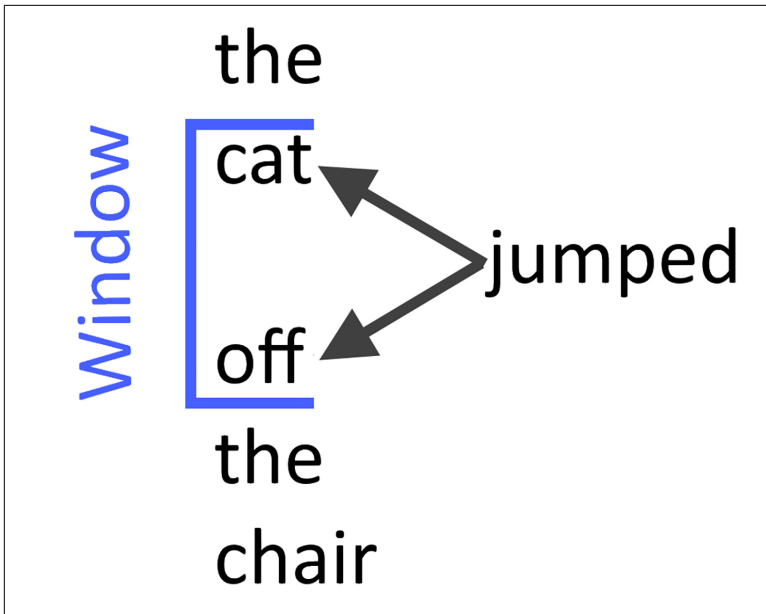


Figure 16-3. Predicting the context given a word (image courtesy of Lior Shkiller)

Coding an Example

(You can find the code for the following example at this [GitHub repo](#).)

The great thing about this model is that it works well *for many languages*.

All we have to do is download a big data set for the language that we need.

Looking to Wikipedia for a Big Data Set

We can look to Wikipedia for any given language. To obtain a big data set, follow these steps:

1. Find the [ISO 639 code](#) for your desired language
2. Go to the [Wikimedia Downloads page](#) and find the latest complete dump.
3. From the latest dump, download `wiki-latest-pages-articles.xml.bz2`

Next, to make things easy, we will install **gensim**, a Python package that implements word2vec:

```
pip install --upgrade gensim
```

We need to create the corpus from Wikipedia, which we will use to train the word2vec model. The output of the following code is “wiki..text”—which contains all the words of all the articles in Wikipedia, segregated by language:

```
from gensim.corpora import WikiCorpus

language_code = "he"
inp = language_code+"wiki-latest-pages-articles.xml.bz2"
outp = "wiki.{}.text".format(language_code)
i = 0

print("Starting to create wiki corpus")
output = open(outp, 'w')
space = " "
wiki = WikiCorpus(inp, lemmatize=False, dictionary={})
for text in wiki.get_texts():
    article = space.join([t.decode("utf-8") for t in text])

    output.write(article + "\n")
    i = i + 1
    if (i % 1000 == 0):
        print("Saved " + str(i) + " articles")

output.close()
print("Finished - Saved " + str(i) + " articles")
```

Training the Model

The parameters are as follows:

size

The dimensionality of the vectors—bigger *size* values require more training data but can lead to more accurate models.

window

The maximum distance between the current and predicted word within a sentence.

min_count

Ignore all words with total frequency lower than this.

```
import multiprocessing
from gensim.models import Word2Vec
```

```

from gensim.models.word2vec import LineSentence

language_code = "he"
inp = "wiki.{}.text".format(language_code)
out_model = "wiki.{}.word2vec.model".format(language_code)
size = 100
window = 5
min_count = 5

start = time.time()

model = Word2Vec(LineSentence(inp), sg = 0, # 0=CBOW , 1=
SkipGram
                size=size, window=window, min_count=min_count,
                workers=multiprocessing.cpu_count())

# trim unneeded model memory = use (much) less RAM
model.init_sims(replace=True)

print(time.time()-start)

model.save(out_model)

```

Training word2vec took 18 minutes.

fastText

Facebook's Artificial Intelligence Research (FAIR) lab recently released **fastText**, a library that is based on the work reported in the paper “**Enriching Word Vectors with Subword Information**,” by Bojanowski et al. fastText is different from word2vec in that each word is represented as a bag-of-character n -grams. A vector representation is associated with each character n -gram, and words are represented as the *sum of these representations*.

Using Facebook's new library is easy:

```
pip install fasttext
```

The first thing to do is train the model:

```

start = time.time()

language_code = "he"
inp = "wiki.{}.text".format(language_code)
output = "wiki.{}.fasttext.model".format(language_code)
model = fasttext.cbow(inp,output)

print(time.time()-start)

```

Training fastText's model took 13 minutes.

Evaluating Embeddings: Analogies

Next, let's evaluate the models by testing them on our previous example:

$$W(\text{"woman"}) \approx W(\text{"man"}) + W(\text{"queen"}) - W(\text{"king"})$$

The following code first computes the weighted average of the positive and negative words.

After that, it calculates the dot product between the vector representation of all the test words and the weighted average.

In our case, the test words are the entire vocabulary. At the end, we print the word that had the highest cosine similarity with the weighted average of the positive and negative words:

```
import numpy as np
from gensim.matutils import unitvec

def test(model, positive, negative, test_words):

    mean = []
    for pos_word in positive:
        mean.append(1.0 * np.array(model[pos_word]))

    for neg_word in negative:
        mean.append(-1.0 * np.array(model[neg_word]))

    # compute the weighted average of all words
    mean = unitvec(np.array(mean).mean(axis=0))

    scores = {}
    for word in test_words:

        if word not in positive + negative:

            test_word = unitvec(np.array(model[word]))

            # Cosine Similarity
            scores[word] = np.dot(test_word, mean)

    print(sorted(scores, key=scores.get, reverse=True)[:1])
```

Next, we want to test our original example on fastText and gensim's word2vec:

```

positive_words = ["queen", "man"]

negative_words = ["king"]

# Test Word2vec
print("Testing Word2vec")
model = word2vec.getModel()
test(model, positive_words, negative_words, model.vocab)

# Test Fasttext
print("Testing Fasttext")
model = fasttxt.getModel()
test(model, positive_words, negative_words, model.words)

```

Results

```

Testing Word2vec
['woman']
Testing Fasttext
['woman']

```

These results mean that the process works both on fastText and gensim's word2vec!

$$W(\text{"woman"}) \approx W(\text{"man"}) + W(\text{"queen"}) - W(\text{"king"})$$

And as you can see, the vectors actually capture the semantic relationship between the words.

The ideas presented by the models that we described can be used for many different applications, allowing businesses to **predict the next applications they will need**, **perform sentiment analysis**, **represent biological sequences**, **perform semantic image searches**, and more.

