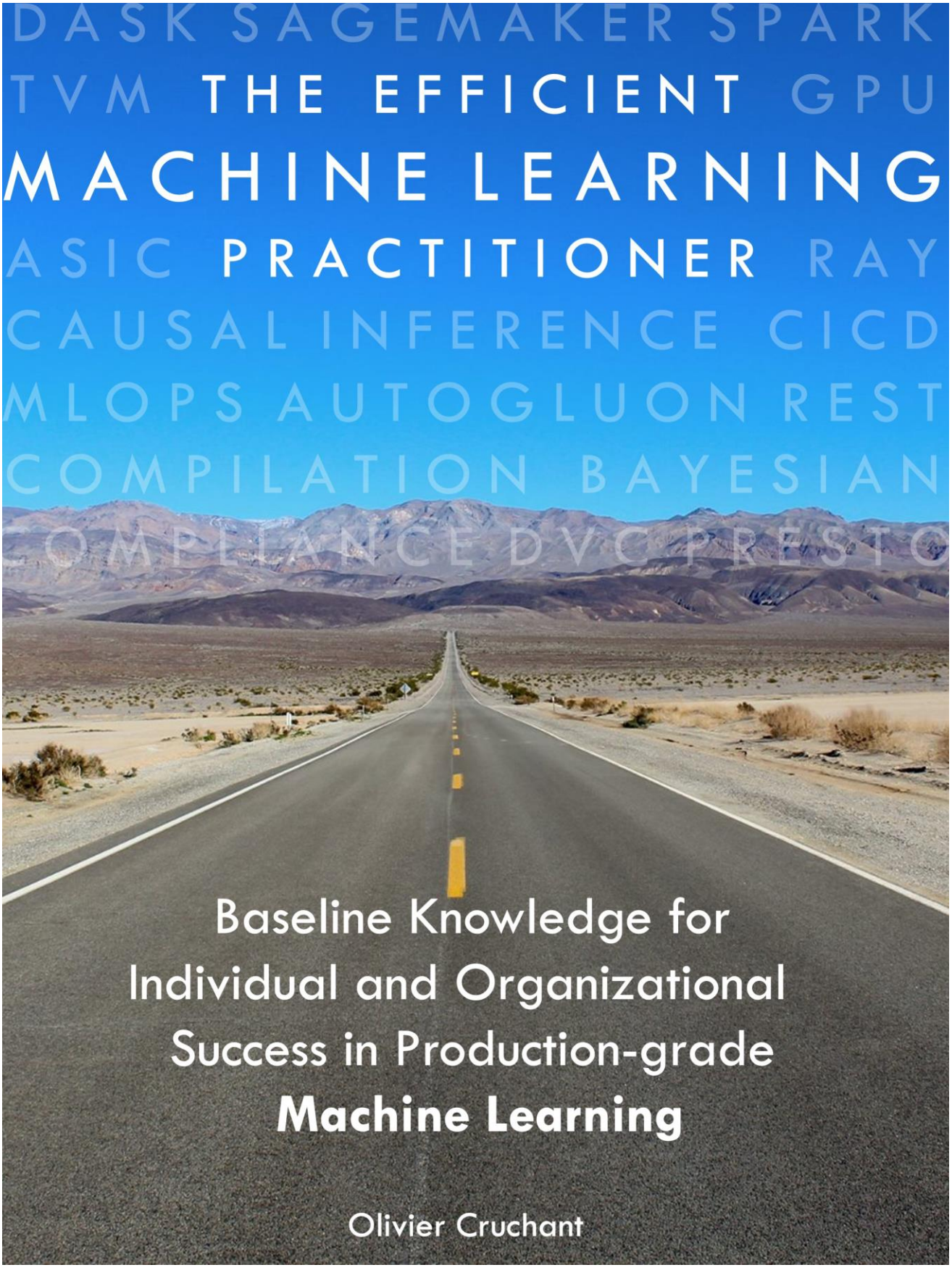
DASK SAGEMAKER SPARK
TVM
THE EFFICIENT
GPU
MACHINE LEARNING
ASIC
PRACTITIONER
RAY
CAUSAL INFERENCE
CICD
MLOPS AUTOGLUON
REST
COMPILATION
BAYESIAN
COMPLIANCE DVC PRESTO
Baseline Knowledge for
Individual and Organizational
Success in Production-grade
Machine Learning
Olivier Cruchant

THE EFFICIENT ML PRACTITIONER

Baseline Knowledge for Individual and Organizational Success in Production-grade Machine Learning

Notices

Readers are responsible for making their own independent assessment of the information in this document. The views expressed in this work are those of the author and do not represent the views of the author employer. The author has used good faith to ensure accuracy of information and disclaims all responsibility for errors, omissions and damages resulting from the use or reliance on this work. Referred products, services, software and links may change without notice. If you decide to use ideas, techniques, open-source or commercial software mentioned in this book, it is your responsibility to ensure that your use is compliant with possibly associated patents, terms of use and license terms. It is your responsibility to ensure that your work complies with any regulatory and legal requirements in your industry and location. Use of the information and knowledge contained in this work is at your own risk.

Introduction

"What knowledge do I need for a machine learning (ML) career? How do we transform our data science projects into production systems? Do you have examples of real production ML use-cases?" ML enthusiasts, from scientists and engineers up to business leaders, ask me those questions every day. For individuals and organizations alike, the road to impactful ML is exciting yet bewildering: tools change fast, the scientific bar intimidates, the community boils with new ideas and the outer world is confused. Yet, when you start working on real ML use-cases, you realize that while tools are changing, relevant algorithms are not changing as fast. While the equations behind some algorithms are daunting, many of them use teenager-level math. While there are blog posts and research papers published every day, you only need to read a fraction of them – the right fraction - to delight your customers and yourself. Becoming an effective ML practitioner does not require super-human reading abilities nor a Fields Medal. You can be impactful and happy by investing in targeted knowledge gaps, learning skills that matter and helping others. All you need is to identify those areas deserving attention on your ML journey.

In this book, I present the differentiators that make ML impactful. I share learnings and best practices that helped me embrace a career in ML and that successful ML organizations and practitioners apply.

More about myself

I am Olivier, a French ML practitioner passionate about distributed systems, ML architecture, bandits, Apache MXNet and Amazon Web Services (AWS) cloud. I also enjoy ML talent management such as recruitment and training and I like mentoring people into ML jobs. The idea for this book came out of reading lists and knowledge checks I was repeatedly producing and curating for friends and colleagues looking for ML knowledge growth. I figured it was time I clean and write down in a single book all the tips and best practices I was passing around, and here it is! I have been working at Amazon for 7 years and at AWS for 3 years. This book is positively opinionated towards AWS.

If you want to discuss to share comments, ideas, feedback including typos, errors and omissions, please reach out to me on LinkedIn (www.linkedin.com/in/oliviercruchant) or at pro.olivier.cruchant@gmail.com.

Target Audience and Objectives

In this book I answer three questions:

1. **What are AI and ML in practice?**
2. **What baseline knowledge is needed** to embrace a technical career path in ML?
3. **Which skills are rare and valuable** for ML practitioners and organizations?

This book is valuable for multiple personas:

- **Product managers and business leaders** will learn ML success factors and discover associated best practices and field stories
- **Individuals considering a career in ML** will be able to prioritize their learning path and accelerate their transition
- **Experienced ML practitioners** will learn new techniques, use-cases and stories to sharpen their capabilities and increase their impact

What this book is not:

There are dozens of great books about machine learning. My goal is to complement them and not overlap them. This book focuses on topics not extensively covered yet in ML literature, and deliberately omits few topics.

- **This book is not an ML coding book. It contains almost no code.** If you are interested in learning ML code, I recommend the following books: *Python Data Science Handbook* by Jake VanderPlas, *Deep Learning with Python* by François Chollet, *Deep Learning for Coders* by Jeremy Howard and Sylvain Gugger or *Dive into Deep Learning* by Zhang et al.
- **This book is not an ML science book. It contains almost no equation.** If you are interested in ML science, I recommend the following books: *The Elements of Statistical Learning: Data Mining, Inference, and Prediction* by Hastie et al. and *Pattern Recognition and Machine Learning* by Bishop. Deep Learning and its practical implementation are well covered in *Dive into Deep Learning* by Zhang et al.
- **This book does not cover all ML use-cases.** I focus on topics frequent enough in production to consider them part of baseline knowledge. For example, although interesting and promising, I do not cover reinforcement learning nor generative adversarial networks (GANs).
- **This book does not cover ML interpretability**, already well covered in Christoph Molnar's book, *Interpretable Machine Learning*. As AI regulation and compliance frameworks mature, they will influence the convergence and refinement of model inspection tooling, which is still in its infancy and changing fast.

Important: This book does not teach Python programming nor any ML framework. Although it mentions many specific software libraries, the emphasis is on production-grade ML concepts and principles, which are mostly technology-agnostic. I encourage the reader to adopt a technology-agnostic mindset beyond the pages of this book. Consider specific technologies like sails powering your career: they will be pleasant travel companions powering your journey for a while, but they get old and at one point you will have to replace them. Yet changing sails has little impact on your sailing skills and your travel plans. In summary: don't over-obsess in ML frameworks and specific algorithms. Always learn what you need at the moment and grant a never-ending attention to the technology-agnostic core tenets detailed in this book.

Technical skills are like sails powering your career: they are tools helping you achieve personal and organizational goals. But you will have to change them from time to time! Don't over-obsess in ML frameworks and specific packages. Instead, focus on technology-agnostic scientific and architectural principles.

Table of Contents

Appendixes

I A Review of Real-Life Machine Learning

In this section I explain how real-life production machine learning (ML) looks like. I provide definitions, give actual examples of ML deployment and describe ML projects and organizations.

I.1 What are AI and ML?

"If it is written in PowerPoint, it is AI. If it is written in Python it is ML". This sarcastic comment from a colleague does not crisply define artificial intelligence (AI) nor machine learning (ML), yet it warns the reader that practical and concrete ML is often renamed AI in order to increase its aura. Over the years, I found that people and companies tend to agree on the following definitions.

- **AI** refers to the techniques and applications that mimic human cognitive abilities, such as problem solving, logical reasoning and analysis of abstract patterns and concepts.
- **ML** is a subset of AI that consists of algorithms capable of learning data transformations from empirical data. There exist forms of AI that are not ML: for example, you could build decent complex reasoning systems by combining pre-defined business rules.
- **Deep learning (DL)** refers to a family of ML algorithms consisting of several layers of neural networks - an elementary modelling unit doing a weighted sum of its input. Those cells are chained together into a larger model, and during training each of them applies tiny changes to its coefficients so that the whole model performs better and better in the task of interest. Each layer learns a data transformation that will help the subsequent layer reduce the overall model error. This collaboration between tens to millions of parameters makes neural network able to learn complex tasks directly from raw data such as natural text, images or speech. Modern computer vision systems powering self-driving cars or modern machine translation systems are mostly powered with deep learning.

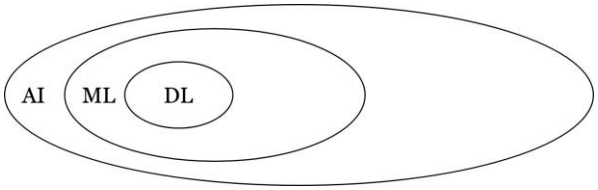

Figure 1. Deep learning (DL) is a subset of machine learning (ML) which is a subset of artificial intelligence (AI)

In practice, most of the time I met people talking about AI – be they business or technical persons – they were in particular talking about ML.

AI is simpler than you think

AI is not a near-magic black box accessible only to elite scientists. What people call AI often consists only of few lines of open-source code copied in seconds from public code packages. Anybody with a computer can train and use AI. AI algorithms are of a varying complexity yet many of them rely on centuries-old math concepts such as the difference, average, derivative, dot product, and trigonometry functions. They have been in our lives for decades already, and are more friendly and accessible than we are led to think.

To further illustrate the genuine face of ML and AI, I propose to debunk the following common myths:

- **Myth 1: *"In ML, algorithms are self-improving".*** More than 99% of the deployed ML algorithms I ever saw or read about were batch-trained, statically-deployed models. In that paradigm a developer or system trains a model on batches of historical data offline, isolated from live traffic. Once the model reaches a satisfying offline performance on historical data, it is deployed as frozen, static files in a host application. The model will not improve unless edited by a developer or a system. A rare variety of algorithms called *online-learners* support record-by-record training, also called *incremental training*. Online learners can keep updating their weights while being exposed to live traffic, and never stop evolving. Since they ingest data piece by piece, online learners have no training data size limit. Those algorithms are promising yet near-nonexistent in production. In my experience, this is because they

are harder to train, harder to deploy and harder to evaluate. To make matters worse, online-learners are also harder to promote to business leadership: they require comfort with change, exploration and randomness. Online learning is extremely rare and only seen in conferences and publications of experienced ML firms such as Microsoft, Amazon or Netflix. The open-source library Scikit-Learn contains few online-learners (*6.1. Scaling with instances using out-of-core learning*), and the most mature open-source online-learning library is Vowpal Wabbit.

- **Myth 2: *"ML refers to algorithms that can do a task without being explicitly programmed".*** The notion of "not being explicitly programmed" is surprisingly popular yet surprisingly far from the reality of data science. In order to program an ML model, one needs to write code indicating to the algorithm which dataset to use, over which timeframe, which data samples to model on and which task to run. It is common to write additional custom code to transform the data and run model validation tasks. This represents tens to several thousand lines of code. There is indeed a field of research named *AutoML* aiming to automate that process and reduce the amount of ad-hoc code required to produce an ML model. AutoML is in its infancy and rare in production. AWS contributes to AutoML by developing both a commercial, production-grade, transparent AutoML service named SageMaker Autopilot, and the state-of-the-art, benchmark beating, open-source AutoML package AutoGluon. AutoGluon is notable for its predictive performance and computational efficiency. According to its authors (*AutoGluon-Tabular: Robust and Accurate AutoML for Structured Data*, Nick Erickson et al.), *"In two popular Kaggle competitions, AutoGluon beat 99% of the participating data scientists after merely 4h of training on the raw data."*

Will AutoML replace ML practitioners?

No, it will create more of them! AutoML makes ML more accessible, similarly to the concomitant advents of digital photography and photo editing tools made good-looking photography accessible to everyone. Some steps of the model lifecycle will require less and less specialized skills: training an ML model to good accuracy on an offline dataset is now feasible by anybody with a computer. However, in 2021 there are still several steps of the ML lifecycle requiring human wisdom: for example, knowing on which problem to apply ML and where to get data from. It is wise to keep the human in command for model design

as well: ad-hoc business constraints such as cost optimization, interpretability or validation against regulatory guidelines still require human experts to get involved in the problem.

In order to concretely depicts what ML is, I present real ML stories in the next section.

I.2 Real-life ML Stories

I am often asked to give examples of business success stories driven by ML. This is a fair question actually difficult to answer. Numerous are the scientific achievements that never leave the research phase and never materialize into tangible use-cases. ML currently surfs a wave of popularity garnering attention and investment, thereby limiting the pressure on practitioners to deliver finished products and a return on investment. In this book, I highlight success stories and provide the keys to unlock business value with ML. It is important for ML practitioners and leaders to read and know ML success stories: it provides data points teaching which solution makes sense for a given problem. In the following paragraphs I narrate notable public ML stories that I found helpful to train my inner model of what production ML looks like.

Advertising

Advertising is prime ML ground for 3 reasons: (1) data is abundant, (2) money is abundant, (3) failure is tolerable - people will not die if you accidentally display the wrong ad. Many organizations report using machine learning for ad selection by predicting clicks on ads. A decade ago, Graepel et al. presented an online classification model used to serve **Microsoft** Bing's Sponsored Search traffic – a magnitude in the dozen billion ad impressions per year. (*Web-Scale Bayesian Click-Through Rate Prediction for Sponsored Search Advertising in Microsoft's Bing Search Engine*). Trained on 14 days of data, it improved the area under curve metric (AUC), measuring prediction strength, by 7% vs a relevant baseline. **Alibaba** researchers also reported significant business impact from a click-through rate (CTR) prediction model trained on 60 million users, 600 million products and 2.14 billion interactions. The CTR is a metric commonly seen in online advertising, equal to the proportion of ad displays that result in a click on the ad. Predicting CTR is a valuable goal for

advertisers, because it helps anticipate the success of an ad and prioritizing which ad to show. Learning from two weeks of data, the model described by Alibaba researchers improved revenue per impression by 3.8% (*Deep Interest Network for Click-Through Rate Prediction*, Zhou et al). In *Network–Efficient Distributed Word2vec Training System for Large Vocabularies,* **Yahoo** researchers Ordentlich et al. reported 9.4% incremental revenue in their Gemini sponsored search advertising platform thanks to a weekly-trained model predicting which URLs users click after a search query. The model learned to fill artificial holes in historical query-URL browsing sequences, a training design called *word2vec* (invented by Tomas Mikolov et al., *Efficient Estimation of Word Representations in Vector Space*).

Personalization

Personalization of the user experience based on metadata and interaction data is a major ML use-case, if not the most popular since democratized by the Netflix Prize. Launched in October 2nd 2006, this competition proposed a $1MM reward in exchange of significant improvement of the recommender system used by the video rental platform. **Netflix** invests considerably in personalization systems, and wrote in a 2015 paper (*The Netflix Recommender System: Algorithms, Business Value, and Innovation*, Gomez-Uribe et Hunt) that those systems contribute to maintaining the user churn rate low and represent an estimated $1B savings per year. Algorithms rank the recommendation carousels on the user homepage, estimate movie similarity and personalize search results, among others. **Airbnb** often publishes about its ML efforts, and in 2018 Grbovic et al. provided practical details of personalization models used to power search and similarity widgets, two channels driving 99% of conversions (*Listings Embeddings in Search*, *Real-time Personalization using Embeddings for Search Ranking at Airbnb)*. To address the *cold start* problem – many Airbnb listings have insufficient history for ML training - Airbnb researchers use bucketing and geographically close listings. In 2018, **Amazon** mentioned achieving significant purchases increase (numbers undisclosed) via a recommender system update. The model only used previous purchases as input and predicted future interactions as outputs. Amazon developers used the Apache Spark open-source big data processing framework to produce training and prediction datasets. To handle large dimensionality – the model input was a vector of same dimension as the catalog – developers used model-parallel GPU training which they open sourced in the DSSTNE library. Clipping the number of recommendable products and maintaining dataset size constant kept training time under control. A notable

finding is that all tested neural network architectures used two layers, and that deeper networks did not yield better results (*The effectiveness of a two-layer neural network for recommendations*, Rybakov et al).

Translation

Translation is a major win for machine learning and deep learning. In 2016, **Google** researchers introduced their Google neural machine translation (GNMT) model that became the backend of Google Translate and reduced translation error by an average of 60% (*Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation* We et al). GNMT is an 8-layer recurrent neural network (RNN) written in Google's open-source DL framework TensorFlow. The model trained in 6 days on the 36MM sentence pairs of the *WMT English to French* benchmark dataset. The advent of deep learning and the growth in computational capacities powered the development of commercial translation tools. **AWS** excels in this domain; its translation service Amazon Translate ranks number 1 Machine Translation Provider by Intento in 2020 (*Amazon Translate ranked as #1 machine translation provider by Intento*, Greg Rushing, AWS ML Blog)

Self-driving vehicles

In their April 2019 *Autonomy Day* YouTube video, Elon Musk and **Tesla** engineers Pete Bannon and Andrej Karpathy presented their custom in-car hardware and neural networks for self-driving. Notably, Tesla indicated not relying on popular LIDAR laser-based depth sensors and a portion of the video presented image-based depth perception. Later in 2019 Andrej Karpathy spoke at a PyTorch conference where he indicated that the Tesla self-driving vision stack consisted of 48 neural networks, processing the inputs from at least 8 cameras, outputting 1k distinct predictions and requiring 70k GPU hours to build (*PyTorch at Tesla*, Andrej Karpathy). Equally impressive, the company **TuSimple** uses machine learning to power 70-feet long self-driving long-haul trucks! TuSimple mentions having over 100 AI systems embedded in the trucks to analyze the environment, many of which built with the Apache MXNet open-source deep learning framework (*TuSimple Uses AI to Train Self-Driving Semis*, AWS Case Study). Notably, TuSimple developed and open-sourced the computer vision toolkit SimpleDet.

Real Estate

The online real-estate broker **Compass** mentioned having increased homepage CTR by 153% and listing engagement by 107% by deploying a *Similar Listings and Recommendation* feature on its website. The model is a neural network also developed with Apache MXNet and trained to associate listings likely to be viewed one after the other (*Launching Similar Homes and Real-Time Personalized Recommendations*, by Gautam Narula). Equally interesting is the history of **Zillow**, an online real-estate listing database with 5k+ employees and $2.7B revenues as of 2019. Zillow became famous for its *Zestimate*, an ML-based home value price estimation. The Zestimate is available for more than 100MM US properties and uses hundreds of variables. Zillow computes Zestimates in seconds using Apache Spark on Amazon EMR distributed computing service (*Zillow Increases Accuracy of 'Zestimates' Using Amazon Kinesis*, AWS Case Study)

Logistics

Supply chain operations can be facilitated with ML. **Aramex** is a Dubai-based international shipping company that presented a number of ML use-cases at the 2019 AWS re:Invent Summit (*AWS re:Invent 2019: How to build high-performance ML solutions at low cost, ft. Aramex (AIM306-R*). For example, Aramex mentions using ML to predict the transit time of their shipments based on multiple features such as origin and destination, weight, size, payment type. The Aramex data science team developed and trained the model in less than 2 weeks via the Amazon SageMaker managed ML platform. Aramex also mentions using machine learning to disambiguate text-based customer-provided addresses into more specific delivery coordinates.

Recruitment

Professional online social networks revolutionized the job market in the past decade. In a 6-week A/B test **LinkedIn** measured a 31% recruiter click-through rate increase in skill-based search thanks to a new candidate scoring model using a recommender system-like approach to learn non-revealed skills. (*Personalized Expertise Search at Linkedin*, Ha-Thuc et al). The dataset was a large matrix consisting of 40k skills and 350MM members. A later improvement of the recommender system increased the number of job applications increased by +7.8% and average job posting conversion rate (applications per impression) by +3.6%.

Education

The educational, language-learning application **Duolingo** mentions having improved user engagement by 12% using deep learning (*Using AI to Teach 300 Million People*, AWS Case Study). For example, one model can predict the probability that students will give correct answers in a given context. In order to develop models Duolingo used the PyTorch open-source deep learning framework on Amazon EC2 P3 GPU instances over datasets ranging from 100k to 30MM datapoints. Models make more than 300MM predictions every day.

Sports analytics

As documented in the AWS blog post *Accelerating innovation: How serverless machine learning on AWS powers F1 Insights* (Figdor et Syschikov), **Formula 1** made its video coverage more entertaining by augmenting it with ML-based stop duration prediction and overtake probabilities. Additionally, **Major League Baseball** used recommender-system like models to predict the outcome of a given batter-pitcher match. An AWS blog post describes this use-case (*Calculating new stats in Major League Baseball with Amazon SageMaker*, Karimi et al).

Farming

Agriculture is another domain where ML already creates value. The South African company **Aerobotics** provides crop quality monitoring using ML models running on drone imagery (*Aerobotics improves training speed by 24 times per sample with Amazon SageMaker and TensorFlow*, Michael Malahe). Below sea level, **Aquabyte** computer vision models create value by providing to the aquaculture industry non-intrusive fish counting and measurement techniques (*AWS re:Invent 2019: Deep learning in deep nets: Helping fish farmers feed the world (MLS210-4)*). Documented in a PyTorch case study, **Blue River Technology** uses computer vision models deployed at edge in agricultural equipment to accurately and selectively spray weeds (*AI for AG: Production machine learning for agriculture*, Chris Padwick)

Safety & Hazard Prevention

Security is a priority for both enterprises and individuals. ML shows great promise in automating a number of security and safety controls. For example, **Alchera Inc** mentioned in January 2020 being rolling out a vision-based fire and smoke detection model in both

South Korea and California (*Using MXNet to Detect Fire from Live Video*, Jonggon Kim). Furthermore, in 2020 **AWS** announced a commercial, pre-trained application programming interface (API) enabling any developer to detect the presence of personal protective equipment (PPE) in images, a frequent health & safety use-case (*Automatically detecting personal protective equipment on persons in images using Amazon Rekognition*, Agrawal et al.)