Lior Shkiller

Lior Shkiller is the cofounder of **Deep Solutions**. He is a machine learning practitioner and is passionate about AI and cognitive science. He has a degree in computer science and psychology from Tel Aviv University and has more than 10 years of experience in software development.

Deep Solutions delivers end-to-end software solutions based on deep learning innovative algorithms for computer vision, natural language processing, anomaly detection, recommendation systems, and more.

Use Cases

This section analyzes two of the leading-edge use cases for artificial intelligence: chat—should we say “discussion”?—bots and autonomous vehicles. First, Jon Bruner summarizes the development and current state of the bot ecosystem—and offers insight into what’s coming over the horizon. Shaoshan Liu then takes us deep into the current sensing, computing, and computation architecture for autonomous vehicles.

Bot Thots

Jon Bruner

Bots have become hot, fast. Their rise—fueled by advances in artificial intelligence, consumer comfort with chat interfaces, and a **stag-nating mobile app ecosystem**—has been a **bright spot** in an otherwise darkening venture-capital environment.

I've been speaking with a lot of bot creators and have noticed that a handful of questions appear frequently. On closer inspection, bots seem a little less radical and a lot more feasible.

Text Isn't the Final Form

The first generation of bots has been text most of the way down. That's led to some skepticism: *you mean I'll have to choose between 10 hotels by reading down a list in Facebook Messenger?* But bot thinkers are already moving toward a more nuanced model in which different parts of a transaction are handled in text and in graphical interfaces.

Conversational interfaces can be good for discovering intent: a bot that can offer any coherent response to “find a cool hotel near Google's HQ” will be valuable, saving its users one search to find the location of Google's headquarters, another search for hotels nearby, and some amount of filtering to find hotels that are “cool.”

But, conversational interfaces are bad at presenting dense information in ways that are easy for human users to sort through. Suppose that hotel bot turns up a list of finalists and asks you to choose: that's

handled much more effectively in a more traditional-looking web interface, where information can be conveyed richly.

Conversational interfaces are also bad at replacing most kinds of web forms, like the **pizza-ordering bot** that has ironically become an icon of the field. Better to discern intent (“I want a pizza fast”) and then kick the user to a traditional web form, perhaps one that’s already pre-filled with some information gleaned from the conversational process.

A few people have pointed out that one of WeChat’s killer features is that every business has its phone number listed on its profile; once a transaction becomes too complex for messaging, the customer falls back on a phone call. In the US, that fallback is likely to be a GUI, to which you’ll be bounced if your transaction gets to a point where messaging isn’t the best medium.

Discovery Hasn’t Been Solved Yet

Part of the reason we’re excited about bots is that the app economy has stagnated: “the 20 most successful developers grab nearly half of all revenues on Apple’s app store,” notes the *Economist*. It’s hard for users to discover new apps from among the millions that already exist, and the app-installation process involves considerable friction. So, the reasoning goes, bots will be great because they offer a way to skip the stagnant app stores and offer a smoother “installation” process that’s as simple as messaging a new contact.

Of course, now we’ve got new app stores like **Slack’s App Directory**. Users are still likely to discover new bots the way they discover apps: by word of mouth, or by searching for a bot associated with a big brand.

The next step, then, would be to promote bots in response to expressions of intention: in its most intrusive implementation, you’d ask your coworkers on Slack if they want to get lunch, and Slack would suggest that you install the GrubHub bot. Welcome back **Clippy**, now able to draw from the entire internet in order to annoy you.

That particular example is universally condemned, and anything that annoying would drive away its users immediately, but the community is looking for ways to listen for clear statements of intent and integrate bot discovery somehow, in a way that’s valuable for users and not too intrusive.

Platforms, Services, Commercial Incentives, and Transparency

Conversational platforms will have insight into what users might want at a particular moment, and they'll be tempted to monetize these very valuable intent hooks. Monetization here will take place in a very different environment from the web-advertising environment we're used to.

Compared to a chat bot's output, a Google results page is an explosion of information—10 organic search results with titles and descriptions, a bunch of ads flagged as such, and prompts to modify the search by looking for images, news articles, and so on.

A search conducted through a bot is likely to return a “black box” experience: far fewer results, with less information about each. That's especially true of voice bots—and especially, especially true of voice bots without visual interfaces, like Amazon's Alexa.

In this much slower and more constrained search environment, users are more likely to accept the bot's top recommendation rather than to dig through extended results (indeed, this is a feature of many bots), and there's less room to disclose an advertising relationship.

Amazon is also an interesting example in that it's both a bot platform and a service provider. And it has reserved the best namespace for itself; if Amazon decides to offer a ridesharing service (doubtless after noticing that ridesharing is a popular application through Alexa), it will be summoned up by saying “Alexa, call a car.” Uber will be stuck with “Alexa, tell Uber to call a car.”

Compared to other areas, like web search, the messaging-platform ecosystem is remarkably fragmented and competitive. That probably won't last long, though, as messaging becomes a bigger part of communication and personal networks tend to pull users onto consolidated platforms.

How Important Is Flawless Natural Language Processing?

Discovery of functionality within bots is the other big discovery challenge, and one that's also being addressed by interfaces that blend conversational and graphical approaches.

Completely natural language was a dead end in search engines—just ask Jeeves. It turned out that, presented with a service that provided enough value, ordinary users were willing to adapt their language. We switch between different grammars and styles all the time, whether we're communicating with a computer or with other people. “Would you like to grab lunch?” in speech flows seamlessly into “best burrito downtown sf cheap” in a search bar to “getting lunch w pete, brb” in an IM exchange.

The first killer bot may not need sophisticated NLP in order to take off, but it still faces the challenge of educating its users about its input affordances. A blank input box and blinking cursor are hard to overcome in an era of short attention spans.

Siri used a little bit of humor, combined with a massive community of obsessed Apple fans bent on discovering all of its quirks, to publicize its abilities. Most bots don't have the latter, and the former is difficult to execute without Apple's resources. Even with the advantages of size and visibility, Apple still hasn't managed to get the bulk of its users to move beyond Siri's simplest tasks, like setting alarms.

Developers should give a great deal of thought to why alarm-setting is such a compelling use case for Siri: saying “set an alarm for 7:30” slices through several layers of menus and dialogues, *and* it's a natural phrase that's easily parsed into input data for the alarm app. Contrast that with the pizza-ordering use case, where you're prompted for the type of pizza you want, prompted again for your address, prompted again for your phone number, etc.—far more separate prompts than you'd encounter in an ordinary pizza-ordering web form.

Another challenge: overcoming early features that didn't work well. We've all gotten used to web software that starts out buggy and improves over time. But we tend not to notice constant improvement in the same way on bots' sparse interfaces, and we're unwilling

to return to tasks that have failed before—especially if, as bots tend to do, they failed after a long and frustrating exchange.

What Should We Call Them?

There's not much confusion: people working on bots generally call them bots. The field is young, though, and I wonder if the name will stick. Bots usually have negative connotations: spambots, Twitter bots, "are you a bot?", and botnets, to name a few.

"Agent" might be a better option: an agent represents *you*, whereas we tend to think of a bot as representing some sinister *other*. Plus, secret agents and Hollywood agents are cool.

Jon Bruner

Jon Bruner oversees O'Reilly's publications on hardware, the Internet of Things, manufacturing, and electronics. He has been program chair along with Joi Ito of the O'Reilly Solid conference, focused on the intersection between software and the physical world.

Before coming to O'Reilly, he was data editor at *Forbes*, where he combined writing and programming to approach a broad variety of subjects, from the operation of the Columbia River's dams to migration within the United States. He studied mathematics and economics at the University of Chicago and lives in San Francisco, where he can occasionally be found at the console of a pipe organ.

Infographic: The Bot Platforms Ecosystem

Jon Bruner

Behind the recent bot boom are big improvements in artificial intelligence and the rise of ubiquitous messaging services. In **Figure 18-1**, I've listed some of the most important AI, messaging, and bot deployment platforms, and have also highlighted the emergence of a few interesting stacks: Microsoft, Amazon, Google, and Apple each control a general AI agent as well as at least one entry elsewhere in the stack—and Facebook may not be far behind with its M agent.



Figure 18-1. The bot platform ecosystem (full-size version at oreilly.com)

Jon Bruner

Jon Bruner oversees O'Reilly's publications on hardware, the Internet of Things, manufacturing, and electronics, and has been program chair along with Joi Ito of the O'Reilly Solid conference, focused on the intersection between software and the physical world.

Before coming to O'Reilly, he was data editor at Forbes Magazine, where he combined writing and programming to approach a broad variety of subjects, from the operation of the Columbia River's dams to migration within the United States. He studied mathematics and economics at the University of Chicago and lives in San Francisco, where he can occasionally be found at the console of a pipe organ.

Creating Autonomous Vehicle Systems

Shaoshan Liu

We are at the beginning of the future of autonomous driving. What is the landscape and how will it unfold? Let's consult history to help us predict.

Information technology took off in the 1960s, when Fairchild Semiconductors and Intel laid the foundation by producing silicon microprocessors (hence Silicon Valley). Microprocessor technologies greatly improved industrial productivity; the general public had limited access to it. In the 1980s, with the Xerox Alto, Apple Lisa, and later Microsoft Windows, using the graphical user interface (GUI), the second layer was laid, and the vision of having a “personal” computer became a possibility.

With virtually everyone having access to computing power in the 2000s, Google laid the third layer, connecting people—indirectly, with information.

Beginning with Facebook in 2004, social networking sites laid the fourth layer of information technology by allowing people to directly connect with one another, effectively moving human society to the World Wide Web.

As the population of internet-savvy people reached a significant scale, the emergence of Airbnb in 2008, followed by Uber in 2009, and others, laid the fifth layer by providing direct internet commerce services.

Each new layer of information technology, with its added refinements, improved popular access and demand (Figure 19-1). Note that for most internet commerce sites, where they provide access to service providers through the internet, it is humans who are providing the services.

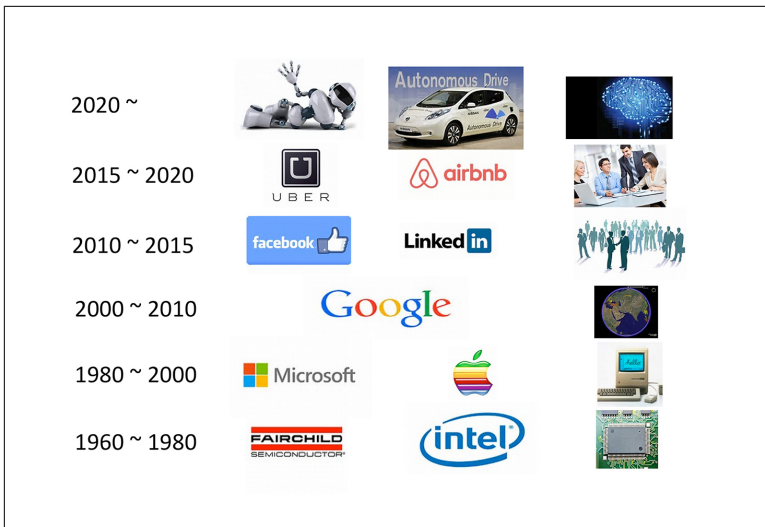


Figure 19-1. The history of information technologies (image courtesy of Shaoshan Liu)

Now we are adding the sixth layer, where robots, rather than humans, provide services.

One example of this is the advent of autonomous vehicles (AVs). Autonomous driving technologies enable self-driving cars to take you to your destination without the involvement of a human driver. It's not one technology, but an integration of many.

In this post, we'll explore the technologies involved in autonomous driving and discuss how to integrate these technologies into a safe, effective, and efficient autonomous driving system.

An Introduction to Autonomous Driving Technologies

Autonomous driving technology is a complex system (as seen in Figure 19-2), consisting of three major subsystems: algorithms, including sensing, perception, and decision; client, including the

robotics operating system and hardware platform; and the cloud platform, including data storage, simulation, high-definition (HD) mapping, and deep learning model training.

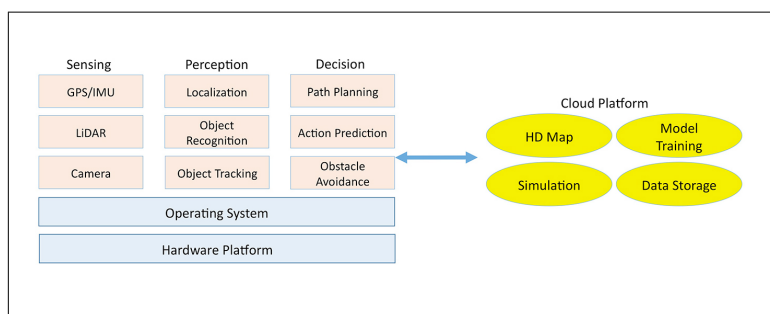


Figure 19-2. Autonomous driving system architecture overview (image courtesy of Shaoshan Liu)

The algorithm subsystem extracts meaningful information from sensor raw data to understand its environment and make decisions about its actions. The client subsystem integrates these algorithms to meet real-time and reliability requirements. (For example, if the sensor camera generates data at 60 Hz, the client subsystem needs to make sure that the longest stage of the processing pipeline takes less than 16 ms to complete.) The cloud platform provides offline computing and storage capabilities for autonomous cars. Using the cloud platform, we are able to test new algorithms and update the HD map—plus train better recognition, tracking, and decision models.