Capacity planning

Machine learning can improve infrastructure capacity planning. According to a 2020 **AWS** blog post, the data warehousing service Amazon Redshift internally uses Amazon Forecast to predict service demand and right-size a warm pool of instances. According to the team, the use of this service improved warm pool capacity utilization by 70%. (*Automating your Amazon Forecast workflow with Lambda, Step Functions and CloudWatch Events rule*, Ma et al.)

It is still early days for production-grade ML. Although the ecosystem of open-source and commercial tools is changing fast, few patterns consistently emerged at mature ML-intensive organizations. In the following sections, I detail those trends and explain that ML projects can be classified into 3 categories.

I.3 Data-driven management and the 3 flavors of ML

Machine learning cannot exist without data. I often witnessed how easy it is for organizations with a solid data management foundation to build up an ML practice. When the right ingredients are in place, impactful machine learning ecosystems develop at a frenetic pace. In this section, I share the recipe that I refined over the years. Data-driven companies come in diverse shapes and cultures, yet they will often exhibit the below principles. They cluster in three categories: People & Culture, Tools and Processes.

People & Culture

The most important ingredient for successful ML is a group of humans with the right skillset and the right mindset. Several cultural characteristics matter.

Hiring bar

Data-driven companies tend to hire people that are curious and eager to learn. Given how fast data processing technologies replace one another, hiring fast-learners resilient to change is a better bet than hiring the expert of a currently trending technology. Data-driven companies also hire people with a culture of measurement. Valuing measurement is not new: it has been trending in business for decades with the advent of *lean manufacturing*. In companies where data can easily be accessed and processed, measurement-driven individuals create value and are happy.

Selective, data-driven investment

The culture of measurement drives rational and automation-prone operations: metrics and impact estimation fuel processes and decision-making. Doing something because "Everybody does it in the industry" or "we've always done it" is not accepted.

Tech-aware leadership

It is common and of capital importance to have science and technology advocates up in all the leadership layers, to understand and support the scientific initiatives happening on the field. Such a finding is not new: in his excellent book *The Idea Factory: Bell Labs and the Great Age of American Innovation*, Jon Gertner called it out as a likely factor of success of Bell Labs. Bell Labs is an iconic, successful research laboratory ignited in 1925 as the research arm of the telecom firm AT&T; it is credited with most of the major inventions that powered the information technology revolution such as the transistor, the laser, the photovoltaic cell, UNIX, C and C++ among others.

Tools

Data-driven companies have surprisingly similar high-level data processing infrastructure, based on the three following foundational building blocks.

Raw data storage: Data Warehouses and Data Lakes

The foundation of a data-driven company is an easily accessible yet secure raw data storage system. Historically called *data warehouse* when it consisted of databases, it is nowadays called a *data lake* since based on format-agnostic object storage systems. Amazon S3 is a popular commercial service to support data lakes, and you will see its name in the technology blogs and papers of many major technology-focused companies such as Autodesk, Expedia, Netflix, Siemens, Zillow. Size of raw data stored in a data warehouse or data lake varies across teams and organizations and could be as small as a couple gigabytes (GB). At the other end of the spectrum, Netflix mentioned in September 2018 at Strata NY storing 100 petabytes (PB) of data in their Amazon S3 raw data storage. Data warehouses are still valuable to query tabular data at low-latency with complex joins and aggregations. Amazon Redshift is a popular commercial data warehouse from AWS and ClickHouse a popular open-source option developed by Yandex. Major tech companies report using ClickHouse including Spotify *(Spotify's New Experimentation Platform*, Johan Rydberg), CloudFlare (*Cloudflare Bot Management: machine learning and more*, Alex Bocharov) and Contentsquare (*ClickHouse Paris Meetup - ClickHouse at ContentSquare*).

Computing power

Once people access data, they need to process it. Data processing requests vary in terms of compute power and program expressivity so this layer must offer multiple hardware types (CPU, GPU, multi-node) and multiple interfaces (SQL, Spark, Python, drag-and-drop). For exploration and research, data analysts and ML practitioners love environments enabling real-time coding and visualization. The Jupyter notebook (https://jupyter.org/) is a famous open-source browser-based interface to write documents mixing narrative, code and visuals. Jupyter is adored by researchers and scientists working on exploratory and narrative-style analysis. On the other hand, software engineers and developers often dislike Jupyter notebooks and prefer writing scripts files on traditional integrated development environments (IDE) such as PyCharm, on behalf of better readability, portability and repeatability. Because compute is expensive and task-specific, and because centralizing compute resources causes contention risk and operational challenges, it is preferred to have

an infrastructure offering ephemeral, dedicated compute resources per task. The Amazon SageMaker Training job service provides exactly this experience and ML practitioners love it. Once ML code has been written and needs to be scheduled or launched at scale, it does not need any IDE nor notebook to run. It can be launched via an API call and run without human supervision as an ephemeral background task. This "fire and forget" task design is called an *asynchronous* call. Human time and compute time are expensive, and asynchronous computing APIs help using both frugally. Data processing and modelling projects rarely consist of standalone jobs, and are often made of a complex sequence or graph of tasks. Such compute workflows can be designed with *workflow schedulers*, such as the popular Apache Airflow or the commercial offering AWS Step Function and the recent SageMaker Pipelines. Workflow schedulers enable automation and developer expressivity and are a key tool in the data-driven company's toolbox.

<table>
<tr><th></th><th>Interactive querying</th><th>Asynchronous job</th></tr>
<tr><td>Small data (<~10GB)</td><td rowspan="2">SageMaker Notebook</td><td>AWS Lambda</td></tr>
<tr><td>Medium data (~10GB-100GB)</td><td rowspan="2">Amazon Athena
SageMaker Training
SageMaker Processing
Amazon EMR Step</td></tr>
<tr><td>Big data (>~100 GB)</td><td>Amazon Athena
Amazon EMR Notebook</td></tr>
</table>

Figure 2. What compute service will you and your teams need? Here is my personal shortlist and decision matrix for AWS data processing services.

Self-service Dashboards

Dashboards serve two purposes: (i) displaying health of a line of business against trends, goals and limits, (ii) providing self-service access to ad-hoc metrics to non-coders. Dashboards pump data across departments and up to the lesser technical layers of the organization. Apache Superset is a popular dashboarding option open-sourced by Airbnb. Tableau Software is a popular commercial alternative. Organizations should nonetheless be careful not to suffer from dashboard overdose. Dashboards alone are far from enough to run a data-driven business. They are descriptive tools that rarely have advanced modelling nor scripting capabilities. Another problem of dashboards – the most damaging one in my opinion – is that they do not scale. Dashboards and visual analytics are a friction to

automation. They disrupt the flow of data by forcing a human eye on its path. Human have limited attention capacity and one should not expect humans to correctly analyze hundred metrics in visual dashboards. Dashboards, when possible, should be replaced or complemented by automated scripts that monitor metrics against their baseline, and raise requests for human verification only when anomalies are found.

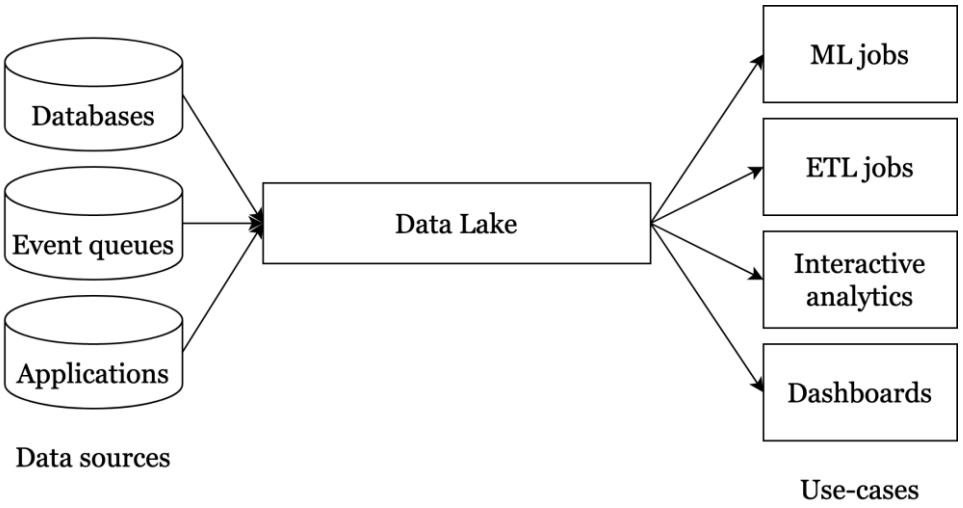

Figure 3. The data lake acts as a storage buffer between data sources and use-cases. Separating compute from storage increases agility and reliability while reducing costs

Processes

People and tools are not enough to deliver results consistently. Organizations need processes to get in motion and create value. Processes drive productivity: they reduce the number of ad-hoc meetings and thinking needed to address a situation. I detail below a non-exhaustive list of baseline processes I believe to be factors of sucess in a data-driven organization.

Goal setting

There needs to be centrally-set goals, known by everyone in the organization and which drip down into sub-goals for units and teams. This ensures everyone works in the right direction and is challenged to produce results. Furthermore, having teams sharing the same compass limits the need for inter-team communication and makes everybody more productive.

Reporting

Reporting of actuals against goals should occur at a frequency suitable for the business – for example weekly - for teams to react to anomalies and fix issues. Machine-generated text - for example a script that runs SQL queries and fills holes in pre-written text - can automate reporting. Goals catalyze reporting automation as they filter signal from noise. For example, when goals take into account fine-grained seasonality it is easy to distinguish an expected decrease from an abnormal decrease, as illustrated in figure 4. Alarms are raised and human time invested only when metrics divert significantly from goals and expectations.

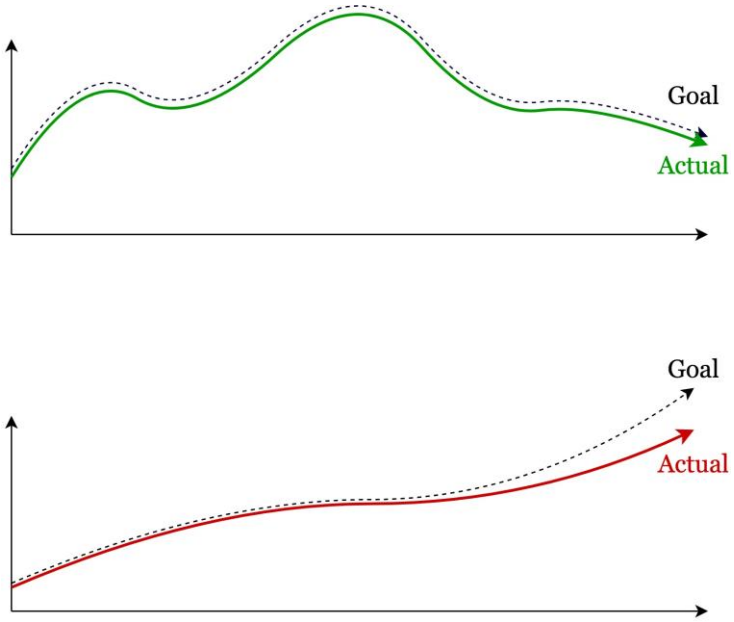

Figure 4. The importance of granular goals: in the top figure, the actuals are aligned with goal, despite the downward trends and the variations, which were anticipated by the goals and could be due to seasonality for example. In the bottom figure, the actuals diverge from the goal, which may trigger further investigation by analysts. Figuring out if a trend is worth worrying about is difficult without quantitative goals and expectations.

Documentation & knowledge sharing

Self-service access to knowledge catalyzes scale and productivity for both individuals and organizations. Internal wikis and documentation portals can document tools and processes, including team onboarding materials. Numerous tools augment the knowledge sharing capacity of a company, notably (1) an internal question & answers forum, (2) an internal

survey tool to acquire data and drive initiatives, (3) a video broadcasting platform to host virtual meetups and tutorials.

Self-service Information Security

Premium cars are equipped with premium safety systems and it is no different with companies. If a company wants its teams to iterate at lightning speed with no concession on quality, it must have solid self-service information security mechanisms. In the digital world security is a key concern for stakeholders and end users; organizations that appropriately prioritize it are rewarded with customer trust and productivity. Self-service information security mechanisms should include: (1) regular employee training to maintain awareness of security practices (2) a data classification and handling standard, (3) an application development standard, (4) a catalog of pre-approved software and code libraries (5) well-staffed security & compliance ticket queues to answer exotic security questions teams will have as they work.

With this baseline in place, a company can create value from ML by embracing 3 types of investments:

Creating new products

Machine learning can power new revenue-generating products exposed to external customers. For example:

- Smart cameras for remote sensing
- Intelligent photo and video editing software
- Translation, transcription and voice synthesis services

Automating and improving processes

Machine learning can automate processes. The return on this type of investment is usually measured in cost savings generated by the automation. There could also be incremental revenue caused by process improvement, if customer-facing. For example:

- Extracting handwritten data from forms and documents
- Creating realistic revenue goals for a sales team
- Learning the right bitrate for video streaming
- Ranking articles in a digital newspaper

- Setting investment size per channel in a marketing campaign
- Enriching an ecommerce catalog with product metadata

Analyzing the business

Many data scientists work on business data analysis for leadership decision support. Such use-cases include:

- Establishing activity forecasts
- Measuring the causal impact of events and programs
- Planning company resource allocation

What type of ML investment is your organization interested in? If you intend to do ML for internal analytics, chances are the ML will be mostly done by scientists and analysts, using interactive computing tools such as Jupyter notebooks. On the other hand, if ML will power an actual product used for internal automation or external revenue generation, you will also need software engineers to develop and operate the system. In the following section, I present the personas involved in ML projects.

I.4 ML practitioners: 13 job titles and counting

There are two learnings when it comes to ML job titles. First, it usually takes more than one person to ship an ML application to production. Not because it is difficult - modern ML platforms like Amazon SageMaker keep lowering the bar day after day - but because at the time of this writing it is rare to find the required combination of skills in a single person. Being able to write and train ML models is not sufficient to make customers happy. How about the service interface, documentation, security, availability, scalability and costs? In other words, hiring data scientists is rarely enough to build end-to-end ML applications. Chances are you will also need to hire an equal number of software developers to transform scientific code into stable production applications. The pattern of an ML team having at most half data scientists and at least half software developers is actually a common evidence of a team being mature in the ML space. From a data science job interview perspective, it is a warning sign if the team you are applying to has no developers nor data engineers: it could

signal a lack of production experience. This is not necessarily a bad thing, since you will be able to demonstrate your versatility and possibly grow a team if successful. Still, in a data science job interview I recommend candidates to always ask about team profiles and about the support possibilities for software development, automation, security and operations, in order to get a sense of how what the day-to-day job would look like.

Furthermore, ML job titles must be taken with a grain of salt. Not all ML practitioners have the *Data Scientist* job title, and both the knowledge and performance of individuals within a given job title vary considerably across teams and organizations. I once met a person with the *Business Analyst* title that was stronger with the Linux command line than with Excel and that was a better scientist than many data scientists I met since. I also met a number of data scientists that were predominantly using drag-and-drop tools, and other that were unfamiliar with most the ML concepts described in the part IV of this book. When you hire ML practitioners, work backward from your outcome expectations to design hiring materials representative of your needs. Your hires may not need knowledge of R, Hadoop or Microsoft Excel. The following 13 job titles are the ones I most frequently see working on ML projects. Note that I provide the associated definitions for directional guidance; they vary across organizations.

Software development engineer (SDE)

An SDE is an engineer that writes software. SDE presence in an ML project is an indicator that things are done seriously. SDEs have been creating production application for ages and have a lot to teach to ML practitioners, such as code versioning, testing, debugging, security, microservice design and scalability management. They work both upstream and downstream of scientists to develop data management systems and convert models into customer-facing software applications.

Data engineer (DE)

DEs process data from one shape to another. The first expectation for a DE is the ability to write batch data transformations with SQL. With the advent of the Apache Spark big data processing framework, it is common to expect from DEs comfort with PySpark and Scala. Strong DEs also master streaming environments.

Data scientist (DS)

The DS job starts with clean, machine-ready data and consists of fitting models on this data. Typical data formats include CSV, text, JSON, Apache Parquet or JPEG. In order to create ML models DS most often use specialized open-source machine learning code libraries that have a Python interface. In the vast majority of cases it is either Scikit-Learn, a gradient boosting variant (XGBoost, CatBoost, LightGBM), or deep learning development frameworks TensorFlow or PyTorch. Apache Spark and Apache MXNet are also seen in large-scale production-grade setups. The statistics language R is occasionally seen in the academia but in my experience is near non-existent in production. ML libraries often come with pre-written models and optimization logic hence DS rarely author ML science themselves. In particular, for neural network development data scientists often use high-level wrappers reducing code verbosity, such as fastai, PyTorch Lightning and Keras. Appendix A6 gives a sense of the respective popularity of all those options. Custom model development happens occasionally in order to invent or adapt a scientific logic to a set of constraints, for example scale, latency or hardware footprint. Although DS job descriptions sometimes ask DS to lead business discussions, own DS projects end-to-end, develop data processing pipeline from collection up to model serving, it is rare to meet DS this versatile and in practice their activity often centers on model development, training and validation.

Research Scientist (RS)

A DS variant with a focus on research and usually more expectations to publish. The focus on research and science generally means less infrastructure and software development skills than DS.

Applied Scientist (AS)

A DS variant with additional attention to the practical considerations of production. Tasks that an AS could do but that DS or RS are less likely to own include model serving engineering and management of code deployment pipelines.

Economist

Economists are a rare resource seen only at a handful companies such as Amazon, Netflix, Uber. One of their specialties is the estimation of the causal relationship between

treatments and metrics of interest. Their skillset is highly valued in data-driven companies eager to quantify the impact of their actions.

ML engineer (MLE)

A software engineer with a focus on ML and associated engineering considerations such as framework performance, continuous integration and deployment, software-hardware integration and model serving.

ML architect (MLA)

MLAs specialize in designing and auditing ML systems. They know which components fit well together, and which building blocks to associate to a given use-case.

Business intelligence engineer (BIE)

BIEs build clean datasets for ML consumption or direct business consumption. They use a variety of scripted and visual tools. BIEs usually use raw data systems created by SDEs and DEs on top of which they occasionally develop their own tools.

Business intelligence analyst (BIA)

Similar to BIE, with a closer proximity to business stakeholders. BIAs use the data engineering systems developed by SDEs, DEs and BIEs and rarely develop their own.

Data analyst (DA)

DAs use dashboards and extraction tools to provide data analysis to business stakeholders. They rarely develop systems.

Business analyst (BA)

In tech companies, BAs produce analytic narratives supporting reporting or decision making. They occasionally have the coding skills to extract the data necessary to their own analysis, and if they do not, they delegate it to people with data extraction skills, typically DAs, BIAs, BIEs.

Product manager (PM)

The PM orchestrates all stakeholders and makes sure roadmap items are economically sound and delivered on time. PMs also gather and own key product information such as

customer feedback and feature requests. Mature engineering companies may also hire technical product managers (TPM) that have a heavier focus on engineering roadmap than on its connection to business outcomes.

Those roles operate in an iteration cycle depicted in figure 5. Individuals with wide skillset may cover multiple of those roles. It is common to alternate across those roles throughout a career. Production ML offers multiple career development opportunities.

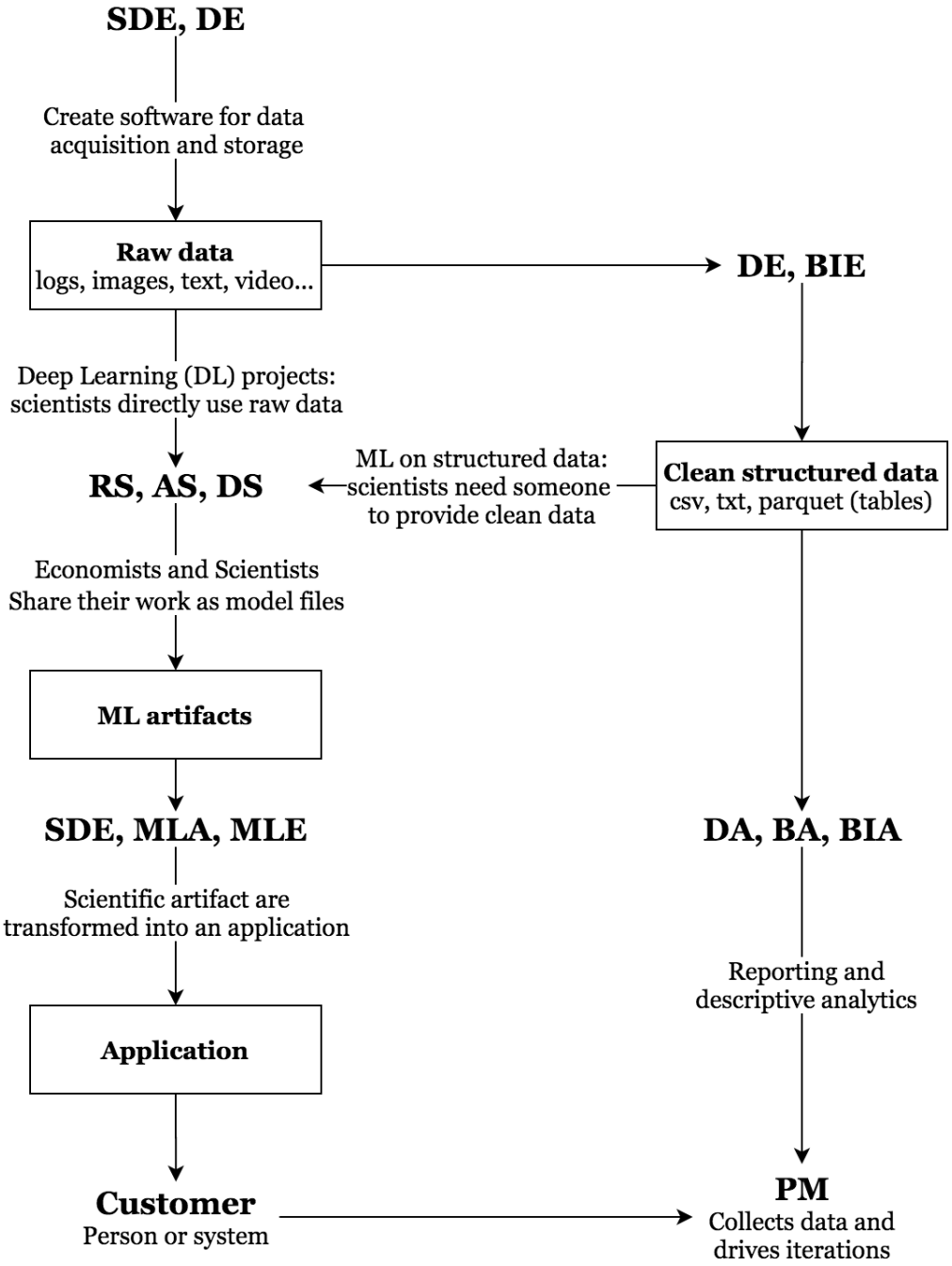

Figure 5 – ML practitioners: personas and interactions

Scientists are not the first and only hires you need. SDEs, DEs and BIEs are equally important in order to create tools for data ingestion, collection, inspection, monitoring, with the appropriate automation and security bar. Other personas, not displayed on this chart, are also important in production-grade ML projects: for example, business stakeholders that validate the relevancy of the task modelled and the data collected. In large-scale or sensitive use-cases, it is important to have access to legal experts, to help answer questions about regulation, appropriate ML and data use, and contractual relationships with partners and customers. Model ethics management is also a team work: ensuring that an ML system is fair and beneficial to the community requires interaction and understanding between technical teams, business owners, end-users and compliance experts.

You need more than data scientists to develop production ML systems
Companies that hire only data scientists to develop their ML initiatives may feel the frustration of the investment not paying off. Indeed, science is only a thin slice of an end-to-end ML project. Over-investing on scientists for production ML is equally frustrating for the scientists themselves, who end up in charge of significant non-scientific work such as handling data pipelines and operational issues. Production-grade data science requires a diverse skillset and is a team's work.

In Part I of the book, I demystified production-grade ML by providing definitions, success stories and organizational patterns. In Part II, I describe in depth the various components of an ML system.