Autonomous Driving Algorithms

The algorithms component consists of sensing and extracting meaningful information from sensor raw data; perception, to localize the vehicle and understand the current environment; and decision, to take actions to reliably and safely reach destinations.

Sensing

Normally, an autonomous vehicle consists of several major sensors. Since each type of sensor presents advantages and drawbacks, the data from multiple sensors must be combined. The sensor types can include the following:

GPS/IMU

The GPS/IMU system helps the AV localize itself by reporting both inertial updates and a global position estimate at a high rate, for example, 200 Hz. While GPS is a fairly accurate localization sensor, at only 10 Hz, its update rate is too slow to provide real-time updates. Now, though an IMU's accuracy degrades with time, and thus cannot be relied upon to provide accurate position updates over long periods, it can provide updates more frequently—at, or higher than, 200 Hz. This should satisfy the real-time requirement. By combining GPS and IMU, we can provide accurate and real-time updates for vehicle localization.

LIDAR

LIDAR is used for mapping, localization, and obstacle avoidance. It works by bouncing a beam off surfaces and measuring the reflection time to determine distance. Due to its high accuracy, it is used as the main sensor in most AV implementations. LIDAR can be used to produce HD maps, to localize a moving vehicle against HD maps, detect obstacle ahead, etc. Normally, a LIDAR unit, such as Velodyne 64-beam laser, rotates at 10 Hz and takes about 1.3 million readings per second.

Cameras

Cameras are mostly used for object recognition and object tracking tasks, such as lane detection, traffic light detection, and pedestrian detection. To enhance AV safety, existing implementations usually mount eight or more cameras around the car, such that we can use cameras to detect, recognize, and track objects in front, behind, and on both sides of the vehicle. These cameras usually run at 60 Hz, and, when combined, generate around 1.8 GB of raw data per second.

Radar and sonar

The radar and sonar system is used for the last line of defense in obstacle avoidance. The data generated by radar and sonar shows the distance from the nearest object in front of the vehicle's path. When we detect that an object is not far ahead and that there may be danger of a collision, the AV should apply the brakes or turn to avoid the obstacle. Therefore, the data generated by radar and sonar does not require much processing and is usually fed directly to the control processor—not through the main computation pipeline—to implement such urgent func-

tions as swerving, applying the brakes, or pre-tensioning the seatbelts.

Perception

Next, we feed the sensor data to the perception subsystem to understand the vehicle's environment. The three main tasks in autonomous driving perception are localization, object detection, and object tracking.

Localization

While GPS/IMU can be used for localization, GPS provides fairly accurate localization results but with a slow update rate; IMU provides a fast update with less accurate results. We can use Kalman filtering to combine the advantages of the two and provide accurate and real-time position updates. As shown in **Figure 19-3**, the IMU propagates the vehicle's position every 5 ms, but the error accumulates as time progresses. Fortunately, every 100 ms we get a GPS update, which helps us correct the IMU error. By running this propagation and update model, we can use GPS/IMU to generate fast and accurate localization results.

Nonetheless, we cannot solely rely on this combination for localization for three reasons:

1. It has an accuracy of only about one meter.
2. The GPS signal has multipath problems, meaning that the signal may bounce off buildings and introduce more noise.
3. GPS requires an unobstructed view of the sky and thus does not work in closed environments such as tunnels.

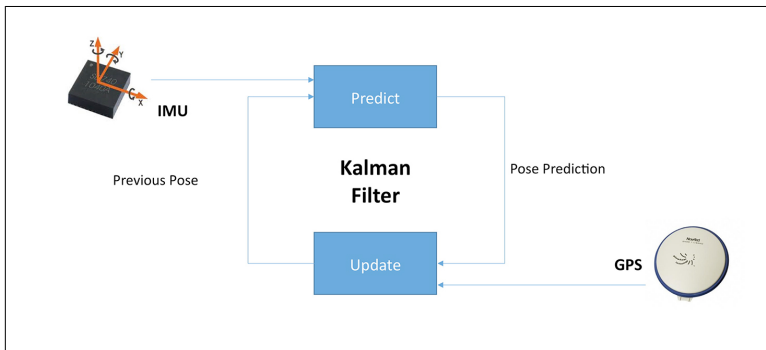


Figure 19-3. GPS/IMU localization (image courtesy of Shaoshan Liu)

Cameras can be used for localization, too. Vision-based localization undergoes the following simplified pipeline:

1. By triangulating stereo image pairs, we first obtain a disparity map that can be used to derive depth information for each point.
2. By matching salient features between successive stereo image frames in order to establish correlations between feature points in different frames, we could then estimate the motion between the past two frames.
3. We compare the salient features against those in the known map to derive the current position of the vehicle. However, since a vision-based localization approach is very sensitive to lighting conditions, this approach alone would not be reliable.

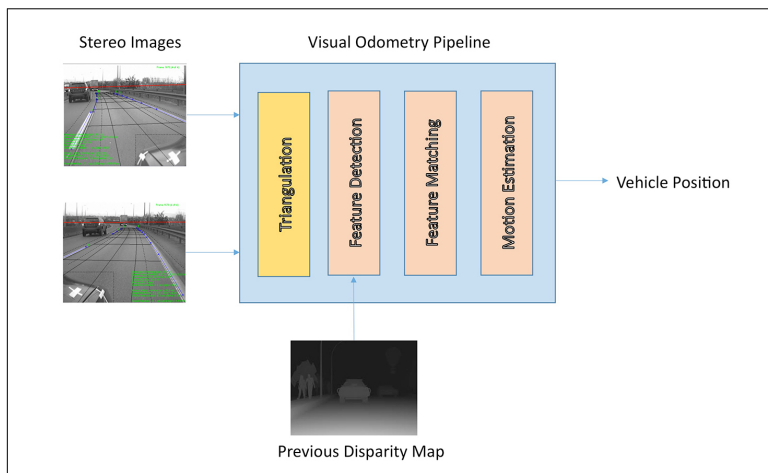


Figure 19-4. Stereo visual odometry (image courtesy of Shaoshan Liu)

Therefore, LIDAR is usually the main sensor used for localization, relying heavily on a particle filter. The point clouds generated by LIDAR provide a “shape description” of the environment, but it is hard to differentiate individual points. By using a particle filter, the system compares a specific observed shape against the known map to reduce uncertainty.

To localize a moving vehicle relative to these maps, we apply a particle filter method to correlate the LIDAR measurements with the map. The particle filter method has been demonstrated to achieve real-time localization with 10-centimeter accuracy and is effective in

urban environments. However, LIDAR has its own problem: when there are many suspended particles in the air, such as raindrops or dust, the measurements may be extremely noisy.

Therefore, to achieve reliable and accurate localization, we need a sensor-fusion process to combine the advantages of all sensors, as shown in **Figure 19-5**.

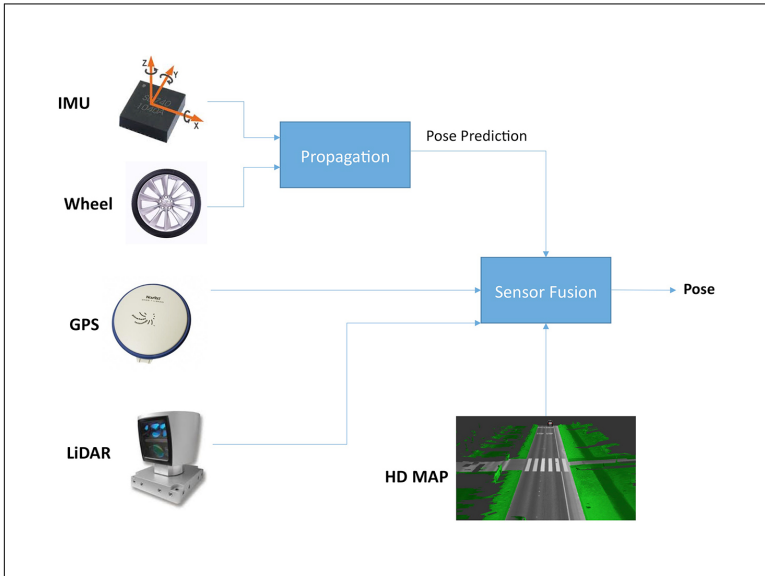


Figure 19-5. Sensor-fusion localization pipeline (image courtesy of Shaoshan Liu)

Object recognition and tracking

Since LIDAR provides accurate depth information, it was originally used to perform object detection and tracking tasks in AVs. In recent years, however, we have seen the rapid development of deep learning technology, which achieves significant object detection and tracking accuracy.

A *convolution neural network* (CNN) is a type of deep neural network that is widely used in object recognition tasks. A general CNN evaluation pipeline usually consists of the following layers:

1. The convolution layer uses different filters to extract different features from the input image. Each filter contains a set of

- “learnable” parameters that will be derived after the training stage.
2. The activation layer decides whether to activate the target neuron or not.
 3. The pooling layer reduces the spatial size of the representation to reduce the number of parameters and consequently the computation in the network.
 4. The fully connected layer connects all neurons to all activations in the previous layer.

Once an object is identified using a CNN, next comes the automatic estimation of the trajectory of that object as it moves—or, *object tracking*.

Object tracking technology can be used to track nearby moving vehicles, as well as people crossing the road, to ensure the current vehicle does not collide with moving objects. In recent years, deep learning techniques have demonstrated advantages in object tracking compared to conventional computer vision techniques. By using auxiliary natural images, a stacked autoencoder can be trained offline to learn generic image features that are more robust against variations in viewpoints and vehicle positions. Then, the offline-trained model can be applied for online tracking.

Decision

In the decision stage, action prediction, path planning, and obstacle avoidance mechanisms are combined to generate an effective action plan in real time.

Action prediction

One of the main challenges for human drivers when navigating through traffic is to cope with the *possible* actions of other drivers, which directly influence their own driving strategy. This is especially true when there are multiple lanes on the road or at a traffic change point. To make sure that the AV travels safely in these environments, the decision unit generates predictions of nearby vehicles then decides on an action plan based on these predictions.

To predict actions of other vehicles, one can generate a stochastic model of the reachable position sets of the other traffic participants, and associate these reachable sets with probability distributions.

Path planning

Planning the path of an autonomous, responsive vehicle in a dynamic environment is a complex problem, especially when the vehicle is required to use its full maneuvering capabilities. One approach would be to use deterministic, complete algorithms—search all possible paths and utilize a cost function to identify the best path. However, this requires enormous computational resources and may be unable to deliver real-time navigation plans. To circumvent this computational complexity and provide effective real-time path planning, probabilistic planners have been utilized.

Obstacle avoidance

Since safety is of paramount concern in autonomous driving, we should employ at least two levels of obstacle avoidance mechanisms to ensure that the vehicle will not collide with obstacles. The first level is proactive and based on traffic predictions. The traffic prediction mechanism generates measures like time-to-collision or predicted-minimum-distance. Based on these measures, the obstacle avoidance mechanism is triggered to perform local-path re-planning. If the proactive mechanism fails, the second-level reactive mechanism, using radar data, takes over. Once radar detects an obstacle ahead of the path, it overrides the current controls to avoid the obstacle.

The Client System

The client system integrates the above-mentioned algorithms together to meet real-time and reliability requirements. There are three challenges to overcome:

1. The system needs to make sure that the processing pipeline is fast enough to consume the enormous amount of sensor data generated.
2. If a part of the system fails, it needs to be robust enough to recover from the failure.
3. The system needs to perform all the computations under energy and resource constraints.

Robotics Operating System

A *robotics operating system* (ROS) is a widely used, powerful distributed computing framework tailored for robotics applications (see [Figure 19-6](#)).

Each robotic task, such as localization, is hosted in an ROS node. These nodes communicate with each other through topics and services. It is a suitable operating system for autonomous driving, except that it suffers from a few problems:

Reliability

ROS has a single master and no monitor to recover failed nodes.

Performance

When sending out broadcast messages, it duplicates the message multiple times, leading to performance degradation.

Security

It has no authentication and encryption mechanisms.

Although ROS 2.0 promised to fix these problems, it has not been extensively tested, and many features are not yet available.

Therefore, in order to use ROS in autonomous driving, we need to solve these problems first.

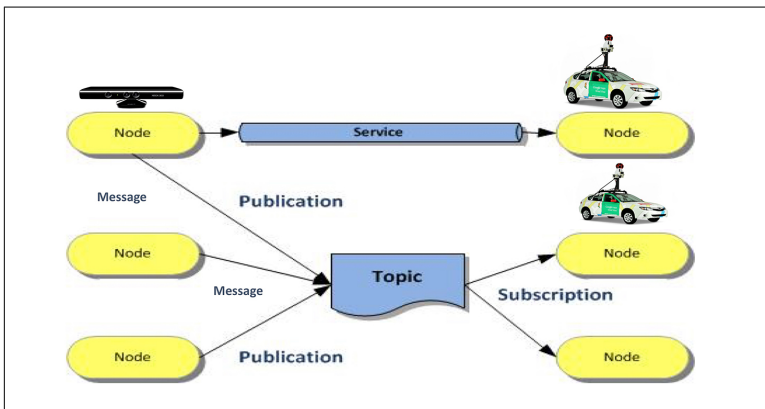


Figure 19-6. A robotics operating system (ROS) (image courtesy of Shaoshan Liu)

Reliability

The current ROS implementation has only one master node, so when the master node crashes, the whole system crashes. This does not meet the safety requirements for autonomous driving. To fix this problem, we implement a ZooKeeper-like mechanism in ROS. As shown in **Figure 19-7**, the design incorporates a main master node and a backup master node. In the case of main node failure, the backup node would take over, making sure the system continues to run without hiccups. In addition, the ZooKeeper mechanism monitors and restarts any failed nodes, making sure the whole ROS system stays reliable.