II. Anatomy of an ML System

A trained ML model consists of one or multiple files representing model structure and parameters. It is usually in the kilobyte to megabyte scale, with rare instances in the multi-gigabyte. The model files alone are useless and need other components to make the model usable: at a minimum computing infrastructure and an execution environment. Those components connected together form an *ML System*. In the following sections I first detail the work cycle that produces model files, then explain the 2 workflows transforming model files into an ML system: the *system workflow* and the *iteration workflow*.

II.1 ML lifecycle

The creation of a machine learning model consists of the steps below:

1. **Problem definition**
2. **Data collection**
3. **Data exploration**
4. **Training code development**
5. **Training and tuning**
6. **Deployment**

Those steps are chained together in the graph depicted on figure 6.

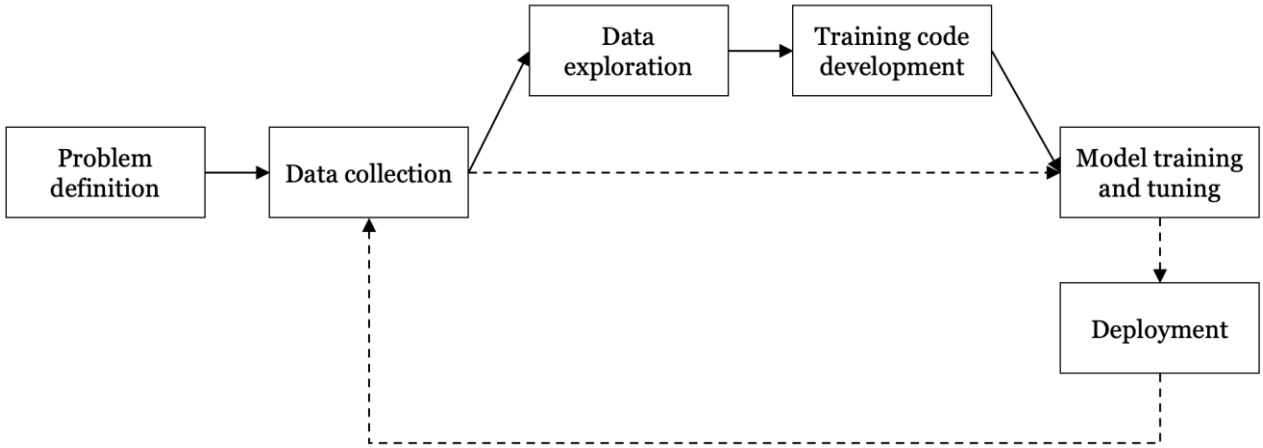

Figure 6. ML Lifecycle. The first iteration on the model, depicted with the full lines, is sequential and involves human effort, such as defining the problem, the data collection strategy, and initial training code development. Once a model is obtained, in can be operationalized in an automation-prone flywheel: its operations bring incremental data that can be added to the training data pool to produce new models.

Problem definition

Defining a business problem to solve is the most important task in the ML lifecycle. It breaks itself in a number of subtasks:

Validating business relevancy

Talking to business stakeholders to find a use-case that will create value for a customer while being fair, compliant, and beneficial to the planet and its diverse residents.

Finding a specific question to answer

Translating the business use-case in a question that an ML model can answer. For example, do you want to predict a class? A number? Learn a data transformation?

Setting a high-level target architecture

How will the system be used? In batch? In real-time? Embedded on a device? What are the latency, cost and hardware footprint constraints? System architecture is further discussed in the rest of this book.

Defining metrics

You need metrics to determine whether the project is successful and what are its areas of improvements.

Mistakes often occur at problem definition stage. Over the years, I even found that when ML projects fail it is almost always because of problem definition mistakes, and rarely because of technical issues downstream. Below are three problem definition mistakes I often witness:

Mistake 1: working on something that is not valuable for the business

Several machine learning models never make their way to production because they are useless, insofar as the question they answer is not relevant to the business. For example, predicting customer churn may be easy for an ML model, yet is it useful for your business peers? What actions are going to be taken upon model predictions? Wouldn't they prefer a full system that prevents churn, instead of a model that only predicts it? Always check business relevancy before coding. This problem often happens when a ML project is driven by technical curiosity more than business need, a risk described in the next paragraph.

Mistake 2: Starting with an algorithm instead of a business question

It is tempting for technology enthusiasts to start a project by applying the latest hot technique to their data. This however often leads to never-ending, slowly-decaying projects: the lack of overarching question and metrics to measure progress against makes it impossible to determine project success. An example of this is market segmentation projects. Because there is a whole field of ML dedicated to clustering, it is tempting to jump at them to cluster customers. Yet what is the point? Understanding the customer better? What does that mean? What is the metric for success? How would such segments be used? In the age of internet, connected devices and data deluge, customer segmentation can often be replaced by fine-grained personalization. Before launching a customer segmentation project, spend time defining the business goals and associated metrics. You may find that other ML paradigms can answer those questions directly.

Mistake 3: not having quantitative success criteria

If your project is not associated with quantitative goals and success metrics, you will never be able to track its progress and never be able to declare it successful. You will fly blind throughout all the project duration. Loose business requirements often cause the absence of

success metric, such as *"we'd like to try and see what ML can do on this"*, or *"We'd like an ML model that is better than our current system", or "we want a recommender system"*. It is critical that you push back on unclear business requirements and insist on quantifying expectations. If your business stakeholders cannot help you with those numbers, then try and fit in their shoes to figure out priorities and gently make proactive proposals. In which envelope should costs, latency and model quality fit? Are false negatives more costly than false positives? Is there a current baseline to compare against? Is there a business metric the model should improve?

Data collection

After problem definition, data needs to be gathered for model training and inference.

How to select a data source?

Data source selection can make or break an ML project. Firstly, variables used for training must be available at prediction time. For example, if you develop a model to predict a daily commodity price based on the weather of the same day (fictional example), your model may perform well on historical data but be unable to predict tomorrow's prices: it would need to know tomorrow's weather, something probably equally hard to predict in the first place. Similarly, if you train a model for real-time ad scoring using extra features brought from an offline source, make sure that this source will be available at inference. Furthermore, data availability and quality may change over time. I once had to restart an ML project from scratch because my data source was discontinued… Conduct appropriate due diligence to sense if a data source is available and reliable enough for you to use it.

The two data collection philosophies

There are two mindsets for ML data collection: acquiring what you want or using what you have. I have a preference for the second philosophy, which leads to frugal and agile projects.

Training on the features you want

We sometimes believe to know which exact data to use on a given problem. For example, to train a house price estimate model it is natural to believe that house size, number of rooms, geolocation and age of construction could be good features to include to the training dataset. That means I would have to spend a lot of time studying possible data

sources and writing code to extract the data and format it into the appropriate dataset. And if some of those dreamed features are not readily available, I would need to find ways to get them. For example, if I want to use the walls material as a feature but do not have it in my data lake, I would need to purchase the data or find an acceptable public source for it, or even create a separate ML model to guess the data. This grocery-list approach to data collection represents a down payment at the beginning of the project, that has uncertain return-on-investment: you do not know the value of a feature until you trained a model on it.

Training on the features you have

Instead of starting your project by making a grocery list of your dream dataset, you can think frugally and work backwards from the data you have readily available. For example, in the fictional above-mentioned house valuation scenario, if all you have is city name and pictures of the houses, you could start from there with one or multiple baseline models regressing price from pictures, and then expand the dataset step by step with tweaks you believe would improve accuracy while not overly increasing costs and complexity.

In practice, ML practitioners often combine the two approaches. They receive insights from business stakeholders on which features to use for a given task, which guide initial data collection. If some features are readily available in the system it does not hurt to add them to the dataset and see if the model picks some relevant signal from them.

Data annotation

Supervised ML algorithms learn a function between an input and an output by analyzing input-output pairs. If there are not enough historical input-output samples, they need to be generated. One way to create input-output samples is to have humans labelling inputs with a desired output, a task called *data labelling* or *data annotation*. Data annotation is sometimes precisely the value-creating mechanism, when annotators enhance the raw data with rare, specialized knowledge. This is for example the case in medical imagery analysis, animal identification, artistic curation or vessel acoustic classification. There are several questions to answer when designing a dataset annotation workflow:

- **Who?** Who will annotate your data? Is it acceptable to crowdsource the task to the public, or do you want to restrict who produces the label, for expertise or confidentiality reasons?

- **How?** How are the annotators producing the labels? Which software and experience can you use so that they are as productive and accurate as possible?
- **What?** What specific records need to be annotated? Could you possibly annotate only the minimum records required to make your model accurate enough?

There are numerous tools for data annotation, and my favorite is Amazon SageMaker Ground Truth. It allows developer to answer the 3 questions above, providing both (1) flexible workforce control (private, vendor, public), (2) customizable interfaces for a variety of tasks and (3) active learning capacities so that easy-to-annotate images can get processed by a model trained on early human annotations, thereby reducing total annotation cost.

Is data annotation a mandatory step in the ML lifecycle?

Of course, no! First, there are many models called *unsupervised* that learn a transformation without modelling a specific output. Such models include clustering, dimensionality reduction and some variants of anomaly detection. Furthermore, in some instances of supervised learning, you can use raw historical events as labels, for example in time series forecasting or recommender systems. Besides, it is sometimes possible to find naturally-occurring labelled data within your business: for example, social networks can use post tags as labels to train image classifiers and text classifiers. For example, in *Exploring the Limits of Weakly Supervised Pretraining*, Facebook researchers Mahajan et al. show that pre-training an image classifier on billions of Instagram tagged images vastly improves the accuracy of the classification quality on a different task.

Feature engineering

Many models cannot make sense of an input that is too large or insufficiently correlated to the target output. Consequently, data collection is often followed by a *feature engineering* step during which a human writes data transformations to create variables believed to be more correlated to the target. Feature engineering used to be the most time-consuming step in the model lifecycle, requiring both intuition and skill. With the advent of AutoML and deep learning, manual feature engineering is less and less needed. DL models compute their own internal intermediary data representations as part of training hence they can work directly on raw data such as text or images. In that regard, DL requires much less feature engineering, and the typical saying that “ML is 80% data engineering and 20% model development” is reversed, as depicted in figure 7.

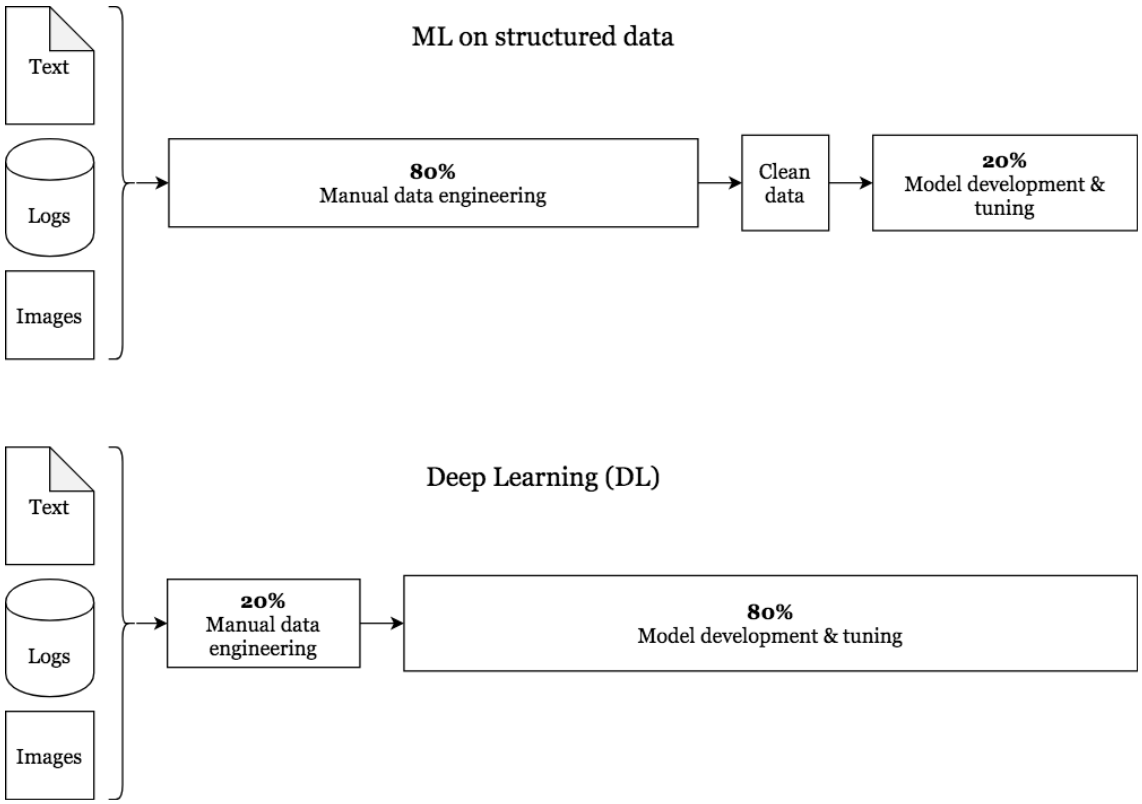

Figure 7. High-level comparison between ML and DL from a data engineering standpoint

Data exploration

Once model-ready data is available, the ML practitioner spends a couple hours to a couple days exploring it, notably: (1) inspecting its properties, (2) looking for anomalies and edge cases, (3) gathering summary statistics to document the problem. Data collection, data exploration and feature engineering occasionally happen in a cyclical fashion: exploration provides extra modelling ideas, that may require additional data collection and engineering work.

Training code development

Writing the code that executes the training of an ML model is usually the easiest part of the ML lifecycle, sometimes taking as little as few minutes. Model training is abundantly documented and facilitated by the fast-developing ecosystem. Several software packages provide pre-written models and associated training code, such as Scikit-Learn and XGBoost. Even easier to use, commercial APIs such as Amazon Personalize (recommender systems) and Amazon Forecast (time series forecasting) do not require developers to write any

scientific code. Furthermore, a number of graphical interfaces open ML access to non-coders, such as Dataiku, KNIME, BigML or RapidMiner. Authoring training code gets non-trivial in deep learning, where it is often required to write custom data ingestion and training instructions to adapt off-the-shelf architectures. Advanced ML practitioners occasionally author fully custom code, tailored to their use-case constraints and objectives.

Training and tuning

Model training refers to the phase during which the model finds its coefficients by diminishing its error on provided data. Training time varies with the data size, model architecture, computing hardware and precision goals. It can be fast: fitting a linear regression on a tabular dataset of few hundred values takes less than a second on a consumer laptop. I rarely met models needing more than couple hours to train. ML publications occasionally describe multi-day, multi-thousand-dollars trainings, but those are the exception more than the rule. Model tuning refers to finding a satisfying combination of model hyperparameters. Hyperparameters are design invariants – constants or architectural decisions - that developers choose out of intuition or by brute force search. An ML model can search and find hyperparameters for another model. For example, Bayesian optimization is a family of models that intelligently browse and test hyperparameters combinations of other models, to find satisfying hyperparameters faster than via brute force. Part IV.6 of the book further discusses hyperparameter tuning.

Model Deployment

A trained model must be part of a business-facing application to create value. As I explained in Part I of the book, this application could either be a customer-facing product, or an internal system for analytics or automation. There are two manners of interaction with a model, and those guide the deployment style.

Batch invocation

In that scenario the model runs periodically, on demand or on a schedule, to process a group of inputs. A real-life example of a batch-invocated model is the sea floor analysis model presented by Fugro at AWS re:Invent 2019 (*Using machine learning to find boulders on the*

sea floor, Al-Saadoon et al), where batches of sea floor images are regularly processed in the cloud for boulder detection.

Real-time invocation

In this scenario the model is permanently reachable and makes prediction on-demand, as fast as possible, often for a single record at a time. Examples of real-time inference deployments include advertising scoring systems that compute click probability of ads in real-time, fraud detection system that score live transaction risks or visual quality checking models in factories. In real-time invocation, latency and availability generally are the priorities hence it is common to locate the model as close as possible to its point of consumption, which could be an embedded device or a smartphone.

After model deployment, data needs to be collected for later model training and application health monitoring. This fresh data, along with customer and business feedback, drives new iteration cycles. In the next section, I present the key architectural components of an ML system, beyond the model.

II.2 ML system workflow

In order to transform an ML model into a production application, several additional components are needed.

Computing infrastructure for ML training and inference

The training and prediction logic of an ML model consist of code files. Those text files are useless on their own: they need an underlying machine to run. The execution infrastructure can consist of physical local computers, devices, or remote virtual servers hosted in the cloud. Amazon SageMaker is a popular commercial option for ML computing infrastructure, using Amazon EC2 virtual machines as a computing resource. For tasks fitting in 10 Gb of memory and 15min of execution time, the micro-virtual machines service AWS Lambda - a compute paradigm called *serverless* because the developer has no server to manage - provides outstanding economics. Organizations report successfully using AWS Lambda to run ML inference, including for deep vision models. For example, the fashion computer vision startup Curalate wrote in a 2019 AWS ML Blog post successfully deploying ResNet-152 image

classification models in Lambda (*Serving deep learning at Curalate with Apache MXNet, AWS Lambda, and Amazon Elastic Inference*, Jesse Brizzi). Curalate found that deployment was cheaper in Lambda than in permanent EC2 servers up to 7.5MM requests a month. Equally impressive, Ciaran Evans from the UK Hydrographic Office describes in a 2020 blog post a serverless aerial mangrove analysis pipeline, running TensorFlow UNet segmentation models in AWS Lambda (*Creating a global dataset: using serverless applications in deep learning*, Ciaran Evans)

Computing infrastructure for non-ML data transformations

Ad-hoc, hard-coded data transformations are numerous upstream and downstream of ML models: learning set preparation, feature engineering, offline validation analysis, batch inference results analysis, among others. Those transformations are written in Python, SQL or Scala, depending on the expressivity required. They are sometimes called *Extract, Transform, Load* (ETL) transformations. In the AWS cloud, Amazon SageMaker is a good place to run ETL workloads. Dedicated data processing APIs such as Amazon EMR, AWS Glue (serverless Spark), Amazon Athena (serverless Presto SQL) and still AWS Lambda are also relevant. In the open-source, Apache Spark, Dask and Pandas are the most popular data processing options.

Pandas is the most popular open-source data analysis library. It enables the manipulation and analysis of tabular data and works well with CSV files, Excel workbooks and more. Pandas is easy to learn and runs equally well on personal computers and large servers; it is occasionally branded as a scripted alternative to Microsoft Excel. Pandas however suffers from two limitations: (1) being written primarily in Python, some of its functions run in a single thread of instructions and do not utilize all the cores of a machine, (2) it has been primarily developed for a single machine use and requires some extra development to be used in a distributed environment. Because of both those frictions, Pandas struggles to handle datasets in the multi-gigabyte scale, an area where the distributed computing framework Apache Spark becomes relevant.

A storage system

Storage is necessary to save and load files such as model coefficients, batch input and output data, monitoring results and configuration. Object storage such as Amazon S3 is appropriate for this.

An orchestration system

A task orchestrator is a sophisticated name for a system that enables the creation, automation, scheduling and organization of computing tasks. Orchestrators are useful abstractions to define the flow of data from component to component. Popular options for this are Apache Airflow (open-source) and AWS Step Functions (commercial).

A monitoring system

A log collection and analysis system is helpful to audit performance, collect metrics and inspect anomalies. Developing this component in the AWS cloud is vastly facilitated by the native connection many services have with the monitoring service Amazon CloudWatch.

Figure 8 summarizes the components of an ML system (use-case agnostic)

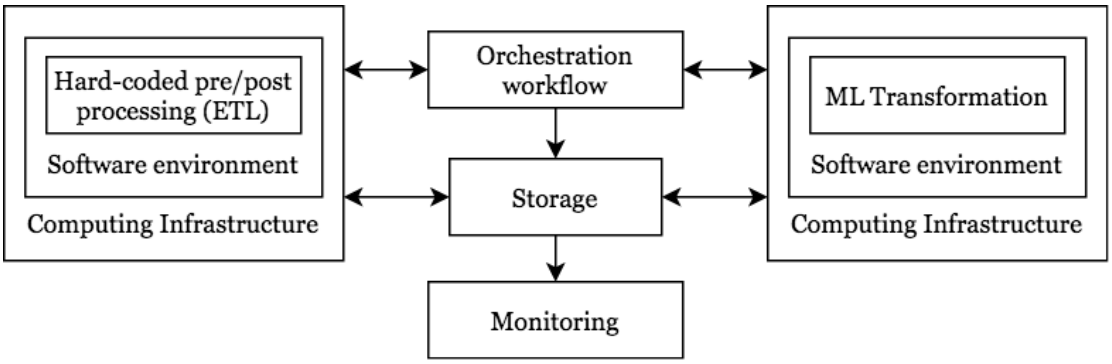

Figure 8 – Building blocks of an ML system.

ML practitioners may be responsible for a subset of the system only. The first thing to do as you join or start an ML project is to establish a contract with your customer, internal or external to your organization, and define what is the product you should deliver. It could be:

- A trained model file for a team to take over and deploy as an API.
- A file containing prediction or scores from an ML model that a team will use in its system.
- A Docker container image, for example a SageMaker hosting image containing the libraries and logic necessary for a downstream team to serve your model in their infrastructure.

- A multi-component complex system.

Once you know your expected outcome, you need to put in place a development workflow that makes you deliver result fast, while maintaining a high bar on quality and security. In the rest of this book, I interchangeably call it the *iteration workflow* or the *Continuous Integration, Continuous Delivery (CICD) workflow*. CICD automates the transformation of source code into products. It is a mature practice in traditional software engineering, that organizations are extending to machine learning projects. Part II.3 below provides an overview of ML iteration workflow, and part IV.3 dives deeper into CICD applied to ML. For more information on CICD, you can browse the associated AWS documentation: *What is Continuous Integration? What is Continuous Delivery?*

II.3 Iteration workflow

An ML system permanently evolves. Customer feedback, bug reports and business roadmap drive feature requests at all levels. It is important to build from day one a system that supports fast, safe, continuous iteration. This is enabled by the extra concepts and components detailed below.

Code versioning

Versioning the code is a necessary step so that system iterations are visible, can be audited, reproduced and rolled back if need be. Code versioning platforms also enable productive collaboration by structuring the interaction of multiple authors and facilitating peer reviews. I mostly saw code versioning done with the open-source version control system (VCS) Git. Git has several popular commercial implementations, for example Microsoft-owned GitHub, Gitlab, Atlassian Bitbucket and AWS CodeCommit.

ML training job versioning

The execution of ML training code on training data produces an ML model. In order to make this process reproducible, you can persist the triplet constituted of ML model, ML code, which may contain randomness controls to freeze, and training data. Such a triplet constitutes an

ML experiment or *ML job*. ML models can be saved to object storage such as Amazon S3. ML code can be saved in version control along with configuration files if any. Finally, the data can be saved as files on object storage. The experiment tuple consists of versioned identifiers for model, code and data, and can be stored in a database called the *experiment store*. One of my favorite features of Amazon SageMaker Training is that it comes with a searchable serverless managed experiment store. In the open-source, MLFlow is the most popular library for ML experiment management.