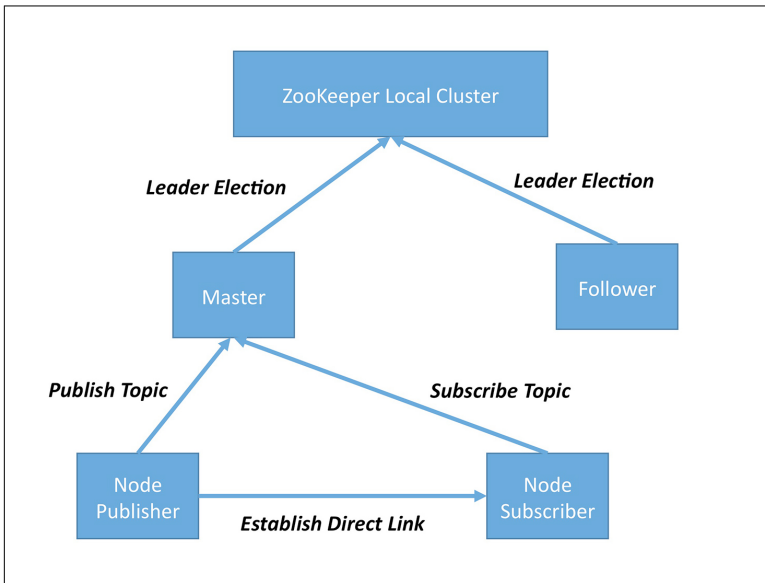


Figure 19-7. ZooKeeper for ROS (image courtesy of Shaoshan Liu)

Performance

Performance is another problem with the current ROS implementation—the ROS nodes communicate often, as it's imperative that communication between nodes is efficient. First, communication goes through the loop-back mechanism when local nodes communicate with each other. Each time it goes through the loopback pipeline, a 20-microsecond overhead is introduced. To eliminate this local node communication overhead, we can use a shared memory mechanism such that the message does not have to go through the

TCP/IP stack to get to the destination node. Second, when an ROS node broadcasts a message, the message gets copied multiple times, consuming significant bandwidth in the system. Switching to a multicast mechanism greatly improves the throughput of the system.

Security

Security is the most critical concern for an ROS. Imagine two scenarios: in the first, an ROS node is kidnapped and is made to continuously allocate memory until the system runs out of memory and starts killing other ROS nodes and the hacker successfully crashes the system. In the second scenario—since, by default, ROS messages are not encrypted—a hacker can easily eavesdrop on the message between nodes and apply man-in-the-middle attacks.

To fix the first security problem, we can use Linux containers (LXC) to restrict the number of resources used by each node and also provide a sandbox mechanism to protect the nodes from each other, effectively preventing resource leaking. To fix the second problem, we can encrypt messages in communication, preventing messages from being eavesdropped.

Hardware Platform

To understand the challenges in designing a hardware platform for autonomous driving, let us examine the computing platform implementation from a leading autonomous driving company. It consists of two compute boxes, each equipped with an Intel Xeon E5 processor and four to eight Nvidia Tesla K80 GPU accelerators. The second compute box performs exactly the same tasks and is used for reliability—if the first box fails, the second box can immediately take over.

In the worst case, if both boxes run at their peak, using more than 5,000 W of power, an enormous amount of heat would be generated. Each box costs \$20k to \$30k, making this solution unaffordable for average consumers.

The power, heat dissipation, and cost requirements of this design prevent autonomous driving from reaching the general public (so far). To explore the edges of the envelope and understand how well an autonomous driving system could perform on an ARM mobile SoC, we can implement a simplified, vision-based autonomous driv-

ing system on an ARM-based mobile SoC with peak power consumption of 15 W.

Surprisingly, the performance is not bad at all: the localization pipeline is able to process 25 images per second, almost keeping up with image generation at 30 images per second. The deep learning pipeline is able to perform two to three object recognition tasks per second. The planning and control pipeline is able to plan a path within 6 ms. With this system, we are able to drive the vehicle at around five miles per hour without any loss of localization.

Cloud Platform

Autonomous vehicles are mobile systems and therefore need a cloud platform to provide supports. The two main functions provided by the cloud include distributed computing and distributed storage. This system has several applications, including simulation, which is used to verify new algorithms, high-definition (HD) map production, and deep learning model training. To build such a platform, we use Spark for distributed computing, OpenCL for heterogeneous computing, and Alluxio for in-memory storage.

We can deliver a reliable, low-latency, and high-throughput autonomous driving cloud by integrating these three.

Simulation

The first application of a cloud platform system is simulation. Whenever we develop a new algorithm, we need to test it thoroughly before we can deploy it on real cars (where the cost would be enormous and the turn-around time too long).

Therefore, we can test the system on simulators, such as replaying data through ROS nodes. However, if we were to test the new algorithm on a single machine, either it would take too long or we wouldn't have enough test coverage.

To solve this problem, we can use a distributed simulation platform, as shown in [Figure 19-8](#).

Here, Spark is used to manage distributed computing nodes, and on each node, we can run an ROS replay instance. In one autonomous driving object recognition test set, it took three hours to run on a

single server; by using the distributed system, scaled to eight machines, the test finished in 25 minutes.

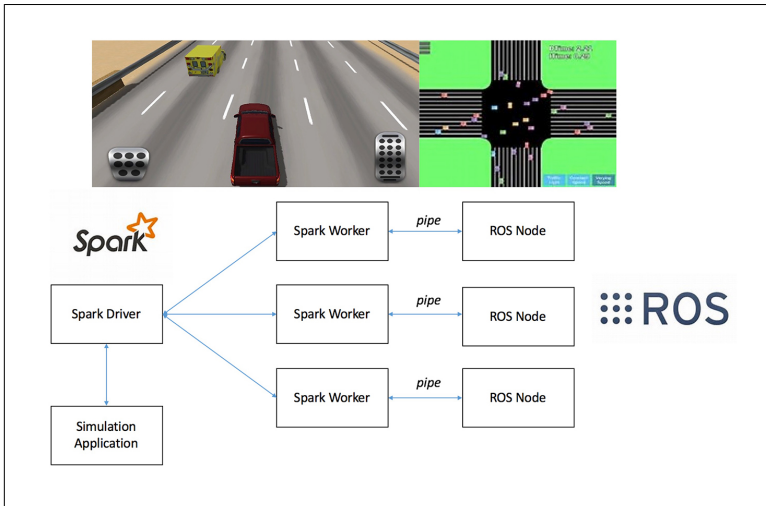


Figure 19-8. Spark and ROS-based simulation platform (image courtesy of Shaoshan Liu)

HD Map Production

As shown in [Figure 19-9](#), HD map production is a complex process that involves many stages, including raw data processing, point cloud production, point cloud alignment, 2D reflectance map generation, and HD map labeling, as well as the final map generation.

Using Spark, we can connect all these stages together in one Spark job. A great advantage is that Spark provides an in-memory computing mechanism, such that we do not have to store the intermediate data in hard disk, thus greatly reducing the performance of the map production process.

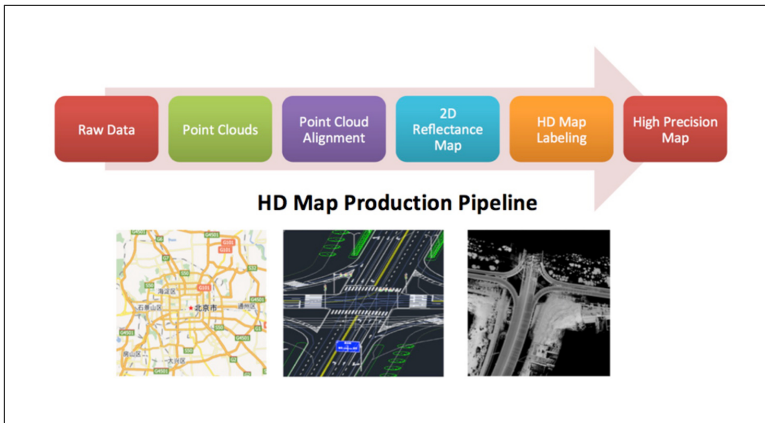


Figure 19-9. Cloud-based HD map production (image courtesy of Shaoshan Liu)

Deep Learning Model Training

As we use different deep learning models in autonomous driving, it is imperative to provide updates that will continuously improve the effectiveness and efficiency of these models. However, since the amount of raw data generated is enormous, we would not be able to achieve fast model training using single servers.

To approach this problem, we can develop a highly scalable distributed deep learning system using Spark and **Paddle** (a deep learning platform recently open sourced by Baidu).

In the Spark driver, we can manage a Spark context and a Paddle context, and in each node, the Spark executor hosts a Paddler trainer instance. On top of that, we can use Alluxio as a parameter server for this system. Using this system, we have achieved linear performance scaling, even as we add more resources, proving that the system is highly scalable.

Just the Beginning

As you can see, autonomous driving (and artificial intelligence in general) is not one technology; it is an integration of many technologies. It demands innovations in algorithms, system integrations, and cloud platforms. It's just the beginning, and there are tons of

opportunities. I anticipate that by 2020, we will officially start this AI era and see many AI-based products in the market. Let's be ready.

Shaoshan Liu

Shaoshan Liu is the cofounder and president of PerceptIn, working on developing the next-generation robotics platform. Before founding PerceptIn, he worked on autonomous driving and deep learning infrastructure at Baidu USA. Liu has a PhD in computer engineering from the University of California, Irvine.

PART VI

Integrating Human and Machine Intelligence

In this final section of *Artificial Intelligence Now*, we confront the larger aims of artificial intelligence: to better human life. Ben Lorica and Adam Marcus discuss the development of human-AI hybrid applications and workflows, and then Ben and Mike Tung discuss using AI to map and access large-scale knowledge databases.

Building Human-Assisted AI Applications

Ben Lorica

In the [August 25, 2016 episode](#) of the *O'Reilly Data Show*, I spoke with [Adam Marcus](#), cofounder and CTO of [B12](#), a startup focused on building human-in-the-loop intelligent applications. We talked about the open source platform [Orchestra](#) for coordinating human-in-the-loop projects, the current wave of human-assisted AI applications, best practices for reviewing and scoring experts, and [flash teams](#).

Here are some highlights from our conversation.

Orchestra: A Platform for Building Human-Assisted AI Applications

I spent a total of three years doing web-scale structured data extraction. Toward the end of that period, I started speaking with Nitesh Banta, my cofounder at B12, and we said, “Hey, it’s really awesome that you can coordinate all of these experts all over the world and give them all of these human-assisted AIs to take a first pass at work so that a lot of the labor goes away and you can use humans where they’re uniquely positioned.” But we really only managed to make a dent in data extraction and data entry. We thought that an interesting work model was emerging here, where you had human-assisted AIs and they were able to help experts do way more interesting knowledge work tasks. We’re interested, at B12, about pushing all of this work up the knowledge work stack. The first

stage in this process is to build out the infrastructure to make this possible.

This is where **Orchestra** comes in. It's completely open source; it's available for anyone to use on GitHub and contribute to. What Orchestra does is basically serve as the infrastructure for building all sorts of human-in-the-loop and human-assisted AI applications. It essentially helps coordinate teams of experts who are working on really challenging workflows and pairs them up with all sorts of automation, custom-user interfaces, and tools to make them a lot more effective at their jobs.

The first product that we built on top of Orchestra is an intelligent website product: a client will come to us and say that they'd like to get their web presence set up. Orchestra will quickly recruit the best designer, the best client executive, the best copywriter onto a team, and it will follow a predefined workflow. The client executive will be scheduled to interview the client. Once an interview is completed, a designer is then staffed onto the project automatically. Human-assisted AI, essentially an algorithmic design, is run so that we can take some of the client's preferences and automatically generate a few initial passes at different websites for them, and then the designer is presented with those and gets to make the critical creative design decisions. Other folks are brought onto the project by Orchestra as needed. If we need a copywriter, if we need more expertise, then Orchestra can recruit the necessary staff. Essentially, Orchestra is a workflow management tool that brings together all sorts of experts, automates a lot of the really annoying project management functionality that you typically have to bring project managers onboard to do, and empowers the experts with all sorts of automation so they can focus on what they're uniquely positioned to do.

Bots and Data Flow Programming for Human-in-the-Loop Projects

Your readers are probably really familiar with things like data flow and workflow programming systems, and systems like that. In Orchestra, you declaratively describe a workflow, where various steps are either completed by humans or machines. It's Orchestra's job at that point, when it's time for a machine to jump in (and in our case its algorithmic design) to take a first pass at designing a website. It's also Orchestra's job to look at which steps in the workflow have been completed and when it should do things like staff a project, notice that the people executing the work are maybe falling off course on the project and that we need more active process management, bring in incentives, and so forth.

The way we've accomplished all of this project automation in Orchestra is through bots, the super popular topic right now. The way it works for us is that Orchestra is pretty tightly integrated with [Slack](#). At this point, probably everyone has used Slack for communicating with some kind of organization. Whenever an expert is brought into a project that Orchestra is working on, it will invite that expert to a Slack channel, where all of the other experts on his or her team are as well. Since the experts on our platform are using Orchestra and Slack together, we've created these bots that help automate process and project automation. All sorts of things like staffing, process management, incentives, and review hierarchies are managed through conversation.