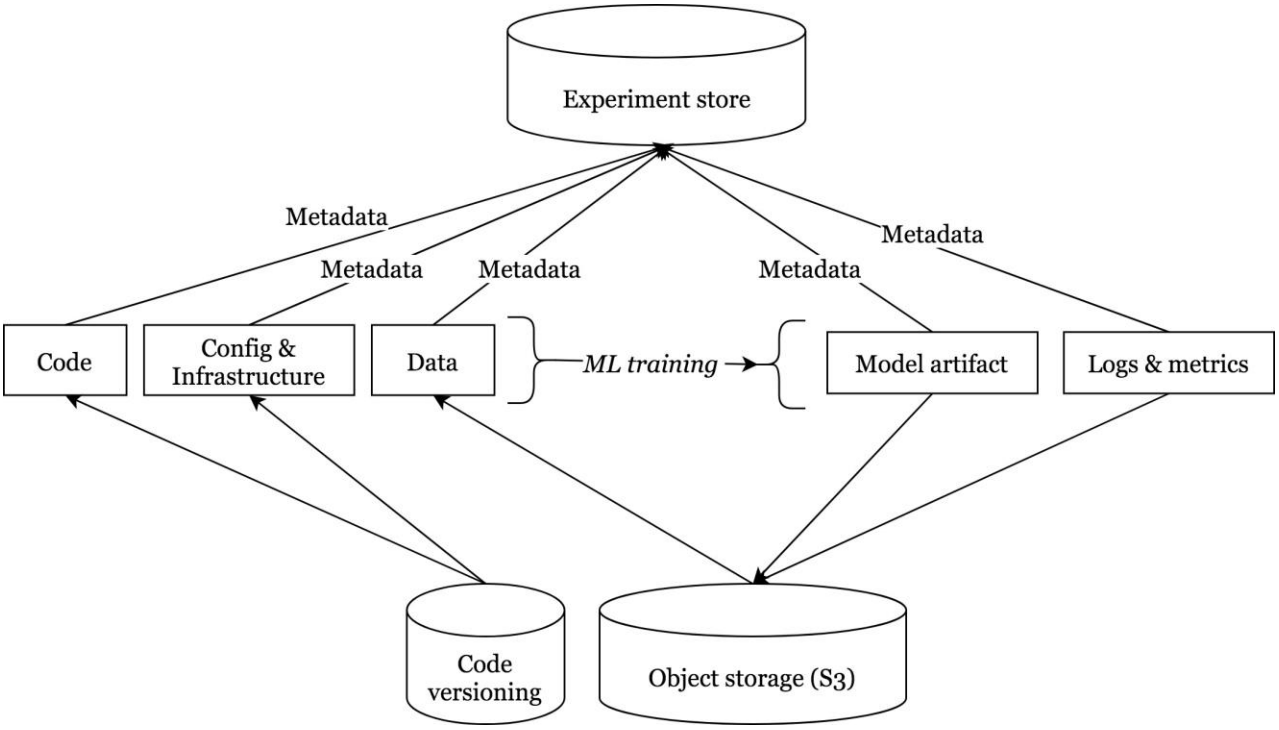

Figure 9. To enable training reproducibility, you need to version all the moving parts: code, configuration, infrastructure and data. The experiment store tracks all associated metadata

How to version datasets?

Dataset versioning is still an open problem that has multiple answers depending on data format and use-case constraints. The easiest form of versioning is to persist the full dataset with a given ML experiment. This can however become unacceptably expensive or and slow as data grows. A more frugal way to version a dataset, when it is constituted of multiple large objects (videos, images, songs…) is to keep an immutable copy of each object in object storage and persist, for a given ML experiment, a file containing the list of objects used in the experiment. Those object lists, sometimes called *manifests* are smaller and easier to

version than the data itself. DVC is an open-source dataset versioning tool that adopts a similar mindset to the previously described manifest concept. It stores large files in object storage and maintains in code versioning smaller metadata files tracking dataset composition. DVC is promising, however as of May 2021 I have not seen it in production yet.

CICD pipeline

The CICD pipeline is a semi-automated workflow that converts source code into a product available to your customers, be they external or external to your organization. Thanks to automation, the CICD pipeline helps you maintain the bar high in quality and in delivery speed. At a high-level it usually consists of two steps, the *build* and the *deployment*. The *build* transforms human-written code into application artifacts, and the *deployment* transforms those artifacts into customer-facing products. Several tools support the creation and operation of CICD pipelines, such as Amazon CodePipeline, Jenkins, TravisCI, CircleCI. You can also automate your ML software integration and delivery with standard workflow schedulers. CICD is the link between your code and your customers and is recommended knowledge to have in production ML projects. CICD pipelines can either be developed and maintained by partner teams working in tandem with ML practitioners or by ML practitioners themselves.

Isolated environments

Isolated environments limit the impact of problems. It is advised to isolate research from production and to test in a dedicated environment before launching in production.

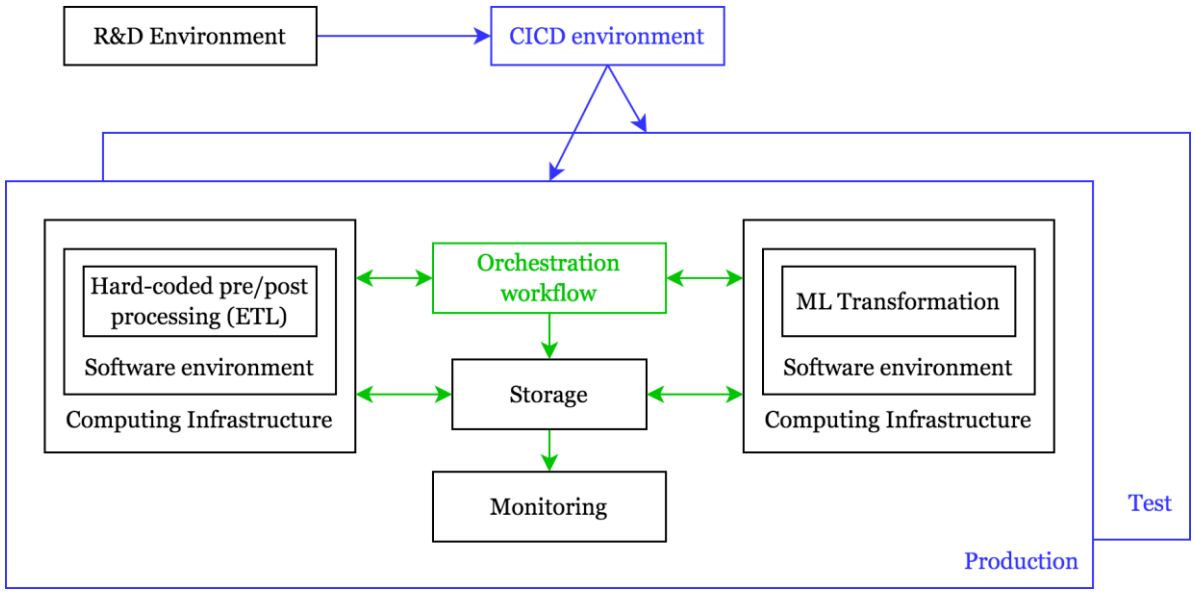

Figure 10. System workflow and iteration workflow. The system workflow (in green) animates the system within a given system version. The iteration workflow (in blue), also called CICD workflow, animates the system across versions. When ML artifacts are frozen within a system version, the ML training can be removed from the product workflow and be part of the CICD. Part IV.3 further dives in ML CICD.

A production ML system is animated by two orthogonal workflows: (1) the *system workflow*, that I also call *product workflow* or *in-version workflow*, connects the various components together within a system version and (2) the *CICD workflow*, that I also call *iteration workflow* or *cross-versions workflow*, consists of the steps that allow to transform source code into an application and iterate across system versions. Historically, those workflows have been supported by different tools. In 2020, AWS released the SageMaker Pipelines suite, which supports both workflows in a single framework. CICD knowledge is a common gap among ML organizations, and a painful one since its absence prevents ML practitioners from shipping their work to production. It is a valuable, differentiating skill for ML practitioners to develop. It is further discussed in the part IV.3 of the book.

In the Part II of this book, I detailed the specific phases of an ML project and the constituents of an ML system. In Part III I present the baseline knowledge to successfully execute production-grade ML.

III. Baseline skills for the ML practitioner

In this section, I present skills that I believe necessary for an ML practitioner to make a positive impact. I do not dive on the mathematics of machine learning for three reasons: first, I expect the reader to be familiar with the required fundamentals, which I consider to be linear algebra (vector, distances) and analysis (derivatives, exponential, logarithm). Second, several other books already cover ML science well, like the excellent *Dive into Deep Learning* by Zhang et al. Finally, as quality science is getting abstracted into open-source and commercial ML tools, it is increasingly possible for practitioners to develop ML applications without expertise of the underlying mathematics.

III.1 Core behaviors

ML practitioners need more than technical skills to be successful. I found the following behaviors to positively impact organizations, businesses health and individual careers.

Customer-focus

The most important behavioral trait to acquire and refine throughout your career is customer focus. Customer focus will contribute to your success in every career path, and machine learning is no different. Customer-focus takes several shapes in the ML practitioner's daily job. In the below list, the word *customer* refers to both internal customers and external customers.

Being a great listener

> Suspend your ego when interacting with customers and listen to them. Collect and document their goals, frictions and pain points. This will help you produce work that makes them successful.

Being a crisp communicator

Customers may not share your level of comfort with your product. Regularly and proactively communicate with your customers in their own language.

Asking the "5 Whys"

The "5 Whys" is a problem-solving tool that consists of recursively asking "why" five times to find the root cause of a problem. It is also useful to distill customer requests into their essential, underlying needs. See how effective it is in the following fictional conversation:

- *CUSTOMER: "Could you help me build an ML platform on Kubernetes?"*
- *ML PRACTITIONER: "Why?"*
- *CUSTOMER: "We would like to use Kubeflow"*
- *ML PRACTITIONER: "Why?"*
- *CUSTOMER: "We would like to use the Katib hyperparameter search"*
- *ML PRACTITIONER: "Why?"*
- *CUSTOMER: "We are interested in Bayesian optimization for model tuning"*
- *ML PRACTITIONER: "How about SageMaker Model Tuning? It supports Bayesian optimization, with the additional benefits of managed hardware provisioning and a serverless metadata store"*
- *CUSTOMER: "Sounds good! Tell me more about it"*

Humility to start simple

Do not force yourself to implement the latest sophisticated model. In production-grade ML, simpler is better, from at least three perspectives:

Simple systems are investor-friendly

The less computation a model needs, the cheaper and faster it is. Therefore low-latency, high-traffic applications often run linear models.

Simple systems are customer-friendly

The simpler a model is, the easier it will be to pitch it to customers and regulators.

Simple systems are operations-friendly

The simpler a model is, the faster it will make its way to production. The more components a distributed system has, the tougher it is to operate and diagnose. Netflix mentioned not deploying the winning model of its $1MM recommendation challenge because of its complexity (*Netflix Recommendations: Beyond the 5 stars*, Amatriain and Basilico) among other things. Similarly, despite their popularity in Kaggle data science competitions, I never read about ensembles and models of models used in production.

Production obsession

The fast development of open-source and commercial ML development toolkits opens to everyone the excitement of creating learning algorithms. It is easy to fit models on data in a laptop or a cloud computing instance. However, customers rarely care about an offline, silent model. Models start creating value when integrated into functional, durable and available applications. ML practitioners must keep in mind that the customer clock is permanently ticking. Laptop-hosted proofs of concept (POCs), summary visualizations and research papers are not actionable by your customer, except when they are precisely your expected deliverable, which is less and less frequent. You must strive to turn your ML research into an actionable service or product useful to your customer. Shaping your ideas in a customer facing format as early as possible is also a great way to quickly get feedback and iterate.

Appetite for learning

I will always remember the time *transformers* and pre-trained language models changed the face of Natural Language Processing (NLP). Within couple years, NLP techniques that were taught to a generation of practitioners – such as TF-IDF or Naïve Bayes – got outdated by token embeddings and attention layers. Computer vision and time series went through a similar transition. Technologies change fast, and the successful ML practitioner must be permanently learning to stay relevant.

Data-driven thinking at all levels

In France we have a popular saying that says “les cordonniers sont les plus mal chaussés” - which I recently found to exist also in English - “the shoemaker's son always goes barefoot". This saying applies equally well to data scientists and ML practitioners, who routinely apply ML to other people’s problems but never to their own. I often discuss with ML practitioners

that spend months developing sophisticated models, yet have no idea how to do goal setting, hardware sizing, infrastructure tuning or monitoring. It turns out that the problems faced by ML practitioners can often be solved with ML itself! The periphery of an ML project is full of opportunities to apply ML:

- How do you match a given algorithm to training hardware? Use a recommender
- How to you predict training cost and duration? Use a regression model
- How do you tune your system configuration? Use Bayesian optimization
- How do you predict traffic for your service? Use timeseries forecasting
- How do you balance traffic between model versions? Use a multi-arm bandit
- How do you detect problems? Use unsupervised anomaly detection over your logs

III.2 Security

Security is the job of everyone, including the ML practitioner. Security is important in at least the three following aspects.

Customer trust
Customers and end users expect the respect of their privacy and want their assets, including their data, not to be misplaced or misused. Security issues alter customer trust, which leads to customer churn, in addition to possible compliance risks.

Compliance with laws and regulations
Laws and regulations govern several online activities. Such frameworks include the General Data Protection Regulation (GDPR), the Health Insurance Portability and Accountability Act (HIPAA), the Payment Card Industry Data Security Standard (PCI DSS), among others. Non-compliance with laws can significantly impact organizations and individuals.

Asset protection
Independently of considerations for laws and regulation, it is in any organization's best interests to protect its assets, including assets its customers and employees entrusted it with.

You may build the fastest ML system or the most accurate image classifier, yet if it can be accessed and altered by anyone, it will lose its value and possibly propagate issues to the rest of your services, customers and assets.

Technology organizations should heavily invest in security mechanisms. Security fosters productivity. Firstly, when programmatic, transparent security mechanisms are in place the security dimension of projects is easier to anticipate and address. Secondly, when applications are built with an emphasis on security less issues happen and employees can focus on core business. As an ML practitioner, applying the 10 following practices will increase the quality of your work and make your organization stronger:

1. **Know and apply the security rules** and policies in place at your organization. Better: know the laws and regulation applicable to your industry.
2. **Protect data** with encryption at rest and in transit.
3. **Review your data anonymization requirements**. If you need to anonymize data, choose a technique adapted to the data and to the requirements. Just removing personal identifiers may not be enough to make the data anonymous. For example, in 2006 a major consumer internet company publicly released a search dataset for research purposes. While the data contained no direct identifiers, the search queries themselves contained identifying data. Though the dataset was left online only 3 days, it was mirrored and later led to abundant research including revelation of some user identities.
4. **Deny access by default** and provide granular access on a **least privilege basis.** Grant no more access to users and systems than what they need. Data access permissions should be personalized by role: BIE, DS and SDE do not necessarily need access to the same files, datasets, or even columns within a dataset.
5. **Publish with extreme care** on GitHub and blog posts. Many security incidents start with accidental publication of credentials or other confidential data.
6. **Review** your peers' work and ask them to review your work.
7. **Never trust external tools and parties**. Do not provide sensitive data to a third-party software or company unless it has been deemed safe to work with by your security team or your local external party security validation process. This vetting process should also apply to open-source software. If your organization has no external party security validation process, work to get one in place.

8. **Escalate quickly** security doubts to your leadership and to security stakeholders.
9. **Treat ML models like sensitive data.** Be thoughtful in how you expose the model artifact and its APIs. Models and ML APIs should be given appropriate protection such as access control, throttling, results post-processing, to reduce the risk of knowledge extraction, either about the training data or about the model.

III.3 Programming

You can embrace a career in machine learning without being a programmer. Firstly, numerous job opportunities do not require coding: product managers, business developers, account managers, team leaders – many people use machine learning to create value without writing one line of code. Secondly, several graphical tools exist for non-developer ML practitioners, for example Dataiku, KNIME, BigML or RapidMiner. Their intuitive interfaces allow ML development without code; they are frequently seen in the field for exploratory research and prototyping. That being said, if you aspire to work in technical ML roles, it is beneficial to learn and use programming languages. I found that developing systems using code has several additional benefits over drag-and-drop interfaces, detailed below.

Scalability & reusability

With experience, writing and running code is faster than navigating a screen with a mouse. This is true with ML and data processing in general: for example, drawing 20 bar-chart diagrams in Microsoft Excel will probably take you half an hour or so. Generating those 20 bar-charts with a Python visualization library like Pandas or Seaborn could take as little as few seconds with a `for` loop. Logic written as code can easily scale up and be reproduced. It is more difficult to obtain those properties with drag-and-drop workflows.

Quality

Programs are text files. They are easy to version, review, comment, test and port across systems. Collaboration and quality verification are easier.

Customization

Code enables customization. This has positive impact on system quality, speed and cost.

A piece of code consists of algorithmic logic written in a programming language. You need to have baseline understanding of both in order to embrace a technical career in ML. An algorithm is a set of instructions that can be described in plain English, like the `for` loop or recursion. A programming language is a technology allowing to write algorithm scripts that can be executed on computers. Algorithms are immutable theoretical concepts that programming languages bring to life. Languages evolve permanently and are easy to learn when you are familiar with algorithmic concepts. Baseline algorithmic knowledge that will facilitate your technical learning curve includes data structures (list, dictionaries, trees, graphs), loops, conditions, classes. Baseline programming languages that you will most probably need as you embrace technical ML responsibilities include Linux command line, Git and Python. The command line is a code interface for UNIX and Linux-based machines. Such machines include Apple computers and many datacenter and cloud machines. ML practitioners like the command line, which helps them write automation scripts and interact with remote machines. Git is the most popular programming interface for code versioning. Remarkably, both Git and Linux were created by the Finnish-American programmer Linus Torvalds. Python is a wise learning choice that saw tremendous adoption by the ML community for numerous reasons: (1) it is lean and intuitive - in my experience the easiest language to learn, not too different from natural language, (2) it has extensive tooling for scientific computation and (3) it has extensive tooling for web and cloud development. Most modern ML development frameworks have a Python interface. I list in appendix A5 20 Python libraries I encourage ML practitioners to be familiar with.

Python was developed with simplicity as a priority in order to provide the best developer experience as possible. Lower-level languages such as C++, Rust or Java may give better performance, at the cost of more verbose development. To combine the best of both worlds it is common to see mission-critical, core components developed in C++ and their developer-facing interface developed in Python. This produces tools that are both performant and easy to use. Such libraries with a C++ backend and a Python frontend include the two nearest-neighbor search libraries Annoy and Facebook AI Similarity Search (FAISS) and the two deep learning frameworks Apache MXNet and TensorFlow. Organizations that program custom ML algorithms from scratch value C++ skills.

III.4 ML Algorithms

Algorithms are the visible end of the ML iceberg, on which academic research, community blogs and learning materials focus. The diversity, quantity and complexity of ML algorithms is a roadblock on the path to ML mastery: newcomers to the field are discouraged by the steep learning curve they believe must be climbed. Indeed, popular ML packages contain a bewildering abundance of models: Scikit-Learn's landing page features more than 60 algorithms, GluonCV features 270 computer vision architectures and HuggingFace more than 9,000 NLP models! Not all of them are worth attention. In fact, top ML practitioners use a tiny subset of them in their daily job. What makes an algorithm implementation useful in production are its simplicity, speed, cost, accuracy, portability and explainability. Algorithms rarely have all those qualities at the same time and the ML practitioner constantly makes trade-offs between simplicity and accuracy, speed and portability or between costs and accuracy. Those choices are easy to make if the model answers a quantitative business question with objectives and priorities. Some algorithms combine more qualities than others and are rewarded with wider adoption.

How many ML algorithm should ML practitioners know?
I recommend to have good knowledge of a minimal, compact toolbox of twenty or so complementary algorithms that passed the test of time or that you already saw in production. When choosing an ML algorithm, consider your own metrics of success beyond algorithm precision. There is no point using a the latest state-of-the-art, high-accuracy algorithm if it is too slow or too expensive to deploy. Consider all constraints of your real-world situation. Accuracy rarely is the only one.

I provide in figure 11 my personal shortlist of algorithms and learning paradigms that I recommend my mentees to be familiar with. Those algorithms are often seen in public real-life use-cases, and are strong baselines covering nearly all the use-cases I encounter. As an optional read I provide in Appendix A1 an overview each algorithm along with implementation tips and real-life public use-cases.

<table>
<tr><th>Type of input data</th><th>Learning paradigm</th><th>If speed is a priority</th><th>If accuracy is a priority</th></tr>
<tr><td rowspan="4">Tabular</td><td>Classification and regression</td><td>Linear and logistic regression</td><td>Random forest
XGBoost and variants
AutoGluon</td></tr>
<tr><td>Similarity search</td><td colspan="2">HNSW</td></tr>
<tr><td>Anomaly detection</td><td colspan="2">Random Cut Forest</td></tr>
<tr><td rowspan="2">Clustering</td><td colspan="2">DBSCAN</td></tr>
<tr><td colspan="2">K-Means</td></tr>
<tr><td rowspan="4">Images</td><td>Image Classification</td><td>MobileNetV3</td><td>ResNet</td></tr>
<tr><td>Detection</td><td>YoloV3, SSD</td><td>FasterRCNN</td></tr>
<tr><td>Video Classification</td><td colspan="2">I3D</td></tr>
<tr><td>Segmentation</td><td colspan="2">DeepLabV3</td></tr>
<tr><td rowspan="3">Text & sequences of tokens</td><td>Classification</td><td>BlazingText</td><td>BERT and variants</td></tr>
<tr><td>Token representation</td><td colspan="2">Word2vec</td></tr>
<tr><td>Sequence representation</td><td colspan="2">LDA</td></tr>
<tr><td></td><td>Scoring pairs of tokens</td><td colspan="2">Matrix Factorization and deep variants</td></tr>
<tr><td>Time series</td><td>Forecasting</td><td colspan="2">DeepAR</td></tr>
</table>

Figure 11. Key learning paradigms and algorithms I recommend my mentees to be aware of. They are not necessarily state of the art, but are strong baselines. Algorithm in bold are algorithms that I already saw documented in public real-life use-cases. References are in Appendix A1

In addition to the algorithms pictured in Figure 11, two generic algorithmic concepts agnostic of the data type often appear in ML literature: **autoencoders** are custom neural networks learning fixed-size numerical representations from abstract input. Numerical representations learned from autoencoders are useful for similarity search and anomaly detection. Furthermore, the **multi-arm bandit** is an algorithm deciding which option to select in a portfolio of N choices, in order to maximize a metric of interest. Thompson Sampling, by the eponymous scientist, is a popular technique to answer the multi-arm bandit problem.

Anybody comfortable with elementary math, security, programming and above-mentioned algorithms can apply to ML positions with reasonable odds of being recruited. However, being recruited in a job does not necessarily result in being successful in that job. In Part IV of this book, I present differentiating ML skills that bridge the gap from proof-of-concept to cost-effective, impactful production and foster individual and organizational success.

IV Differentiating Skills to Increase your Impact

In this fourth and last part of the book, I list skills that make the difference in a machine learning project and in a machine learning career. Knowing those concepts will help you handle the end-to-end machine learning project lifecycle and create value.

1. **Systems Architecture**
2. **Cloud computing**
3. **CICD**
4. **Baseline GPU knowledge**
5. **Neural network training optimization**
6. **Bayesian search of hyperparameters**
7. **Parallelism**
8. **Real-time Inference optimization**
9. **Causal inference**

Skills enabling the transformation of ML code in a production system are often categorized under the *MLOps* domain. In the list above, systems architecture, cloud computing, CICD, real-time inference optimization could be seen as specific MLOps facets. MLOps definition varies from party to party, because the concept is recent and production ML in its infancy. when two or more persons mention MLOps, they may have different things in mind, which can have detrimental consequences when scoping a project or pitching a service to a customer. In order to reduce the risk of misunderstanding, I encourage readers to use more specific words when possible, such as *CICD*, *model serving*, *model monitoring, security,* etc.

IV.1. Systems Architecture

Why does it matter for the ML practitioner?

As I repeatedly wrote above, algorithms alone are not sufficient to satisfy customers. Customers want turn-key services that are secured, durable and available. Systems architecture knowledge is a common gap among ML practitioners, in my experience the main reason we do not see more ML in production. Joint knowledge of ML and computer systems is so rare that in 2019 69 prominent ML technical leaders authored a paper on this missing knowledge and proposed the creation of an *MLSys* conference dedicated to the intersection of machine learning and systems (*MLSys: the New Frontier of Machine Learning Systems*, Ratner et al). Among the authors were Alex Smola, VP of Science at AWS, Jeff Dean, VP of AI at Google and Ion Stoica, co-inventor of Spark and Ray. The transformation of algorithm code into fully functional systems is driven by the use-case. When facing an ML use-case, ask yourself the following 7 questions to draft the high-level system architecture:

1. **What are the system goals and their respective priorities?**
2. **Is there a constraint on system locality?**
3. **Should the model be exposed as an API?**
4. **What is the inference mode?**
5. **Is there one or multiple models?**
6. **What is the data flow for training and inference?**
7. **What are the re-training policies?**

What are the system goals and their respective priorities?

Considerations for low-latency deployments

In some use-cases such as advertising or robotic control, prediction speed is the utmost priority. Several architectural options should be considered. Model compilation is an inference acceleration technique that produces an optimized graph and an associated optimized execution application, adapted to the destination hardware. Compilation is further discussed in section IV.8. Orthogonal to compilation, one can also rethink hardware choices

and deploy to fast CPU, GPU or to application-specific integrated circuit (ASIC) such as AWS Inferentia. Field Programmable Gate Arrays (FPGA) are occasionally seen in ML research but rare in production. Optimizing model location also drives customer-facing system latency down. By deploying the model is the same location as its consuming application, one can reduce the query latency by several dozen millisecond, or even several hundred if removing a cross-planet HTTP call. Caching can also drive latency down by avoiding to re-compute predictions that were already served in the recent past. You can build caching solution using fast key-value stores like Amazon DynamoDB and Redis.