I'll give you an example in the world of staffing. Before we added staffing functionality to Orchestra, whenever we wanted to bring a designer onto a project, we'd have to send a bunch of messages over Slack: "Hey, is anyone available to work on a project?" The designers didn't have a lot of context, so sometimes it would take about an hour of work for us to actually do the recruiting, and experts wouldn't get back to us for a day or two. We built a staffbot into Orchestra in response to this problem, and now the staffbot has a sense of how well experts have completed various tasks in the past, how much they already have on their plates, and the staffbot can create a ranking of the experts on the platform and reach out to the ones who are the best matches.

...Orchestra reaches out to the best expert matches over Slack and sends a message along the lines of, "Hey, here's a client brief for this particular project. Would you like to accept the task and join the team?" An expert who is interested just has to click a button, and then he or she is integrated into the Orchestra project and folded into the Slack group that's completing that task. We've reduced the time to staff a project from a few days down to a little less than five minutes.

Related Resources

- [“Crowdsourcing at GoDaddy: How I Learned to Stop Worrying and Love the Crowd”](#) (a presentation by Adam Marcus)
- [“Why data preparation frameworks rely on human-in-the-loop systems”](#)
- [“Building a business that combines human experts and data science”](#)
- [“Metadata services can lead to performance and organizational improvements”](#)

Ben Lorica

Ben Lorica is the Chief Data Scientist and Director of Content Strategy for Data at O'Reilly Media, Inc. He has applied business intelligence, data mining, machine learning, and statistical analysis in a variety of settings including direct marketing, consumer and market research, targeted advertising, text mining, and financial engineering. His background includes stints with an investment management company, internet startups, and financial services.

Using AI to Build a Comprehensive Database of Knowledge

Ben Lorica

Extracting structured information from semi-structured or unstructured data sources (“dark data”) is an important problem. One can take it a step further by attempting to automatically build a **knowledge graph** from the same data sources. Knowledge databases and graphs are built using (semi-supervised) machine learning, and then subsequently used to power intelligent systems that form the basis of AI applications. The more advanced messaging and chat bots you’ve encountered rely on these knowledge stores to interact with users.

In the **June 2, 2016 episode** of the *Data Show*, I spoke with **Mike Tung**, founder and CEO of **Diffbot**, a company dedicated to building large-scale knowledge databases. Diffbot is at the heart of many web applications, and it’s starting to power a wide array of intelligent applications. We talked about the challenges of building a web-scale platform for doing highly accurate, semi-supervised, structured data extraction. We also took a tour through the AI landscape and the early days of self-driving cars.

Here are some highlights from our conversation.

Building the Largest Structured Database of Knowledge

If you think about the web as a virtual world, there are more pixels on the surface area of the web than there are square millimeters on the surface of the earth. As a surface for computer vision and parsing, it's amazing, and you don't have to actually build a physical robot in order to traverse the web. It is pretty tricky though.

... For example, Google has a knowledge graph team—I'm sure your listeners are aware from a startup that was building something called **Freebase**, which is crowdsourced, kind of like a Wikipedia for data. They've continued to build upon that at Google adding more and more human curators. ... It's a mix of software, but there's definitely thousands and thousands of people that actually contribute to their knowledge graph. Whereas in contrast, we are a team of 15 of the top AI people in the world. We don't have anyone that's curating the knowledge. All of the knowledge is completely synthesized by our AI system. When our customers use our service, they're directly using the output of the AI. There's no human involved in the loop of our business model.

...Our high-level goal is to build the largest structured database of knowledge. The most comprehensive map of all of the entities and the facts about those entities. The way we're doing it is by combining multiple data sources. One of them is the web, so we have this crawler that's crawling the entire surface area of the web.

Knowledge Component of an AI System

If you look at other groups doing AI research, a lot of them are focused on very much the same as the academic style of research, which is coming out of new algorithms and publishing to sort of the same conferences. If you look at some of these industrial AI labs—they're doing the same kind of work that they would be doing in academia—whereas what we're doing, in terms of building this large data set, would not have been created otherwise without starting this effort. ... I think you need really good algorithms, and you also need really good data.

... One of the key things we believe is that it might be possible to build a human-level reasoning system. If you just had enough structured information to do it on.

... Basically, the semantic web vision never really got fully realized because of the chicken-and-egg problem. You need enough people to annotate data, and annotate it for the purpose of the semantic

web—to build a comprehensiveness of knowledge—and not for the actual purpose, which is perhaps showing web pages to end users.

Then, with this comprehensiveness of knowledge, people can build a lot of apps on top of it. Then the idea would be this virtuous cycle where you have a bunch of killer apps for this data, and then that would prompt more people to tag more things. That virtuous cycle never really got going in my view, and there have been a lot of efforts to do that over the years with RDS/RSS and things like that.

... What we're trying to do is basically take the annotation aspect out of the hands of humans. The idea here is that these AI algorithms are good enough that we can actually have AI build the semantic web.

Leveraging Open Source Projects: WebKit and Gigablast

... Roughly, what happens when our robot first encounters a page is we render the page in our own customized rendering engine, which is a fork of **WebKit** that's basically had its face ripped off. It doesn't have all the human niceties of a web browser, and it runs much faster than a browser because it doesn't need those human-facing components. ...The other difference is we've instrumented the whole rendering process. We have access to all of the pixels on the page for each XY position. ...[We identify many] features that feed into our semi-supervised learning system. Then millions of lines of code later, out comes knowledge.

... Our VP of search, Matt Wells, is the founder of the **Gigablast** search engine. Years ago, Gigablast competed against Google and Inktomi and AltaVista and others. Gigablast actually had a larger real-time search index than Google at that time. Matt is a world expert in search and has been developing his C++ crawler Gigablast for, I would say, almost a decade. ... Gigablast scales much, much better than Lucene. I know because I'm a former user of Lucene myself. It's a very elegant system. It's a fully symmetric, masterless system. It has its own UDP-based communications protocol. It includes a full web crawler, indexer. It has real-time search capability.

Editor's note: Mike Tung is on the advisory committee for the upcoming **O'Reilly Artificial Intelligence conference**.

Related Resources

- Hadoop cofounder Mike Cafarella on the *Data Show*: “From search to distributed computing to large-scale information extraction”
- *Up and Running with Deep Learning: Tools, techniques, and workflows to train deep neural networks*
- “Building practical AI systems”
- “Using computer vision to understand big visual data”

Ben Lorica

Ben Lorica is the Chief Data Scientist and Director of Content Strategy for Data at O'Reilly Media, Inc. He has applied business intelligence, data mining, machine learning, and statistical analysis in a variety of settings including direct marketing, consumer and market research, targeted advertising, text mining, and financial engineering. His background includes stints with an investment management company, internet startups, and financial services.