Considerations for cost-efficient deployments

The above-mentioned compilation and computation reduction are equally important for cost control. The faster the model runs, the less hardware it needs. Hardware acceleration may paradoxically be a good idea to reduce costs as well: a GPU-equipped instance may have a higher hourly price than a CPU instance but the superior parallelism ability of the GPU may make it tolerate a higher amount of concurrency at equivalent or lower costs. In the AWS cloud, AWS Lambda is a micro-virtual machine service that provides pay-per invocation ephemeral virtual machines with up to 10Gb RAM and 15min execution time. Deployments in AWS Lambda incur zero charges when receiving no traffic, in that regard their economics are unbeatable for low-traffic applications.

Considerations for high-concurrency deployments

If concurrency is the priority, it is possible to run multiple copies of the model on multiple machines and balance the traffic between them, a practice called *horizontal scaling*. Batching the requests is a good idea as well to minimize communication overhead. Modern ML servers such as TorchServe, Multi-Model Server (MMS), TensorFlow Serving and NVIDIA Triton expose controls to batch records on the server side. Section IV.8 dives deep in model serving.

Is there a constraint on system locality?

Will the system have connectivity to internet? To the cloud? Does the system run under a limited power and compute footprint? Being able to deploy the system in the cloud opens up numerous possibilities in terms of hardware, automation and security. Deployment on the edge has varying complexity, depending on the embedded hardware and software. For

example, devices compatible with AWS Greengrass software can run cloud-developed, locally-deployed AWS Lambda functions and ML models, and can feed input and output data back to the cloud via proprietary *ML Feedback Connectors*. Device manufacturers occasionally provide optimized ML runtimes simplifying embedded ML deployment experience. For example, Apple developed the CoreML stack for ML use the iOS ecosystem, Qualcomm devices feature Snapdragon Neural Processing Engine (SNPE) and NVIDIA features the TensorRT stack for quantization, compilation and ML inference.

Should the model be exposed as an API?

Does the end user need to access the model synchronously over a REST API? If yes, it means you have to choose and develop a web server to host your model. Flask is a popular option that is known to scale well when paired with Gunicorn for concurrency management. NVIDIA Triton, MMS and TorchServe are recent promising options that have dedicated ML features, such as server-side batching, multi-model and inference statistics. TensorFlow Serving is the official webserver for TensorFlow. In the AWS ecosystem, using AWS Lambda avoids using a web server, as long as your model fits within Lambda limits.

What is the inference mode?

Should the system return results in real-time, or are invocations realized in batches, scheduled or occasionally invoked? Batch prediction is cost-effective, since the compute expense exactly matches the compute needs. It is also simpler to develop than real-time inference, since it usually requires no serving stack and has less constraints on payload size and processing time. Note that the Amazon SageMaker Batch Transform API is an exception to that last rule: despite being a batch inference service it requires a serving stack, since its design leverages the same inference contract than the real-time SageMaker Hosting feature.

Is there one or multiple models?

An ML system may consist of multiple models called sequentially, in parallel or in an arbitrary directed acyclic graph (DAG). A fictional example of such a DAG-based ML system is represented in figure 12.

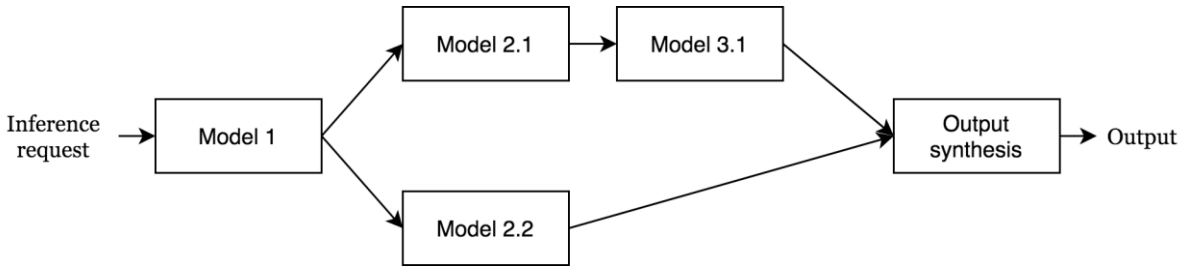

Figure 12. Example of fictional ML system DAG with 4 ML components

In such designs, having different code bases and infrastructure for each model has several benefits: (1) iteration, replacement, ablation studies can easily be conducted (2) different software and hardware technologies can be used for each ML component, (3) correlated failures are less likely thereby increasing system robustness. However, splitting the system into independent services comes with additional communication needs: inbound requests need to be routed to each component and the final result assembled. In an asynchronous, batch inference context, standard workflow managers such as AWS Step Functions and Apache Airflow can be used to orchestrate a DAG of batch transformations. In a real-time, synchronous inference context, each component can live in its own synchronous web service endpoint. The DAG and output management can be managed from a central orchestrator, that can either be ad-hoc code in a separate server or a DAG creator such as AWS Step Functions or AWS Express Step Function, the latter being adapted to high-concurrency, high-volume situations. The open-source asynchronous distributed computing Ray is a promising platform for multi-model inference, given its use of a distributed shared memory (Apache Arrow's Plasma Store), likely to accelerate multi-model communication. It is no surprise that the ML server Ray Serve is being developed on top of Ray, and that it supports hosting DAGs of models, named model composition in Ray Serve.

What is the data flow for training and inference?

The flow of training and inference data drives ML system architecture. Two orthogonal system constraints influence the data flow.

Feature accessibility: batch or key-value?

Data access techniques vary depending on the stage of the model lifecycle. For training and batch inference, data is processed in bulk over groups of records. For such use-cases, when

data is structured, traditional analytics and ETL tools (Extract, Transform, Load) easily assemble features into training datasets and inference datasets. On the other hand, in the real-time inference setting the model needs the features of one record only, but needs it fast. In order to satisfy this different access pattern, one needs fast databases that excels at lookup tasks: this is the realm of modern, NoSQL key-value stores like Amazon DynamoDB and Redis. This component is called an *online feature store* and can be seen from time to time in the publications of mature ML organizations. Dedicated feature store offerings emerged in the past few years, such as Tecton, Hopsworks Feature Store or the Amazon SageMaker Feature Store.

Feature freshness: how often is the feature updated?

Let's consider a fictional music recommendation system that predicts which genre to recommend to a user given its past activity and its metadata, such as country and subscription type. Country and subscription type are slow-changing features: they are not likely to change often, and latest values could be extracted regularly from operational databases to the feature store and from there to ML inference systems. On the other hand, features based on past activity are dynamic. As the user interacts with the music service, what the model should view as its "past activity" is permanently evolving. For example, if the recommendation model uses as a feature the "number of minutes of rock listened in the past 30min", it means it must be continuously computing this feature in order to make relevant predictions. Such a design, involving the near-continuous update of features, is supported by streaming computation engines such as Apache Kafka or Amazon Kinesis.

	Batch inference	Real-time inference
Slow-changing features	Features computed and served with batch ETL into batch feature store	Features computed with batch processing ETL and served from a real-time feature store
Fast-changing features	Features computed with streaming ETL and served with batch ETL into batch feature store	Features computed with streaming ETL and served from a real-time feature store

Figure 13. How to store and serve features based on use-case characteristics

What are the re-training policies?

The design of a model re-training policy is use-case specific. Answering the following questions will help you design an adequate model re-training scheme.

Did the training data change significantly?

Such changes can be detected programmatically, by running automated quality checks over the data and raising alerts if anomalies happen. For example, if feature extrema change significantly, or if new categorical features occur in the input data, it is worth considering a retraining. Anticipate change when you can. For example, is a data source going to be discontinued? Is your data provider going to change revenue reporting from euro to dollar? Data sources are your suppliers and you need to monitor their quality.

Can the model make predictions on new data?

Most collaborative filtering recommender systems, including the popular matrix factorization design described in Appendix A1, learn representations for each customer and product. In that regard, they cannot make predictions for customers and products unseen in the training data, a challenge named the *cold start* problem. Collaborative filtering recommenders need to be trained frequently in businesses with frequent new customers or new products. In practice, it is common to see daily training.

Is there a benefit to train on more data?

Model accuracy usually grows with data size. Yet, is the value brought by bigger-data training higher than the cumulated extra costs caused by data acquisition, model training and deployment?

Is there a benefit to train on more recent data?

Some machine learning systems, for example recommendation for live content, require data as recent as possible. In other domains, the incremental value of fresher data is more tenuous. For example, in machine translation, you do not necessarily get better models with a dataset a week more recent. Chances are that languages did not change much over the course of a week. When a system consists of a DAG of models, its different model components can be trained at difference cadences, even by different teams. This is one advantage of distributing

an ML DAG in a DAG of independent micro-services. This can also be achieved on neural networks, and is a reason for their broad adoption: model graphs can have parts frozen during training and trained at difference cadences. For example, in a recommender system, product representations could be trained offline with a first model at a low frequency, and user-item interaction weights be trained by another model at a higher frequency. Other example: in an organization that owns a lot of image or text data, training synergies can be achieved by pre-training large base models at a given frequency, and having each team fine-tune their own model on top of the organization-wide backbone at a different frequency.

Answering the 6 questions above will help you identify application-specific high-level design constraints. Then, to specify the full system architecture, I recommend to make sure software components and mechanisms are in place to cover the best practices summarized in the table on figure 14.

Domain	**Best practice**
Operations	Success metrics and key operational metrics are displayed in a dashboard and deviations detected with anomaly detection systems
Operations	Training and inference infrastructure is monitored
Operations	Training and inference model performance is monitored
Operations	Training and inference feature summaries and extrema are monitored
Operations	Training experiments are persisted so that past trainings can be reproduced (with a relevant retention window)
Operations	Processes are documented in runbooks and have owners
Operations	Operational anomalies trigger alerts
Operations	Operational failures are researched and documented
Operations	Manual interventions to the production system are documented
Operations	Improvement requests are documented and prioritized
Operations	Bugs and anomalies are documented and prioritized
Operations	There are at least 3 environments for development, test and prod
Operations	There is a mechanism to convert source code into a product
Operations	There is automated testing launched at each code integration
Operations	Compliance standards are enforced via automated rules as much as possible
Operations	There is a decommissioning program to detect and manage unused resources
Operations	Changes can be rolled back
Security	Least privilege rules are used for API access, resource access and data access
Security	Interactions with data, compute resources, development, test and production are logged
Security	There is a data handling standard associating handling guidance to data classes

Security	An incident response process is in place
Security	Credentials, license keys and other secrets are not written in code
Security	Developer accounts are secured with MFA
Security	Detective controls raise alarms on anomalous API use, anomalous data move or anomalous system interaction
Security	A threat model has been created, discussed, mitigations listed and applied
Security	Network access is protected
Security	Resource access is protected
Security	Developers have an internal point of contact for security guidance
Security	Developers have an internal point of contact for legal, compliance and regulatory guidance
Security	There is a process to validate new tools, vendors, partners and software packages
Reliability	There is a mechanism to regularly test rare modes including recovery procedures
Reliability	The system can sustain an order of magnitude increase in load without human intervention
Reliability	The service limits of your suppliers are known and accepted. This includes SLAs and usage quotas granted by your service providers

Figure 14. Best practices for production ML. Note that very few of them are specific to ML.

Many of those best practices are native or easy to implement in the AWS cloud. For example, Amazon SageMaker provides default metadata persistence for training tasks and endpoints, monitoring features, and numerous infrastructure options. Similarly, automated testing and deployments can be conducted with services such as AWS Lambda, AWS CodePipeline and AWS CloudFormation. The AWS cloud shines with respects to security and provides a number of controls such as Virtual Private Cloud (VPC) and Identity and Access Management (IAM) to facilitate best practices enforcement.

When choosing an ML cloud infrastructure vendor, you should look further than hourly machine price. Offerings across cloud providers are rarely comparable and different vendors should not be considered as different sellers of the same commodity. Security, availability, product quality, partner ecosystem, peripheral services and integrations vary across clouds and impact project feasibility, pace, quality and costs.

IV.2. Cloud Computing

According to the cloud provider AWS, *"cloud computing is the on-demand delivery of IT resources over the Internet with pay-as-you-go pricing. Instead of buying, owning, and maintaining physical data centers and servers, you can access technology services, such as computing power, storage, and databases, on an as-needed basis from a cloud provider"*. Organizations of all types including non-profits, governments, startups and large enterprises, broadly adopted the cloud to reduce their costs, gain in agility, improve their security posture and provide new customer experiences. As disclosed in the introduction, in this book I predominantly refer to AWS when discussing about cloud concepts. AWS has outstanding support and adoption for machine learning. According to an estimation from Nucleus Research, 89% of cloud deep learning workloads run on AWS.

Why does it matter for the ML practitioner?
Cloud computing knowledge helps the ML practitioner improve its project economics, scalability, and time-to-market.

Benefits of the cloud for the ML practitioner

Good economics

The pay-per-use consumption model of the cloud drives savings compared to the physical infrastructure world. For example, a physical NVIDIA Tesla V100 GPU costs several thousand dollars, excluding server host, installation and facility management. Using a V100-equipped virtual machine on AWS for a 10min-training job would cost around $0.7! And using discount plans such as the Amazon Spot capacity could further reduce that by up to 90%.

Good scalability

The cloud enables usage of computing resources that would be long and difficult to procure in an on-premise world. For example, AWS realized a BERT TensorFlow training on 256 EC2 P3dn.24xlarge instances, cumulating 2,048 NVIDIA V100 GPUs. Amazon EC2 virtual machines have often been orchestrated into large-scale jobs cumulating more than a million

vCPUs, for example by Western Digital (*Western Digital HDD Simulation at Cloud Scale – 2.5 Million HPC Tasks, 40K EC2 Spot Instances*, by Jeff Barr, AWS News Blog) and Clemson University (*Natural Language Processing at Clemson University – 1.1 Million vCPUs & EC2 Spot Instances*, by Jeff Barr, AWS News Blog).

Easier research management

In the cloud, scientific experiments are easier to conduct because of (i) the broad choice of infrastructure, (ii) the easy configuration of resources, (iii) the experiment management ability. For example, Amazon SageMaker includes a managed, serverless metadata store that persists ML training metadata such as hyperparameters, data paths, code images, along with model performance.

Faster application development

In the cloud, end-to-end systems are easier to develop. Top cloud providers provide low-learning curve, managed services for many popular application blocks such as code repositories, relational databases, key-value stores, big data processing engines, data visualization, workflow orchestrators, API managers. Such services make it easier for ML developers to transform their ML creations into end-to-end systems.

Benefits of the AWS cloud for the ML practitioner

Broadest set of features for machine learning

AWS has broad support for custom machine learning workloads, covering annotation, development, training, tuning, features management, CICD, batch and real-time inference, compilation, monitoring and bias management. The SageMaker platform specifically shines, with 18 built-in algorithms and dedicated integrations for open-source frameworks Scikit-Learn, TensorFlow, MXNet, PyTorch, XGBoost, PySpark and Hugging Face. Its flexible container-based interface allows developers to bring their own code, and I saw successful use of numerous other frameworks in SageMaker, including Caffe2, R, LightGBM, PaddlePaddle, Detectron2, NVIDIA RAPIDS and Vowpal Wabbit.

Furthermore, as of 2021, I believe that a number of high-value ML primitives can be found only on AWS. For example, I am not aware of managed, production-grade alternatives to the model tensor inspection toolkit SageMaker Debugger, to the managed SageMaker model

parallelism library, the inference-optimized ASIC AWS Inferentia, the managed, multi-model, multi-container, multi-variant, monitored SageMaker Hosting service or the fractioned GPU Amazon Elastic Inference.

High-bar for security and availability

Data availability and protection are key concerns for ML practitioners, and AWS shines in both. Developers can finely control encryption, permissions, network isolation, audit logging and automated remediations with the numerous security and automation features of the AWS cloud. Some of the most demanding organizations in the world, from some of the most regulated sectors, run on AWS.

I highlight in Appendix A4 of this book notable AWS services useful for production machine learning. Antje Barth and Chris Fregly give a wide and deep presentation of AWS services for data science in their excellent book *Data Science on AWS: Implementing End-to-End, Continuous AI and Machine Learning Pipelines*. *Learn Amazon SageMaker* from Julien Simon focuses on managed ML platform Amazon SageMaker and also gathered excellent feedback.

IV.3 CICD

Continuous Integration & Continuous Delivery (CICD) is a software development practice that consists of automating the application building, testing and deployment process so that code releases are more frequent, bugs caught before customer exposure, and developers more productive. In *Going Faster with Continous Delivery*, Amazon Software Development Manager Mark Mansour indicates that the team that piloted continuous deployment reduced by 90% its lead time from check-in to production.

Why does it matter for ML practitioners?

Thanks to automation, CICD improves both output quality and delivery speed. CICD is a critical skill that makes ML practitioners impactful. ML practitioner without CICD knowledge

need either to spend time learning it, or need to be complemented with extra team members knowing it.

A CICD *pipeline* is a semi-automated workflow transforming code in a customer-ready deliverable. It consists of the steps below.

Code versioning

Developers write code locally on local or remote workstations and occasionally commit it to a remote, central code repository. Contributions from various individuals can be reviewed and merged.

Build

The *build* step transforms source code in artifacts necessary for final software product instantiation. It is a flexible term, that could refer to compiling Java or C++ into executable code, or building a Docker image for an application or ML model. The build can also include early-stage testing, such as static code analysis and unit testing.

ML practitioners are occasionally confused when applying CICD to ML projects: should model training be in the application or in its build? As usual in IT architecture, it depends on the use-case. Most of the ML CICD literature I read run model training within a CICD build step. This is for example the case in the *AWS Well-Architected, ML Lens* whitepaper, in the SageMaker Project MLOps default template and in the blog *What did I Learn about CI/CD for Machine Learning* (Ari Bajo) from the MLOps platform Valohai. However, ML systems that retrain model frequently or autonomously would instead contain the training management code as part of the system code, and in that regard the build phase would not build the models, but instead build the system that will build the models once in production. The Google Cloud MLOps CICD documentation documents this alternate ML CICD pattern. In order to settle this choice, ML practitioners should work backward from the contract they establish with their customers. If customers expect a product with slow-changing, static ML artifacts – in other words if ML artifacts do not change within a product version – it makes sense to train the model as part of a CICD build phase. This is the case 1 in figure 15, and represents the majority of ML CICD literature so far. If the model should update itself within a given product version with no developer intervention, then the training should rather be part of the application itself (case 2 in figure 15).

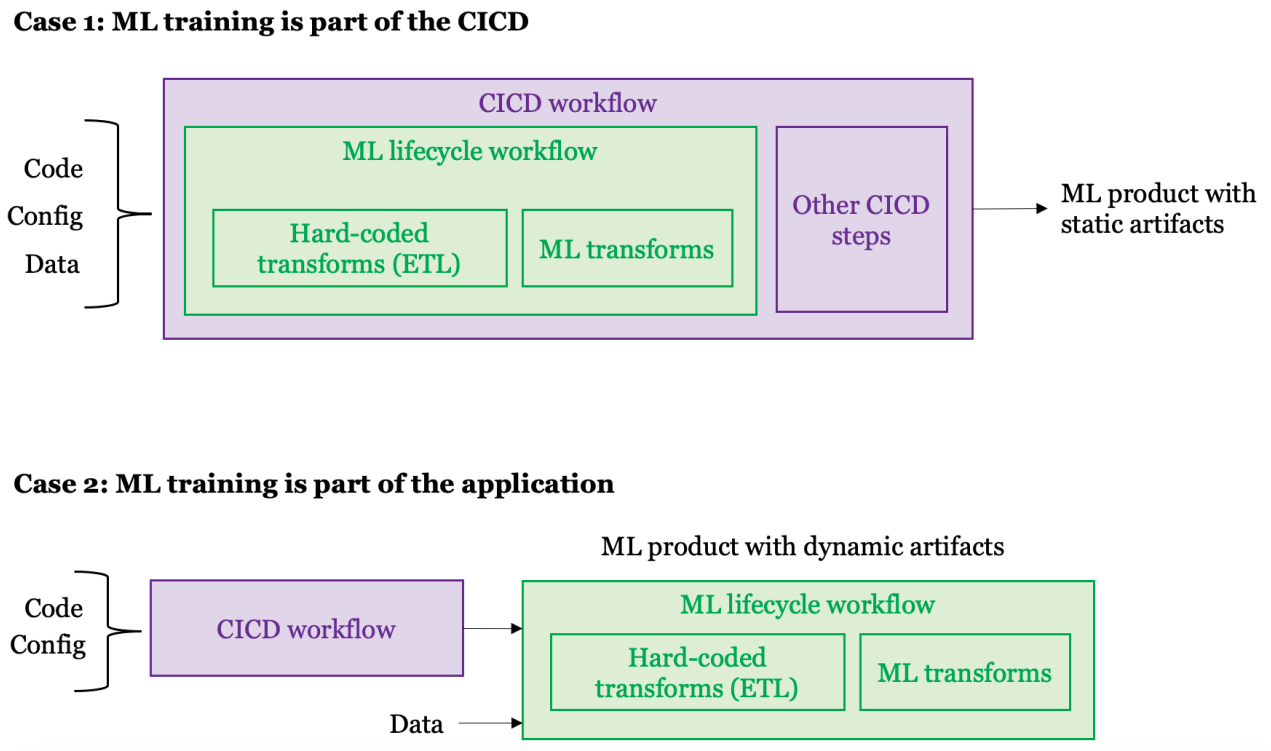

Figure 15. Is ML training part of the CICD or part of the system? It depends on the nature of the product your delivering

Test

The test phase overlaps with the build. It consists of both small, atomic test on isolated functionality (*unit tests*) and broader, end-to-end test of the system. *Integration tests* check how the proposed change behaves when connected to other system components, *load test* checks how the proposed change behaves under traffic.

Deploy

Once an application build is fit for customer exposure, it is deployed in front of live traffic. Phased deployment limits the impact of bugs that have not been caught in previous phases, with techniques such as:

Shadow deployment

Inbound traffic is sent to both version 1 and version 2. Responses from version 1 are sent back to clients and responses from version 2 are analyzed.

Blue-green deployment

Inbound traffic is split between version 1 and version 2, with an initially small amount of request reaching version 2. The traffic share sent to version 2 is progressively shifted from 0% to 100% based on success metrics.

Further reading

To learn more about CICD, consider reading the following documents:

- *Going faster with continuous delivery*, Mark Mansour
- *Automating safe, hands-off deployments*, Clare Liguori
- *Why should you use MLOps?* Amazon SageMaker Documentation

IV.4 Baseline GPU knowledge

Why does it matter for the ML practitioner?

The graphics processing unit (GPU) is a popular computing device for deep learning training and inference. GPU expertise is a valuable skill to have because it keeps project cost under control and reduces time-to-market.

Born in the late 1990s to compute geometrical transformations at scale for graphics workloads, GPUs were later found convenient for machine learning computations. GPUs are commonly misunderstood by ML practitioners. Below are couple things ML practitioners should know about GPUs.

In 2021, most ML GPUs are NVIDIA GPUs

NVIDIA leads the ML GPU market and most non-mobile ML GPU workloads run on NVIDIA GPUs. Excellent at hardware innovation, NVIDIA also maintains well integrated software development kits (SDK) such as the neural network development library cuDNN and the TensorRT inference optimization software suite. NVIDIA SDKs allow full exploitation of NVIDIA hardware and developers enjoy them. NVIDIA is one of the drivers behind the

decade of growth experienced by deep learning. Illustrating the prevalence of NVIDIA hardware, the GPU computation mode in the DL framework PyTorch is NVIDIA-branded: it is invoked with a method named `.cuda()`, which is the name of NVIDIA's GPU programming library. In this book, unless mentioned otherwise, the GPU acronym refers to an NVIDIA GPU

Not all GPUs are equal

There are performance differences between different generation of GPUs. For example, AlexNet training time with the Caffe framework was divided by 2 when ported from NVIDIA K80 to NVIDIA P100 (source), and ResNet training time on MXNet was divided by 3 when switching from NVIDIA P100 to NVIDIA V100. (source). Modern generations of NVIDIA GPU cards such as the Volta V100 and the Turing T4 excel at machine learning acceleration and feature specialize cells, the NVIDIA *Tensor Cores*, that accelerate the matrix multiply-accumulate operation ($AX+B$) commonly seen in deep learning models. Another differentiator of recent NVIDIA cards in the presence of the NVLink interconnect, that enables high-bandwidth direct GPU-to-GPU communication. NVLink contributed to improve the state-of-the-art in multi-device DL training when paired with peer-to-peer gradient accumulation techniques such as the Horovod ring-reduction developed by Uber engineers Alex Sergeev and Mike Del Balso. GPU generally drives the cost of an ML project so it is recommended to benchmark different GPU cards.

Card	CUDA cores	Tensor Cores	Memory (bandwidth)	Single TFLOP	Source
K40	2,880	no	12G (288Gb/s)	4.3 TFLOPS	Data sheet
P100	3,584	no	16G (732Gb/s)	10.6 TFLOPS	Data sheet
V100 (SXM2)	5,120	640	16G or 32G (900Gb/s)	15.7 TFLOPS	Data sheet
T4	2,560	320	16G (300Gb/s)	8.1 TFLOPS	Data sheet
A100	6,912	432 (gen. 3)	40G (1,555Gb/s)	19.5 TFLOPS	Whitepaper

Figure 16. High-level specifics of latest generations of GPUs. CUDA (Compute Unified Device Architecture) cores are individual NVIDIA proprietary arithmetic logic units. A GPU has 100x to 1000x more compute units than standard CPUs, though GPU CUDA core

frequency is about half the frequency of modern Intel CPUs, giving a sense of their bigger processing capacity when fully utilized.

Not all ML frameworks use GPU equally well

Not all frameworks support GPU. Use a GPU-equipped infrastructure only if you are absolutely certain that your ML framework will use it. For example, this is not the case of Scikit-Learn. Furthermore, even if an ML framework runs on GPU, it may not necessarily use GPU memory, CUDA compute cores and Tensor Cores in an efficient way. You should benchmark different frameworks and settings to maximize the GPU compute efficiency. If using a GPU equipped with Tensor Cores such as the V100 or the T4, you should try to use mixed precision training to fully leverage the power of the Tensor Core cell, a trick that can accelerate training by up to a factor 3x according to NVIDIA. In NVIDIA reference training benchmarks, 97 out of 100 benchmarks train in mixed precision. (PyTorch Mask-RCNN, PyTorch Tacotron2 and TensorFlow VAE-CF are benchmarked in TF32 precision)

It is difficult to share a GPU between concurrent applications

When two processes simultaneously launch a computation on the GPU, two *CUDA contexts* are created and their computations done one after the other. Individual GPU computations are called *kernels*, and the NVIDIA documentation states that kernels launched from different contexts cannot be executed concurrently. To bypass this limitation and run concurrent tasks on the same GPUs, at least 4 options are possible:

Batching

If running the same task on multiple data, for example applying the same ML model to multiple records, one should consider concatenating the multiple inputs into a single payload and execute it as one GPU task, a technique called batching.

CUDA Streams

CUDA Streams are queues of instructions executing on an NVIDIA GPU. Executions requests sent to a CUDA stream are run sequentially, first-in first-out. Multiple CUDA streams can run at the same time, allowing the overlapping execution of instructions. It is a low-level concept that ML practitioners rarely directly work with. The NVIDIA Triton

server uses CUDA streams to serve multiple models from a single GPU. Thanks to this ability, ML practitioners with multi-model serving workloads adore it.

NVIDIA Multi-Process Service (MPS)

On recent GPUs it is possible to logically partition the card with NVIDIA Multi Process Service (MPS) so that multiple processes concurrently use the GPU. MPS aggregates computation requests coming from different processes into a single CUDA context, bypassing the inability to run multiple concurrent contexts on the same card. See NVIDIA documentation for exact requirements for MPS use.

NVIDIA Multi-Instance GPU (MIG)

The latest-generation Ampere A100 GPU card has a physical partitioning feature named the Multi-Instance GPU (MIG) that enables more robust program multitenancy that the other 3 options.

Instead of managing GPU partitioning yourself, a production-grade service can do it on your behalf: Amazon Elastic Inference is a commercial infrastructure service created by AWS that enables you to attach granular amounts of GPU accelerator to CPU instances, enabling to right-size accelerator infrastructure for a given workload.

Data loading is a common bottleneck causing low GPU usage

Modern NVIDIA GPUs have a massive computational ability paired with important memory bandwidth. Keeping them busy - "feeding the beast" - is not trivial and requires a well-thought data loading pipeline. Data should come from a low-latency, high-throughput storage and the host should have enough CPU to read and move data. GPU starvation caused by inefficient data loading is reported by Airbnb researchers in their paper *Applying Deep Learning to Airbnb Search* (Haldar et al), by AWS software engineer Sina Afrooze in the blog post *Maximize Training Performance with Gluon Data Loaders* and by Mobileye (an Intel company) ML developer Chaim Rand in *Overcoming Data Preprocessing Bottlenecks with TensorFlow Data Service, NVIDIA DALI, and Other Methods*. In order to alleviate this bottleneck, tune CPU-GPU balance and if both CPU usage and GPU usage are low, allocate more CPU budget to data loading by increasing worker count in the data loader. If CPU usage is high and GPU usage low, reducing the pre-processing work by doing it offline, on a machine with more CPUs or on a different machine may lift CPU pressure and increase GPU use. To

further optimize data transport and maximize GPU use, NVIDIA created the Data Loading Library (DALI), a collection of optimizations for fast data loading and pre-processing.

Training on multiple GPUs usually require writing ad-hoc distribution code

Train on multiple GPUs only if you are certain that you wrote code than supports distributed training. If not all algorithms and framework support GPU training, it is also the case that not all frameworks and algorithms support distributed GPU training. DL frameworks PyTorch, MXNet and TensorFlow offer ad-hoc functions to run on multiple GPUs. Single-host, multi-device scripts are easy to write using data-parallel functions, for example `DistributedDataParallel` in PyTorch, the `tf.distribute.MirroredStrategy()` in TensorFlow or convenient but now-deprecated `multi_gpu_model` in Keras. In Apache MXNet, developers control locality of model parameters and data with the `mxnet.context` object and it is easy to program data-parallel training by duplicating a model on an array of *n* GPU contexts and sending it fractioned data with the `split and load` function. Training on multiple machines is more complex as code runs on multiple machine who need to synchronize their progress. Uber researchers Alex Sergeev and Mike Del Balso simplified it by developing the Horovod distributed SGD library, available for TensorFlow, PyTorch and MXNet. Hugging Face Transformers, the most popular open-source NLP library, is an exception to that caveat on multi-GPU training. In Transformers, ML practitioners can use a `Trainer` class that trains equally well on one or multiple GPUs, but also on multiple machines if used on Amazon SageMaker. This compact, pleasant experience is illustrated by Hugging Face engineer Philip Schmid in *Distributed Training: Train BART/T5 for Summarization using Transformers and Amazon SageMaker*. Managed Effortless GPU distribution experience is the exception more than the norm. With neural network development frameworks other than Hugging Face Transformers, developers generally have to alter their script to make them work on multiple devices. Part IV.5 of the book dives deep in neural network training.

Do you really need GPU for ML inference?

It is feasible to fully utilize a GPU during model training, because (1) developers control the schedule of data and compute tasks sent to the GPU and (2) the compute load is intense. For each training record, the model runs (i) a forward pass, (ii) the error, (iii) a backward pass, and (iv) model weights are updated. For inference however, there are at least 4 reasons you should consider testing on CPU before deploying on GPU:

Smaller parallelism opportunity

The parallelism opportunity at inference often is several orders of magnitude smaller than at training: your application traffic may not generate the same level of concurrency as a batched, well-optimized neural network training, and inference only consists of forward passes. The low GPU usage on GPU-based model serving endpoints often surprises ML practitioners.

CPU have higher clock frequencies

GPU cores generally have lower clock frequencies than CPU cores. For example, an NVIDIA Tesla V100 has an operating clock frequency around 1.3GHz while an iPhone A9 core runs at 1.9GHz and the Intel CPUs of an AWS EC2 C5 instance run at 3GHz. For small models, GPU may not accelerate inference that much. I wrote an elementary benchmark for the MXNet blog, in which an MXNet ResNet-18 runs single-image inference 57% faster on GPU than on CPU (33ms vs 76ms), a meager improvement that comes at a +760% price increase.

CPU relevancy for high-performance algebra keeps increasing

Modern Intel cores feature AVX512 instructions, a *Single Instruction, Multiple Data* (SIMD) parallel computing feature that enhances CPU relevancy for ML workloads. Arm-based AWS-developed Graviton2 processors are also promising for deep learning inference. Able of fp16 and int8 dot-product, the Graviton2-equipped EC2 M6g instance runs BERT NLP inference 28% faster than the Intel-equipped EC2 M5 instance while being 20% cheaper (*AWS re:Invent 2020: Deep dive on AWS Graviton2 processor-powered EC2 instances*)

Compilation is coming

Compilation stacks such as Tensor Virtual Machine (*TVM: An Automated End-to-End Optimizing Compiler for Deep Learning*, Chen et al) keep reducing the latency of CPU-based deep learning inference. In January 2020, Microsoft open-sourced an optimization suite for BERT models that reduce CPU latency by a factor of 17. A unit-batch inference of their optimized ONNX model ran in 9ms on an Azure Standard F16s_v2 CPU Virtual Machine (VM). AWS engineers authored NeoCPU, which runs SSD-ResNet-50 detection

models in less than 100ms on both 18-core Intel Skylake CPU and 24-core AMD Epyc CPU (*Optimizing CNN Model Inference on CPUs*, Liu et al).

Non-NVIDIA DL chips are coming

Many companies are working on DL chips. Couple projects are mature:

AWS Inferentia

The Amazon EC2 Inf1 instance contains 1 to 16 Inferentia chip(s), each made of 4 systolic array-based matrix-multiply *NeuronCores*. The systolic array concept consists of a matrix of inter-connected compute cells, that realizes matrix product by accumulating results cell by cell like a wave front, hence its name. Systolic arrays achieve higher computational throughput than GPUs at the expense of flexibility. According to AWS, Inferentia instances provide the lowest cost-per-inference infrastructure in the cloud. Inferentia shines with its software stack: compatible with TensorFlow, PyTorch and MXNet, its Neuron SDK compiler can apply auto-casting to bfloat16 precision and can partition models across devices to run pipelined, model-parallel inference. several public benchmarks measured double-digit percentage reduction in cost-per-inference compared to the already well-performing NVIDIA T4-equipped G4 instance. Cost-per-inference decreased by 40.7% on a TensorFlow BERT example (Reivent 2019), by 72% on TensorFlow OpenPose (*Deploying TensorFlow OpenPose on AWS Inferentia-based Inf1 instances for significant price performance improvements*, Nonato de Paula et Li, AWS ML Blog) and by 37% on TensorFlow YoloV4 (*Achieving 1.85x higher performance for deep learning based object detection with an AWS Neuron compiled YOLOv4 model on AWS Inferentia*, Nonato de Paula et al). In 2021, Inferentia obtained the lowest BERT cost-per-inference I ever saw: $0.1 per 1MM inferences. (*Achieve 12x higher throughput and lowest latency for PyTorch Natural Language Processing applications out-of-the-box on AWS Inferentia*, Nonato de Paula et al.). Besides, at AWS re:Invent 2020, AWS disclosed that Amazon Alexa migrated its Text-to-speech inference from EC2 P3 instances, equipped with NVIDIA V100 GPU, to Inferentia-equipped Inf1 instances, a move that reduced costs by 30% and latency by 25%. (*AWS re:Invent 2020: The journey to silicon innovation at AWS*).

Google TPU

Google started developing the Tensor Processing Unit (TPU) in 2013, and shipped TPU v1 in 2015. TPUs are said to power real-time Google deep learning inference systems such as Google Search, Google Translate and the search feature of Google Photos. TPU v1 was an inference-only device, and the 2017-launched TPU v2 targets both inference and training workloads. Marketed as a GCP cloud service, the Cloud TPU v2 provides 64Gb memory and 180 TFLOPs per instance. GCP also features *TPU Pods*, clusters of Cloud TPUs interconnected in a network optimized for distributed aggregations, also called *allreduce* operations. Currently in its third generation (TPU v3, 128 Gb memory and 420 TFLOPs), the TPU is based on the matrix multiplication systolic array.

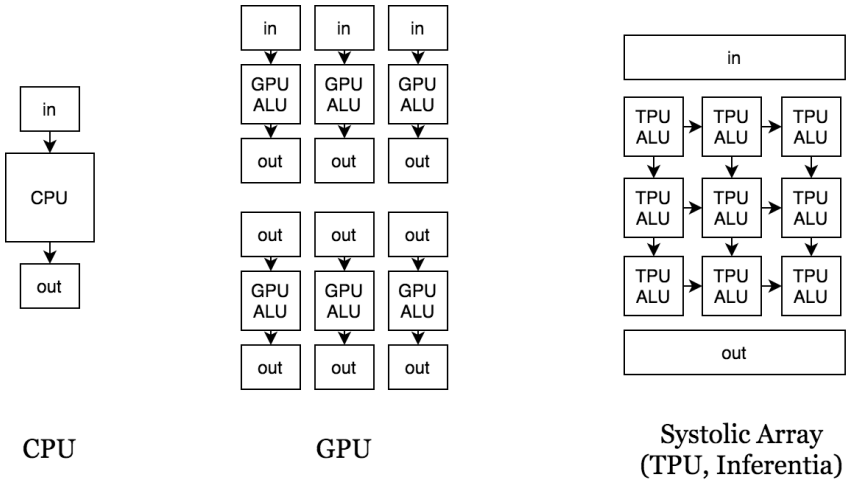

Figure 17. CPU, GPU, Systolic array

Graphcore Intelligence Processing Unit (IPU)

The Graphcore IPU chip has a novel *Multiple Instruction, Multiple Data* (MIMD) design with 1,216 tiles each containing a computing core and 256kiB of low-latency static random-access memory (SRAM). The *IPU Exchange* communication fabric connects the tiles together. A proprietary *IPU-Link* interconnect proposes 320Gb/s of chip-to-chip bandwidth. A single IPU chip has a peak compute performance of 31+ TFLOPS, twice more than an NVIDIA Tesla V100. An April 2020 benchmark realized by Graphcore mentions higher throughput than GPU at equal thermal design power (TDP) on a variety of tasks, yet it is difficult to see how fair the comparison is since no customer-facing costs are displayed for both benchmarked options.

Habana Gaudi (Intel)

The Habana Gaudi gained popularity on December 2020 with AWS announcing its intention to offer new Gaudi-equipped Amazon EC2 virtual machines appropriate for ML training. Gaudi chips are developed by Habana Labs, itself acquired by Intel in 2019. According to AWS, Habana Gaudi will enable up to 40% better training price-performance than current GPU-based EC2 instances. In order to train a model on a Habana Gaudi card, developers use Habana's SynapseAI SDK to port code written with deep learning frameworks TensorFlow and PyTorch. The Gaudi card comprises a number of programmable *Tensor Processing Cores* (TPC) which implement the SIMD (single instruction, multiple data) parallelism paradigm. The Gaudi shines by its inter-chip communication capacities, which allows multi-chip workload scaling both within a server, within a server rack and across racks. According to a 2019 Habana Labs Gaudi Whitepaper, a Gaudi-based cluster of 640 chips delivers 3.8x better ResNet50 training throughput than a 640-NVIDIA V100 cluster (both cards being in comparable range of max power consumption) (*Gaudi Training Platform White Paper*)

Huawei Ascend 910

The Ascend 910 is an AI-dedicated chip featuring more than twice the half-precision compute power of an NVIDIA V100 or a Google TPUv3. It demonstrated its strength by powering one of the fastest ImageNet ResNet 50 trainings, in 60s on the Huawei Atlas 900 supercomputing cluster. Huawei demonstrates leadership in AI by developing a full-stack ecosystem comprising both hardware and software: in addition to Ascend chips, Huawei also developed the open-source DL framework MindSpore.

IV.5 Neural network training efficiency and scalability

Why does it matter for the ML practitioner?

Neural networks are popular for tasks on low-level, raw data such as vision, language, speech and multi-dimensional timeseries. Nowadays, most ML practitioners are familiar with neural network design. However, training distribution and optimization is less known. This topic is however of paramount importance: cost-efficiency of ML teams is a concern for technical leaders, as some large models such as pixel-level vision networks or NLP transformers train only on expensive, multi-machine systems. In this section, I explain high-level fundamentals of training efficiency and distribution, and associated best practices.

A layman's introduction to neural network training

Neural networks learn their parameters by running iterative, tiny coefficients changes in the direction of steepest decrease of the error surface slope. This conceptually simple method carries the sophisticated name of Stochastic Gradient Descent (SGD). Neural network training is generally done by one or multiple GPU card(s). As previously described, GPUs accelerate algebra computations and are an important driver of the advent of deep learning. Several constants and design choices configure the training task; they are called hyperparameters. ML practitioners chose hyperparameters by intuition, brute force search or intelligent search. Several hyperparameters govern the performance of the SGD. In its simplest form, SGD consists of two hyperparameters that should be well understood: the learning rate and the mini-batch size.

$$Coef_{n+1} = Coef_n - learning\ \ rate \times average\ gradient_{\ on\ mini\text{-}batch}$$

Equation 1. Minibatch stochastic gradient descent. Neural network training consists of thousands to millions of such iterations. In practice, the SGD equation often contains other terms to stabilize and accelerate convergence, for example momentum and regularization.

The learning rate

The *learning rate* is the rate of coefficient change and can be the same rate for all model coefficients, for example in standard SGD, or a per-layer learning rate such as the *Layer-Wise Adaptive Rate Scaling* optimizer from You et al. The learning rate can also be different for

each coefficient, for example in the *Adam* optimizer from Kingma et Ba. The learning rate controls how fast a network learns. Analogous to humans that learn at a varying rate in their lifetime – at a fast pace as a child and slower and slower as they grow and exploit their knowledge – neural networks also benefit from a varying learning rate. The science - or shall I say the art - of in-training learning rate adjustment is called *learning rate scheduling*. An example of popular learning rate schedule is the learning rate *linear warmup with cosine decay*, cherished by the GluonCV community and used in three state-of-the-art computer vision papers: *Bag of Tricks for Image Classification with Convolutional Neural Networks* (He et al), *Bag of Freebies for Training Object Detection Neural Networks* (Zhang et al), *ResNeSt: Split-Attention Networks* (Zhang et al). Its rationale is intuitive: at the beginning of training, small learning rates help the network figure out which cape to take to decrease its error. This is the *warmup* phase. As a direction is found, learning rate quickly increases so that the model makes fast progress through the error surface in search of worthwhile error-minimizing areas. As iterations pass by, the learning rate decays progressively in a cosine curve. This reduction of the grain of model update enables finer and finer digging in the error landscape

```
import mxnet
from mxnet.lr_scheduler import CosineScheduler
import pandas as pd

lr = CosineScheduler(
    warmup_begin_lr=0.01, # initial learning rate
    warmup_steps=10,
    base_lr=0.1,  # peak learning rate
    final_lr=0.01,  # final learning rate
    max_update=80)  # end of decay. LR is constant after

pd.DataFrame([lr(epoch) for epoch in range(100)], columns=['Learning Rate']).plot()
```

```
:matplotlib.axes._subplots.AxesSubplot at 0x7f1b3cf8e550>
```

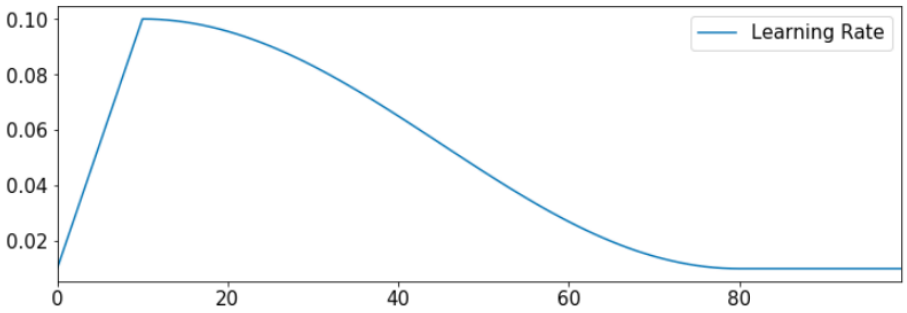

Figure 18. Plot of a learning rate linear warmup with cosine decay schedule over a training of 100 epochs. Common schedules are available out of the box in advanced DL frameworks such as Apache MXNet

The learning rate plays a critical role in SGD training: if it is too small, the model does not change its weights bluntly enough – it is too shy - and could take a long time to train or get locked in a small local error minimum. On the other hand, if the learning rate is too big, the model bumps from slope to slope or from valley to valley and never stabilizes in an area of low error: an overly large learning rate makes the model unable to capitalize on previous learnings.

The mini-batch

The error slope direction used for weight updates is named the *gradient*. It is not the exact error measured over all dataset, but instead a noisy, approximative error measured in a random sample of records called the *mini-batch*. Mini-batches are iteratively sampled without replacement from the dataset, error computed, gradient measured and weights updated, until no data is left. One such pass over the dataset is called an *epoch*. There is no rule governing the number of epoch necessary to train a model. Training jobs can consist of one to hundreds of epochs. The mini-batch plays a critical role in SGD training: it controls the amount of noise in the error gradient. If the mini-batch is too small, the gradient is too noisy – not representative enough of the data – and the model will update its weights in directions that do not make sense. If the mini-batch is too big, the gradient is not noisy enough – too accurate – and the model may get locked in a poor local minimum because following too strictly the error slope.

The mini-batch is a crucial hyperparameter affecting model quality, training time and cost-efficiency. Touching the mini-batch will alter the scientific performance of the model by changing the amount of noise in the error direction measurement; it will also alter training speed and hardware utilization. As the mini-batch is changed, the amount of communication happening during an epoch and the size of the unit of work sent to CPU and GPU at each iteration both change.

Principled approach for training efficiency

There is an abundance of literature about scientific optimizations for model accuracy improvement. Such techniques include research on model architectures, optimizers, learning rate schedules, data augmentation schemes and regularization. I will not discuss those topics

well covered elsewhere – for example in *Dive into Deep Learning* by Zhang et al. - and available in few lines of open-source code for most of them. Joint tuning of software and infrastructure is however a lesser-known field, with little knowledge and materials. In this section, I dive in joint hardware-software optimization tricks that improve hardware usage and enable scalability.

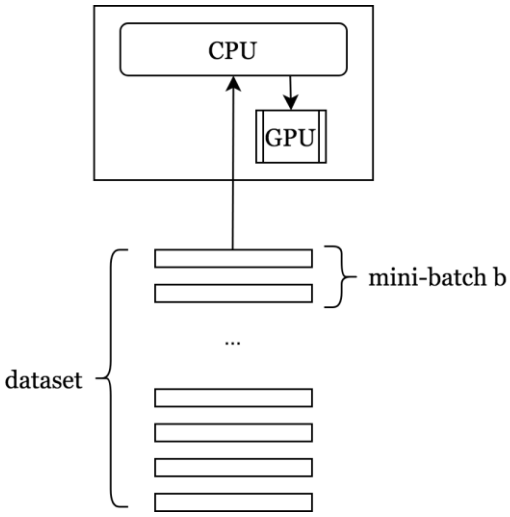

Figure 19. Simplified architecture diagram of a single-GPU neural network training

How to run efficient single-card neural-network trainings

"How do I increase my GPU utilization"? Is one of the questions DL developers ask the most. As mentioned in the section IV.4 dedicated to GPUs, it is not trivial to feed the computational beast that GPUs are. There are challenges - optimization opportunities - at every step of the training pipeline:

Data reading

Data access is a frequent bottleneck. If you expect or measure it to be a bottleneck, try placing data in a low-latency, high-bandwidth location. This could be main memory, fast local disk, for example the NVMe SSD on Amazon EC2 G4, P3dn and P4d instances, or high-performance online storage systems, for example Amazon FSx for Lustre.

Data loading

Data needs to be batched, pre-processed and transported to the GPU. This work usually done by the CPU can also be a bottleneck. In PyTorch and MXNet, developers can control

and tune the number of CPU workers allocated to data loading. Another option is to offload that work to the GPU: NVIDIA created the DALI library for this purpose.

GPU load

The amount of work sent to the GPU can be tuned with 3 levers (1) model size, itself a function of model complexity and input size, (2) mini-batch size, (3) compute precision. Grow mini-batch size to increase GPU use. If mini-batch growth hurts SGD convergence too much, consider applying smaller batches to bigger models, which could improve model accuracy and GPU usage at the same time. Additionally, in order to keep the GPU busy, some frameworks provide the ability to allocate CPU to prepare future records while the GPU is working, so that the inter-batch GPU idle time is minimized. This technique is called *prefetching*.

Mixed-precision training

Mixed precision training is an important training optimization trick that consists in running forward and backward model passes in float16 number precision while applying updates to a float32-precision copy of the model. Mixed-precision training has several benefits: (1) it accelerates communications with GPU memory, (2) it enables storage of bigger batches or models in the GPU memory, (3) on recent NVIDIA GPU equipped with Tensor Core cells (T4, V100, A100), it can use specific NVIDIA Tensor Core cells and accelerate computations. Tensor Cores run a matrix-multiply-accumulate (AX+B) in one operation. Mixed-precision training refers to modifying the precision of the numbers used in training computation and does not necessarily results in a model precision reduction. Mixed-precision training can produce equally-accurate or better models than full float32 precision training. Some DL frameworks such as MXNet and PyTorch provide an Automatic Mixed-Prediction (AMP) functionality.

Mixed-precision training has been a key driver of DL training acceleration. Nowadays, most state-of-the-art vision and NLP training benchmarks are done in mixed-precision and I recommend you to use mixed-precision when using NVIDIA GPUs equipped with Tensor Cores

Figure 20 summarizes optimization tricks improving hardware use

	GPU not busy	*GPU busy*
All CPUs not busy	*Improve disk I/O* *Increase data loading workers* *Increase batch size* *Increase model complexity (if acceptable)*	*Not a huge deal, since costs are GPU-driven.* *Grow GPU/CPU ratio if possible.*
All CPU busy	*Grow CPU/GPU ratio* *Use smaller batch with bigger model* *Pre-process offline* *Pre-process on GPU* *Pre-process on a remote CPU machine*	*Nice!*

Figure 20. How to improve GPU usage. Note that changing batch size and model complexity may disturb convergence quality and require retuning other hyperparameters.

Scaling neural network training: data-parallel SGD

SGD training is conceptually trivial to parallelize. Model prediction, error and its gradient are computed independently on each record of a mini-batch, so that it is possible to fraction a mini-batch and distribute the work with no change to SGD scientific logic. This type of distribution is called *map-reduce* parallelism: units of work (records) are mapped (split) across N arbitrary workers (CUDA cores, CPUs, GPUs, servers) for processing, and individual processing results are reduced (aggregated) once completed. In the case of SGD, the reduction is the average of gradients. This parallelization technique is called *data-parallel SGD*, because the data is distributed, while the model is replicated on each worker. Data-parallel SGD can be synchronous or asynchronous. In synchronous SGD, workers work from the same model and average their gradients after completing their work. In asynchronous SGD, workers update model state without waiting for each other. Synchronous data parallel SGD is more popular than its asynchronous alternative. I actually met asynchronous data parallelism only once: in the Hogwild! (*Hogwild!: A Lock-Free Approach to Parallelizing StochasticGradient Descent*, Niu et al.) training scheme used in SageMaker BlazingText Word2vec (*Amazon SageMaker BlazingText: Parallelizing Word2Vec on Multiple CPUs or GPUs*, Gupta et Khare)

Training on a single machine that has multiple GPUs

This configuration is easy to develop and debug, as a single program runs on a single machine. This could be for example an AWS EC2 p3.16xlarge instance equipped with 8 V100 GPUs. Modern DL frameworks provide abstractions to write single-node multi-device code, such as `torch.nn.parallel.DistributedDataParallel` in PyTorch. Figure 21 represents the high-level architecture of data-parallel SGD in a single-node, multi-device case with 2 GPUs. Note that the mini batch capacity is twice bigger than with one GPU.

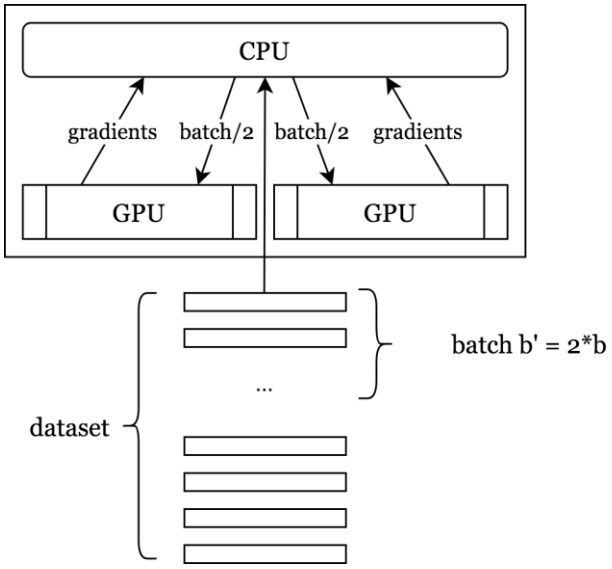

Figure 21. Data-parallel training on a 2-GPU instance

Training on multiple machines that each have one or many GPUs

In this design, several programs must run in sync in multiple machines, themselves equipped with one or many devices. An additional communication challenge arises: the inter-GPU gradient averages now happens across different machines. This over-the-network communication is usually written with NVIDIA Collective Communication Library (NCCL) or the Message Passing Interface (MPI) language, popular in High-Performance Computing (HPC). Two communication topologies are popular: (1) the parameter store and (2) the ring-allreduce.

Parameter Stores

The parameter store paradigm consists of using a dedicated, third-party component to collect gradients from the workers, update the model and distribute the updated model back to the workers. The parameter store can be distributed, and can be collocated on same instances as workers or on different instances. The parameter store was historically the default multi-node distribution method used by MXNet and TensorFlow. In MXNet, the parameter store is called the *Key-Value Store*.

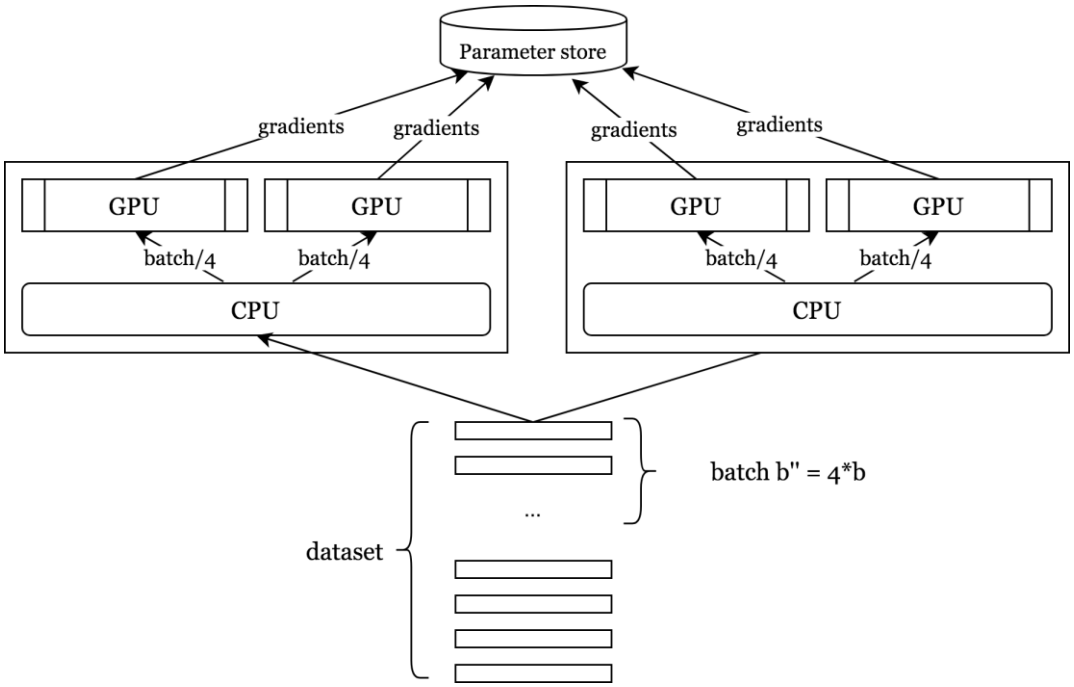

Figure 22. Multi-node, multi-device data-parallel SGD under the parameter server gradient aggregation method, with two 2-GPU nodes, cumulating 4 GPUs

The ring-allreduce

In this design workers form a logical ring where they receive gradients from one neighbor and pass them, along with their own gradients to their other neighbor. After 2(n-1) communication rounds around the ring, each worker has the aggregation of all gradients

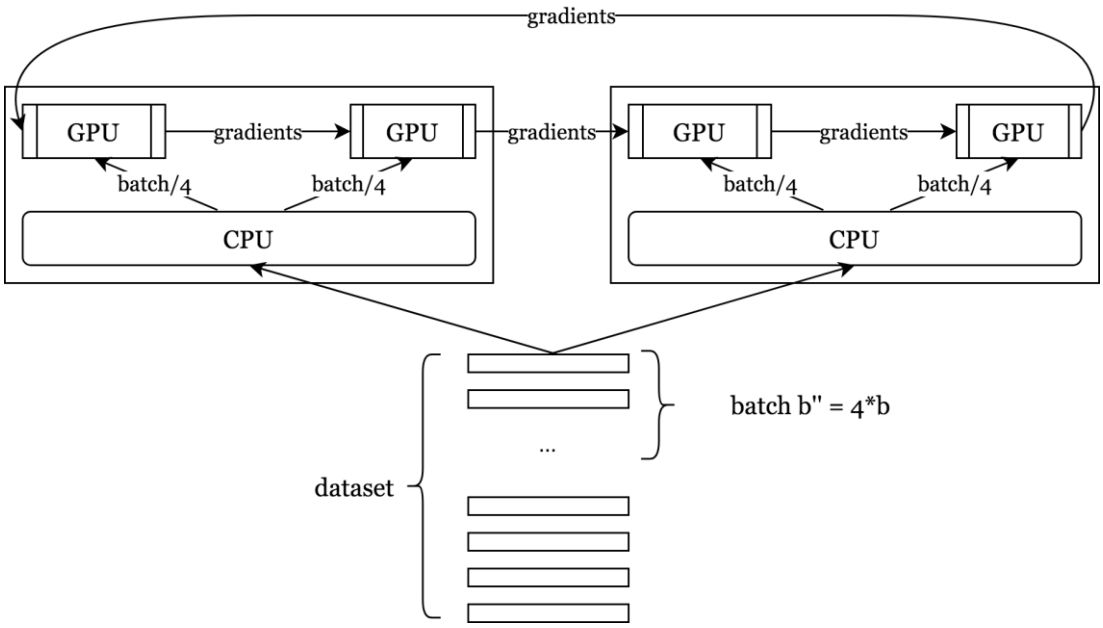

Figure 23. Multi-node, multi-device data-parallel SGD under the ring-allreduce gradient aggregation method, with two 2-GPU nodes, cumulating 4 GPUs

Data-parallel multi-device and multi-node SGD scale mini-batch capacity linearly while preserving the SGD scientific logic. In reality, the actual training speedup is rarely immediately linear. The single-GPU bottleneck of data loading may still happen, and the need for inter-machine communication creates network bottleneck risk. Furthermore, as mini-batch size increases, the SGD gradient average is less noisy and accuracy may degrade. As highlighted by Microsoft researchers Maleki et al. in *Scaling Distributed Training with Adaptive Summation*, there is a trade-off between training cluster throughput and SGD convergence quality. Finding optimizers that maintain SGD convergence despite large SGD batches is the subject of active research.

Because of those 3 bottlenecks permeating both infrastructure and science – data loading, allreduce overhead and SGD convergence - data-parallel SGD is hard. It usually requires additional, ad-hoc research that can make it economically unattractive despite theoretical gains in speed or accuracy. On small models for which the cost of communication is not significantly smaller than the cost of computation, distributed SGD could even be slower than single-device SGD! Do not force yourself to distribute your models if you do not need to. As a Baidu researcher mentioned at the GPU Technology Conference (GTC) 2017, *"don't scale*

until you need to, and know why you're working on scaling". In the next section, we dive deeper into how those frictions can be lifted.

How to realize fast gradient aggregation over the network?

Inter-node synchronization performance is achieved via the combination of 3 items:

Low-latency, high-bandwidth network

Network performance conditions the scaling ability of a distributed training cluster. Many hyperscale distributed SGD trainings occur on super-computers equipped with high-end network. For example, Fujitsu researchers used the Fujitsu-built, AI Bridging Cloud Infrastructure (ABCI) super-computer, 14th biggest super-computer in the world at the time of this writing, to train a ResNet-50 model on the ImageNet dataset. ABCI hosts 4,352 NVIDIA V100 GPUs, 960 NVIDIA A100 GPUs and a 100Gb/s InfiniBand Eighteen Data Rate (EDR) networking (ABCI system overview). InfiniBand is a low-latency high-throughput communication standard that offloads connectivity management from CPU to hardware modules and enables Remote Direct Memory Access (RDMA). A similar InfiniBand EDR high-performance network is found on Summit, the second most powerful super-computer in the world. Summit hosted several notable large-scale distributed trainings over its 27,000+ V100 GPUs, for example in *Exascale Deep Learning for Scientific Inverse Problems* (Laanait et al). InfiniBand network is also used by Facebook (*Scaling Neural Machine Translation*, Ott et al.) and Baidu Silicon Valley AI Lab (SVAIL), the research group that democratized the idea of applying ring-reduction to SGD gradient aggregation and inspired Uber to create Horovod. Given how crucial network infrastructure is for successful high-performance neural network training, and HPC in general, it is not surprising that NVIDIA acquired the InfiniBand pioneer Mellanox in 2020. According to NVIDIA, Mellanox and NVIDIA infrastructure can be found in 250 of the top 500 super-computers in the world. In the AWS cloud, the Elastic Fabric Adapter (EFA) EC2 network interface provides bandwidth of up to 100Gb/s. AWS researchers used EFA to establish training time records of both the BERT NLP model (62 min over 2,048 V100 GPUs) and the computer vision Mask-RCNN model (25 min over 192 V100 GPUs).

Sending less data over the network

A simple yet smart idea to reduce communication overhead is to communicate less. Instead of investing in bigger networking infrastructure, why not using more frugal communication algorithms? For example, the open-source deep learning framework Apache MXNet has a *gradient compression* functionality that sends gradients for weight updates only if they are big enough.

Efficient aggregation algorithm: decentralized allreduce or parameter server?

The previously-described parameter store (PS) reduction suffers from a bad reputation since (1) the addition of the PS component complexifies the system architecture and (2) the PS hosts can be network bottlenecks at the end of every minibatch when they receive gradients from all around the system for aggregation. When it was released, the Uber-developed Horovod ring-reduction was found both simpler and more efficient than native TensorFlow PS SGD distribution. On computer vision tasks, Horovod-distributed models would train up to twice faster as TensorFlow PS distributed models. Additionally, super-scale distributed SGD experiments often happen on supercomputers, where decentralized communication algorithms such as allreduce have been commonplace for decades. Consequently, for the past three years the distributed SGD landscape has been dominated by various flavors of decentralized allreduce algorithms, led by Horovod. SGD gradient aggregation is still an active area of research though. In 2020, the PS made a notable comeback with the publication of BytePS from ByteDance, and of Amazon SageMaker Distributed Data Parallel Library, a state-of-the-art multi-node data-parallel SGD library that internally leverages a parameter server design. The list below gives a sample of the variety and sophistication of gradient aggregation algorithms seen in advanced data-parallel distributed SGD research:

- The *recursive halving and doubling allreduce* is mentioned by researchers at Fujitsu (*Yet Another Accelerated SGD: ResNet-50 Training in 70.4 sec*, Yamazaki) and Facebook (*Accurate, Large Minibatch SGD:Training ImageNet in 1 Hour*, Goyal et al)
- A *2D-torus allreduce* is used by researchers at Sony (*Massively Distributed SGD: ImageNet/ResNet- 50 Training in a Flash*, Mikami et al)
- Tree-based reductions are also popular, appearing in both AWS research (*Tree-based Allreduce Communication on MXNet*, Carl Yang) and NVIDIA NCCL

allreduce backend. Tree-based aggregation algorithms leverage how GPU are positioned to come with the fastest possible aggregation.

- The technology company ByteDance, parent company of the TikTok application, designed the `byteps` parameter server package, that has been found to double the training speed of Horovod in some situations. Among other ideas, BytePS leverages the fact that from the worker perspective the PS paradigm is frugal in communication: it just requires a gradient push and a gradient or parameter pull. Synchronizing parameters across the cluster can be delegated to cheap, CPU-only nodes while GPU workers stay concentrated on model training.
- At the annual summit AWS re:Invent 2020, AWS announced the SageMaker Distributed Data Parallel Library for TensorFlow and PyTorch. It simplifies job management while aiming for superior performance to open-source equivalents Horovod or PyTorch `DistributedDataParallel`. I encourage ML practitioners eager to distribute their neural networks training to consider this option, that should reduce their time to results.

How to maintain SGD convergence with large mini-batches?

As the training capacity of a cluster grows, bigger batches fit in the system memory. This creates a science challenge: using too big a batch size reduces SGD noise and alters convergence. In order to maintain convergence quality, researchers found optimizations to apply to the learning rate schedule and the training optimizer.

Learning rate warmup and scaling

As batch grows beyond a certain point, SGD noise reduces. In order to compensate for the smaller deviations of the sampled gradient around the true gradient, one can increase the learning rate to increase iteration step size and reinstate a healthy level of noise. Additionally, as batch is increased, the number of iterations per epoch decreases. For example, in the extreme fictional case of taking a batch size equal to dataset size, the model would be able to do only one parameter update per epoch! Increasing the learning rate acts as a compensation mechanism: big batches make less iterations, but bigger step size means the model can travel equally far down the error landscape. Learning rate warmup and scaling is discussed in *Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour* by Goyal et al.

Layer-wise learning rate scheduling

Layer-wise Adaptive Rate Scaling (LARS) and its variation Layer-wise Adaptive Moments optimizer for Batch training (LAMB) optimizers have been invented by researchers working on super-scale data-parallel distributed SGD (*Large Batch Training of Convolutional Networks*, You et al ; and *Large Batch Optimization for Deep Learning: Training BERT in 76 minutes*, You et al.). With parameter-agnostic learning rate scaling, there is a risk that the size of the coefficient update gets bigger than the weights. This can create instability. LARS and LAMB counterbalance this risk by adjusting the learning rate per layer. LARS was invented by computer vision researchers and LAMB by NLP researchers. They are available in deep learning frameworks.

Using above-mentioned ideas among others, researchers accomplished notable data-parallel neural network training distributions. Figure 24 provides a non-exhaustive subset of such famous projects.

Title	Model & framework	Optimizer	Scale	Gradient aggregation
Exascale Deep Learning for Scientific Inverse Problems (Laanait et al)	FC-DenseNet (TF)	Adam + LARS	27,600 V100 (Summit)	Modified Horovod
Massively Distributed SGD: ImageNet/ResNet-50 Training in a Flash *(Mikami et al)*	ResNet-50 (NNL)	LARS	3,456 V100 (ABCI)	2D-torus allreduce
Amazon Web Services achieves fastest training times for BERT and Mask R-CNN (Haas et al)	BERT (TF)	*Unknown*	2,048 V100 (AWS)	*Unknown*
Extremely Accelerated Deep Learning: ResNet-50 Training in 70.4 Seconds (Tabuchi et al)	ResNet-50 (MXNet)	Accelerated LARS	2,048 V100 (ABCI)	Variable group allreduce
Highly Scalable Deep Learning Training System with Mixed-Precision: Training ImageNet in Four Minutes Jia et al (Tencent & Hong Kong Baptist University)	ResNet-50 (TF)	LARS	2,048 P40	Hybrid allreduce

Large Batch Optimization for Deep Learning: Training BERT in 76 minutes You et al (Google, UC Berkeley, UCLA)	BERT (TF)	LAMB	1,024 TPU	*Unknown*
Image Classification at Supercomputer Scale Ying et al (Google)	ResNet-50 (TF)	LARS	1,024 TPU	2D-torus allreduce
Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour Goyal et al (Facebook)	ResNet-50 (Caffe2)	SGD with warmup	256 P100	Hierarchical allreduce
From Three Hours to 25 Minutes: Our Journey of Optimizing Mask-RCNN Training Time Using Apache MXNet Yuan et al (Amazon)	Mask-RCNN (MXNet)	SGD with warmup	192 V100	Horovod
Scaling Transformer-XL to 128 GPUs (Bulatov et al)	Transformer-XL (PyTorch)	LAMB	128 V100	*Unknown*
Scaling Neural Machine Translation Ott et al (Facebook)	Transformer (PyTorch)	Adam	128 V100	*Unknown*
Now anyone can train Imagenet in 18 minutes Shaw et al (Fast.ai)	ResNet-50 (PyTorch)	SGD with warmup	128 V100	*Unknown*
Large-Scale Language Modeling: Converging on 40Gb of Text in Four Hours Puri et al (NVIDIA)	LSTM (PyTorch)	Adam	128 V100	Ring-allreduce

Figure 24. Notable data-parallel training achievements

Should you consider data parallel distributed training for your project?

If you cannot fit satisfying batch sizes in one GPU but can still fit at least a unit batch in a GPU, or if your expected training time is unacceptably long, it is a good idea to consider data-parallel distributed training. Given the cost of GPUs, make sure you fully use a single card before scaling to multiple cards. It is almost always a better idea to scale vertically first – using single machine with multiple GPUs – and consider scaling horizontally to multiple nodes only if using the biggest single-machine configuration is not satisfactory. Single-machine trainings are easier to write, easier to debug, and communication overhead is generally smaller within a machine than across machines.

Shouldn't we use ML to tune training configuration?

Of course, we should! We are not there yet though. It is unfortunate: given the size of the search space and GPU costs, applying ML-based optimization to ML training would improve both iteration time and results quality. Neural network training optimization is a multi-objective optimization: one wants to minimize costs and duration while maximizing model accuracy. Several parameters shall be swept to find adequate answers to a given problem, including: data storage and loading configuration, model architecture, hyperparameters, but also communication configuration (how to distribute training) and software configuration (mixed-precision, threading configuration, algebra backends). So far, ML practitioners have been answering that multi-objective problem by answering each question independently, for example considering model accuracy and GPU consumption as different problems, and with intuition and manual trial-and-error. One of my dreams for ML in the next decade is that we see more AutoML tools supporting multi-objective training optimization, so that ML practitioners invest less time learning and applying the tips described in the previous 10 pages. In March 2021, I was excited to see Aerobotics Head of Data Science Michael Malahe use the SageMaker Bayesian Tuner to find configurations of a TensorFlow training job that minimized training time (*Aerobotics improves training speed by 24 times per sample with Amazon SageMaker and TensorFlow*, Michael Malahe). Using ML to tune ML is ML engineering at its finest. It benefits organizations, individuals and the planet. We need to see more of it.

IV.6 Intelligent search of hyperparameters

Why does it matter for the ML practitioner?

The ML practitioner can increase the quality of models while reducing the costs of development by learning and adopting smart hyperparameter search.

Hyperparameters are settings that parametrize the model design and its training. For example, the *learning rate* is a popular numerical hyperparameter in neural network training, controlling the rate of change of the model coefficients between updates. In convolutional object detection model training, the *backbone* is a categorical hyperparameter representing the base model transforming images into vectors of features. ML practitioners often use intuition, heuristics and trial-and-error to find hyperparameters. It is also common to search relevant hyperparameters by programmatically testing a great number of hyperparameter combinations, a brute-force technique called *grid search*. Brute force search is easy to write and is reassuring. However, brute force search costs scale with the power of the number of hyperparameters: 3 hyperparameters make a 3-dimension space to explore. A brute force parameter search on a multi-GPU training tasks with dozens of hyperparameters could literally bankrupt a company or take too long. For example, running a grid search for a fictional 3h-training job with 10 continuous hyperparameters each sampled at 5 values would require 29,296,875 hours of compute, which would cost approximately $117MM if done on V100-equipped AWS p3.2xlarge computing instances and take 3,342 years if done sequentially. When training jobs are expensive, long, or with many searchable hyperparameters, brute for search quickly becomes out of question. Fortunately, there is a smarter way, described below.

Finding hyperparameters relevant to a given problem can actually be assisted by answering the regression question: "*given a set of hyperparameters, can we predict the performance of the model?*". Such a regression could indicate which areas of the hyperparameter space are likely to yield the best performance. Bayesian optimization uses this approach. It is an iterative technique that generates training data by testing hyperparameter combinations, and then uses a model trained on those iterations to suggest areas of hyperparameter space to test next. Bayesian optimization search can lead to order of magnitude faster and cheaper

hyperparameter search steps. AWS makes it more accessible by proposing a managed bayesian optimization hyperparameter search feature in the ML platform Amazon SageMaker. Numerous open-source toolkits are being developed for Bayesian optimization, such as scikit-optimize or BoTorch. Other tools specialize in model hyperparameter search such as Optuna, Hyperopt and RayTune and may use other approaches than Bayesian Optimization. However, I have not seen one become specifically popular among ML practitioners for hyperparameter search yet.

IV.7 Parallelism

Why does it matter for the ML practitioner?

Parallelizing workloads has two benefits: (1) they will execute faster, which often correlates with cheaper, (2) they will better utilize hardware capacity and enable to pack more work on a given hardware perimeter. Costs and time-to-market are major priorities for business leaders. In that regard, the ML practitioner that masters parallel computing will earn trust and love from customers, peers and leadership. Parallel computing is also beneficial for our planet: available infrastructure is used and no resource is wasted sitting idle and waiting for work.

In this section, I propose a principled approach to ML compute parallelism. I present parallelism opportunities in a 4-level hierarchy, from the black-box, specialized parallelism to the ad-hoc, task agnostic parallelism.

1. **Parallelism via packaged model libraries**
2. **Parallelism via DL frameworks**
3. **Parallelism via data processing frameworks**
4. **Parallelism via ad-hoc task scheduling**

Parallelism via packaged model libraries

The easiest way to run parallelized machine learning is to avoid writing parallelization code yourself and use the work of someone else who wrote a distributed algorithm. In this section, "packaged model libraries" refers to algorithms implementations that are developed by other people and that you can instantiate with as little as one line of code. This hands-off experience of ML model development is common thanks to ML libraries such as Scikit-Learn, HuggingFace Transformers and Spark ML. Popular algorithm implementations that run distributed - sometimes without you noticing - include:

- Spark's Matrix Factorization `ALS`.
- Scikit-Learn's Random Forest, whose parameter `n_jobs` trains individual parts of the model, decision trees in this case, in parallel during training.
- Many Amazon SageMaker built-in algorithms have been written to fully use available hardware and train on multiple GPUs and multiple machines without extra effort from developers.
- The collaborative filtering recipes developed in the open-source package `implicit` have multi-threaded implementations leveraging all available CPU cores.

Parallelism via deep learning frameworks

Deep learning frameworks are specific data processing frameworks for neural network development. They abound in parallelism opportunity:

Implicit parallelism: without developer intervention

Many frameworks have multi-threaded backends, for example both MXNet and TensorFlow are written in C++. Consequently, low-level execution of the high-level developer commands will often be parallelized, leveraging multiple CPU cores or the multiple CUDA cores of GPUs.

Explicit parallelism: with the developer intervention

Advanced deep learning frameworks include functions for developers to explicitly program additional levels of parallelism, over multiple CPU, GPUs and multiple machines. This includes:

Data parallelism

Data parallel network training is the object of section IV.5. It consists of splitting the SGD mini-batch across multiple copies of the same neural network hosted in different GPUs. Data parallelism is useful when batch size needs to be increased, for example when a single GPU cannot fit a big enough batch for convergence. This is for example often the case of transformers language models or high-definition computer vision models.

Data parallel training is often presented as a way to speed up training, a claim that has to be taken with a grain of salt. As discussed previously, changing the batch size of neural network training disturbs convergence and it is likely that many hyperparameters will have to be re-tuned. A portion of gains from training time reduction will be lost in model re-tuning.

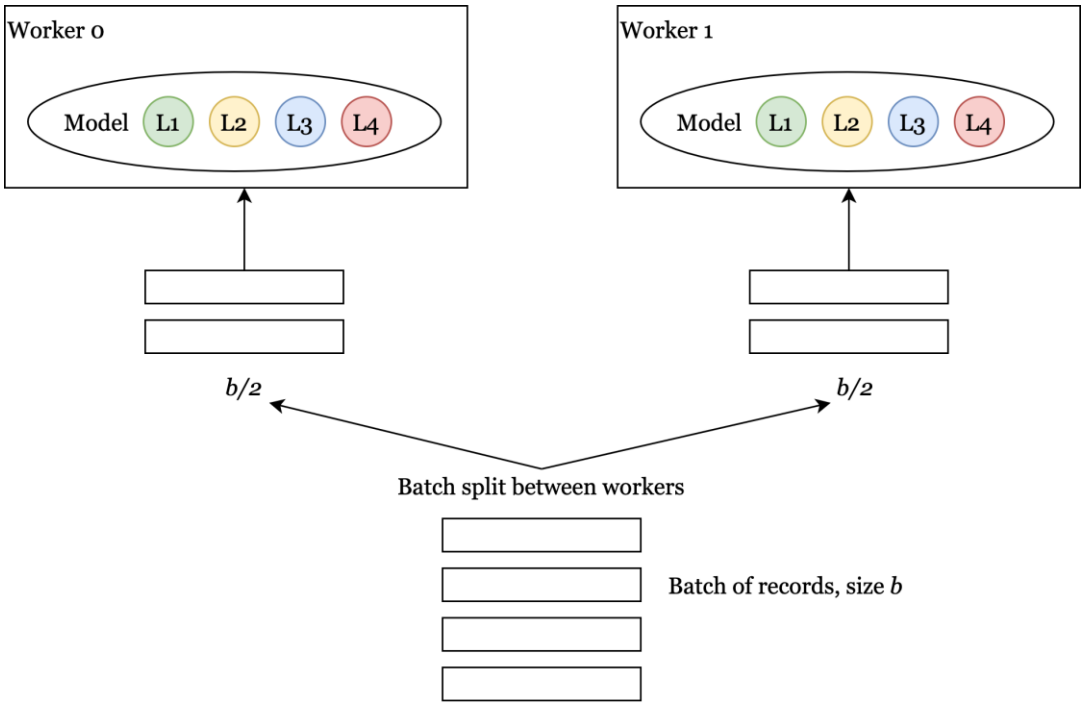

Figure 25. Data parallel training: workers store identical full model copies and split the SGD batch among each other to compute the forward and backward passes. The gradients are averaged across all workers so that each worker applies the same update to its model and models stay identical across iterations

Model parallelism

In model parallelism, different parts of a model reside on different devices. It is sometimes called *workload partitioning*.

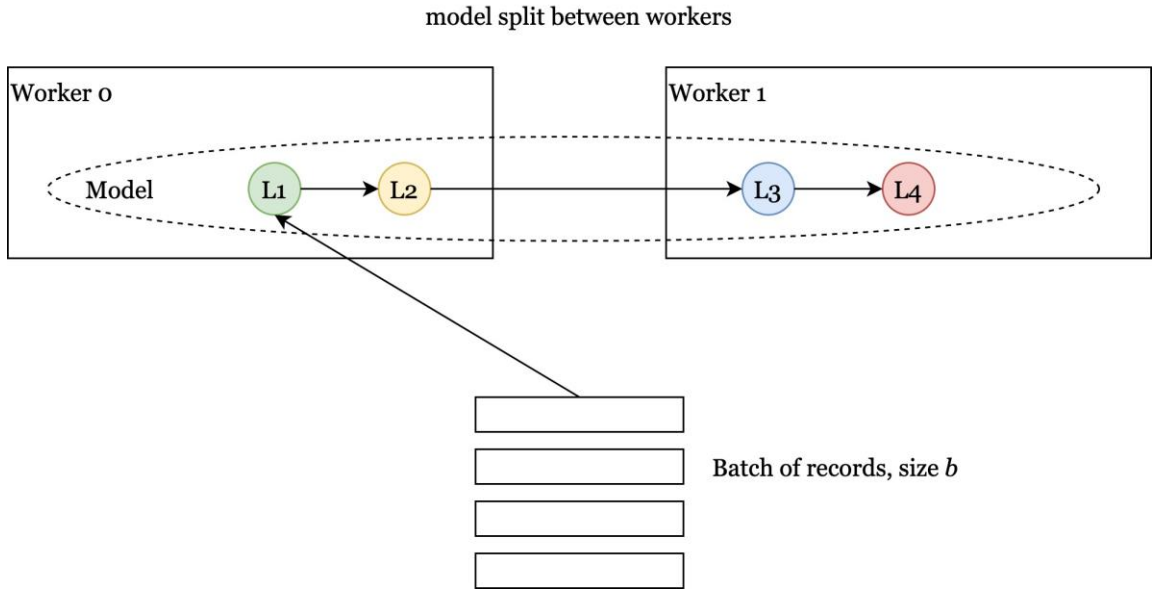

Figure 26. Model-parallel training

Compared to data parallelism, model parallelism is more frugal: the model is partitioned, instead of being replicated, and workers share only intermediary data from layer to layer, instead of sharing all model gradients. However, model parallelism is more difficult to implement. I saw it mentioned in a handful of research papers: In *Google's Neural Machine Translation System: Bridging the Gap Between Human and Machine Translation* (Wu et al)., Google researchers describe an 8-layer RNN encoder-decoder model that mixes model parallelism with data parallelism: during training, each mini-batch is split per nodes, and in each node one single model copy is partitioned over the 8 available GPUs, with one model layer per device. Amazon researchers also published at least twice about model-parallel training: *The Effectiveness of a Two-Layer Neural Network for Recommendations* (Rybakov et al) and *Semantic Product Search* (Nigam et al.). Google released two model-parallelism toolkits (*Mesh-TensorFlow: Deep Learning for Supercomputers,* Huang et and *GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism,* Huang et al). 2020 saw an activity increase in the field of model parallelism: Microsoft engineers published the DeepSpeed open-source library,

that powered several NLP publications. At re:Invent 2020, AWS released SageMaker Distributed for Model Parallelism, in my knowledge the first production-grade, managed model parallelism library. It is notable for its automated model partitioning algorithm that splits models among GPUs in a way that optimizes for speed or for memory usage.

> Model-parallel training is rare and difficult to implement manually. So, far model-parallelism open-source tools were mostly ephemeral research projects. In 2020, Amazon SageMaker Distributed Model Parallel and Microsoft DeepSpeed democratized the concept and attracted attention on the benefits of well-implemented model-parallel training.

Multi-worker data loading

Data loading is another deep learning domain that developers are encouraged to distribute. Multi-worker data loading consists of parallelizing the data ingestion code over multiple CPU cores so that mini-batches can be formed rapidly and the GPU kept well busy. MXNet and PyTorch enable users to tune this level of parallelism via the `num_workers` argument of their `DataLoader` object.

Parallelism via data processing frameworks

If there is no packaged distributed implementation for your task of interest, and if this task is not a neural network training, it is worth considering writing it yourself. Data processing frameworks allow you to write transformations on structured data. They occasionally come with built-in ML algorithm implementations - for example Spark includes `spark.ml` and Dask includes `dask_ml` – and are appreciated for their use-case agnostic interface that enables developers and scientists to create custom data transformations. Numerous generic data processing frameworks have built-in parallelism capacity.

Apache Spark

Spark is the most popular distributed data processing framework, deeply integrated in AWS, both in Amazon Elastic Map Reduce (EMR) cluster computing service, as a managed service in Amazon Glue and as a managed container in Amazon SageMaker. Numerous prominent organization publicly mention using Spark, including Netflix, Amazon, FINRA or Yelp. Spark is written is Scala and has a popular Python front-end called PySpark. According to Spark 3.0.0 release notes, PySpark is the most popular Spark front-end, with 5MM monthly

downloads. Spark leverages three levels of parallelism: A Spark program runs over a cluster of physical or virtual machines, each machine is partitioned in a group of workers called *executors*, and each executor can be provisioned with a user-defined number of CPU cores. If you can write your program in Spark formalism, chances are it will be distributed by default and leverage a decent portion of available resources. Resource usage can be fine-tuned by properly setting the amount of memory and CPU per Spark executor, a task that – last time I used Spark – was mostly a work of trial-and-error. Spark is the only distributed computing framework in this list for which I know several production-grade references.

Apache Spark is routinely seen in production, including in the most demanding organizations. Being written in Scala, it however presents a brutal learning curve for ML practitioners used with to the compacity and simplicity of the Python scientific stack. In the past few years, several projects were created to bridge the gap and offer large-scale data processing capacities with a friendlier experience, for example Dask, Vaex, Modin or NVIDIA Rapids. At the time of this writing, I have not seen yet significant production use of those emerging alternatives, presumably because of their younger age.

Emerging data processing frameworks

Dask is a distributed data processing framework whose API looks more familiar to Python scientific stack interfaces (Numpy, Pandas). Modin is a Pandas clone than can run distributed. Modin is interesting for two reasons: (1) it uses the same API as Pandas, which PySpark and Dask failed to do - Modin initial name was actually *Pandas on Ray* ; (2) it is built on top of Ray, an asynchronous distributed task scheduling framework that has a growing popularity in the RL and DL space. Vaex is another distributed data processing framework, that I heard about often enough to mention in this list, yet not enough to position in figure 25. At the moment Vaex seems to be restricted to single-machine tasks, an acceptable constraint given how big single machines can be: at the time of this writing AWS cloud instances can contain up to 224 cores and 24Tb of RAM. I encourage the audacious ML practitioner to explore this lesser known yet promising framework. Finally, NVIDIA RAPIDS is a set of tools for data analysis on NVIDIA GPUs. RAPIDS includes the GPU dataframe processing library `cudf` that mimics the famous CPU-only `pandas,` and the GPU machine learning library `cuml` mimicking the famous CPU-only `sklearn`. RAPIDS is recent - announced in October 2018 - and I did not see it used in production yet.

Which one should you use? The decision tree on figure 27 proposes a principled selection approach

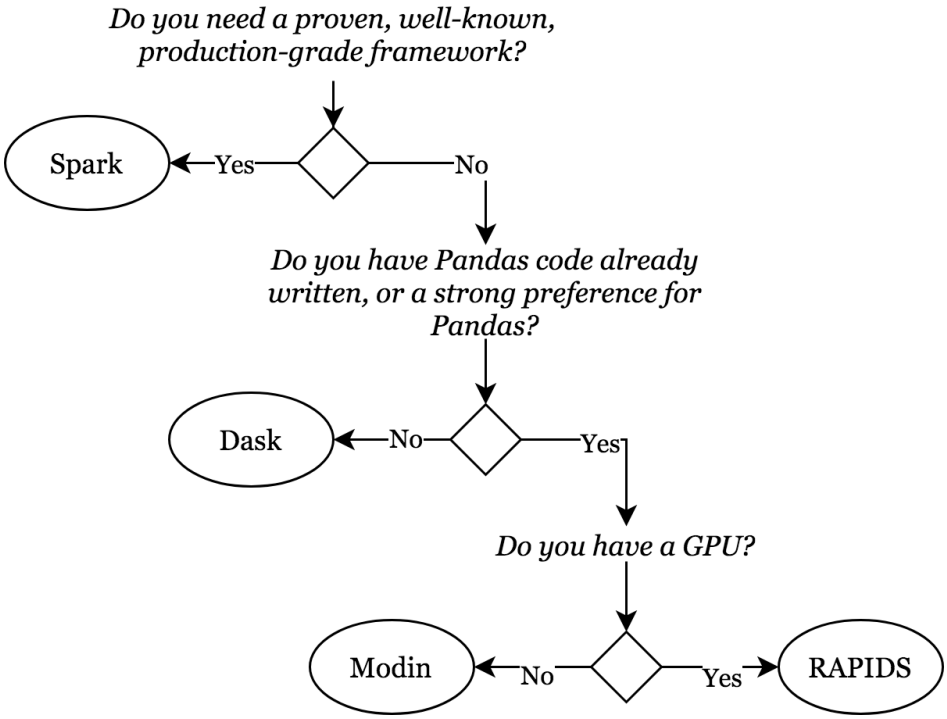

Figure 27. Choosing a Python-based data processing framework. Spark is Scala-based but its most popular frontend, PySpark, is in Python

Parallelism via ad-hoc task scheduling

In the two previous paragraphs I presented specific workloads and associated software that force developers into a given parallelism design: model training or data processing. Beyond those rigid approaches, ML practitioners should feel empowered to decide for themselves which parts of their workflow to parallelize. ML projects are full of parallelism opportunities. In this paragraph, I first present typical ML workflow steps that can be parallelized, and point to tools that can assist users running parallel computation. I skip model training, already covered by previous paragraphs.

Data preparation

This step often offers parallelism opportunities. For example, if your data comes from multiple sources, can you collect it in a parallel fashion? If your data consists of multiple files, can you distribute the treatment over each file in parallel? If your data consists of multiple records, can you distribute the treatment over each record in parallel?

Model testing and validation

Those steps are prone to parallelism. For example, if the model test process consists of training and testing over N possible splits of the data, all N variants of the train-test split can be processed in parallel, at the same time.

Hyperparameter tuning

Hyperparameters are settings that parametrize the model design and its training. There is no fixed list of what ML practitioners consider hyperparameters, which can be numerical or categorical. Hyperparameters can be searched and tested methodically. Most hyperparameter search techniques contain exploration phases during which several hyperparameter combinations are tested. Those exploratory iterations can run in parallel.

Bagged ensembles

Ensembles are models made of several models. Bagged ensembles are made of the average of multiple independent models. The models composing a bagged ensemble can all be trained in parallel.

Batch Inference

Batch inference consists of obtaining model predictions for a group of inputs. In batch inference, the developer controls the schedule of the predictions and several records can be inferred at the same time. This task can be well parallelized.

How to write ad-hoc parallel programs?

Loops with asynchronous APIs

An API call is asynchronous if it does not wait for the end of required processing to complete. Using asynchronous APIs is a simple way to run tasks in parallel. For example,

in the Amazon SageMaker ML platform, both training and processing jobs calls can be asynchronous when launched with `boto3` or when setting `wait=False` in their Python SDK calls. Launching multiple asynchronous training or processing jobs one after the other with a for loop will run them in parallel.

Workflow schedulers (Airflow, Step Function)

Workshop schedulers are software programs that allow developers to define a graph of instructions. Those graphs can be arbitrarily complex and commonly mix sequential steps with parallel steps and conditional executions. Workflow schedulers enable developers to program some tasks to run in parallel. They are usually seen for long-running jobs and low-frequency state changes (> seconds)

Python `multiprocessing`

Multiprocessing is a Python package providing process-based parallelism in Python. It can open several instances of Python to distribute work over the different CPU cores of a single machine. A famous, method of the `multiprocessing` module is its `Pool.map`. It emulates the native Python `map` function and applies a function to the elements of a list in a parallel fashion. For example:

- Let `F(file)` be a Python function that processes a file identified by its name or URL `file`
- Then the following command will apply `F` in parallel to the list of files `files = [file1,…, fileN]`, using one worker per CPU core :

```
import multiprocessing as mp
with mp.Pool(mp.cpu_count()) as pool:
     Results = pool.map(F, files)
```

You can for example use `multiprocessing` to apply a function to a Pandas dataframe column using all CPU cores of a machine:

```
import multiprocessing as mp

with mp.Pool(mp.cpu_count()) as pool:
    df['newcol'] = pool.map(f, df['col'])
```

Ray

Ray is a modern asynchronous distributed computation engine that can run arbitrary tasks in any order of execution, including in parallel, on a single machine or a cluster of machines. Ray is use-case agnostic and fit for tasks requiring parallelism without constraints. ML practitioners appreciate several of its characteristics: it can run tasks asynchronously, it uses a shared memory with low communication overhead (based on the Apache Arrow *Plasma Store*) and runs on CPU and GPU. Consequently, it is no surprise to see Ray spearheading a distributed ML revolution and powering new ML development toolkits across the whole model lifecycle:

- Modin is a Ray-based data processing framework.
- RaySGD is a Ray-based distributed SGD (training) framework.
- RLlib is a Ray-based distributed reinforcement learning framework.
- Tune is a Ray-based model tuning framework.
- RayServe is a Ray-based model inference serving framework.

Ray strengths are also its weaknesses: being an asynchronous, distributed system, the technical bar to entry is high and I would not recommend it as a must-learn to junior ML practitioners. It can be a useful tool for the experienced practitioner that routinely writes custom distributed computation systems.

Ray has strong traction in both academia and enterprise, and will likely grow to be a top ML framework in the next decade, supporting all steps of the ML development lifecycle benefiting from parallel computation.

IV.8 Serving and inference optimization

Why does it matter for the ML practitioner?

Real-time model serving is a common deployment pattern for machine learning models. Use-cases such as fraud detection, personalization, search or translation require highly-available, low-latency models. Related knowledge is still rare among ML practitioners, who tend to focus on algorithm science and model training. ML practitioners possessing a crisp understanding of inference optimization techniques will be appreciated for their ability to create customer-facing products, drive latency down while reducing costs.

The ML practitioner should be suspicious of the word *optimization* seen alone. Optimization is always conducted with one or multiple goals in mind. If you are asked to optimize a system, task or object, always ask what are the underlying goals and which metrics - if any - are to be minimized or maximized. In the context of ML model serving, a common optimization question is to manage to run predictions through the ML system within a given envelope of latency, costs and throughput. Below are typical inference optimization questions:

- *"We want to deploy our image classifier and expect 100 requests per second. What is the minimal possible budget?"*
- *"We want inference to take less than 10ms, the costs are less a priority"*
- *"Our ML system takes 5s to process a request. How can we reduce the latency?"*

In order to answer those questions, one should adopt a principled approach and observe that an ML serving system is made of several components:

1. A **model file** produced as a result of a by the training.
2. A **model execution environment** running the model over input data.
3. A **serving layer**, providing interaction ability. This often is an HTTP web server.
4. A **hardware backend** hosting the execution environment and the serving layer.

Those components cluster in the 3 axes pictured in figure 28. Note that I use a dashed line to represent the model algorithm axis, since changing the algorithm usually changes the system accuracy.

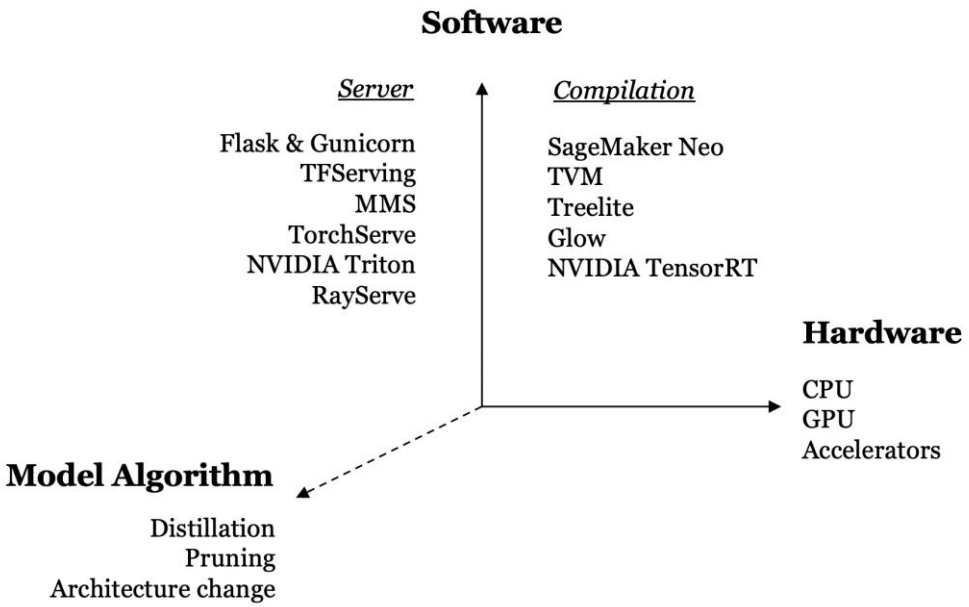

Figure 28. Real-time inference optimization techniques cluster in 3 axes: model, software, hardware. Optimizations that alter the model usually disturb its accuracy, hence the dashed line in the above diagram. Conversely, software and hardware optimization generally do not alter model quality. I initially proposed this representation of the inference optimization problem in a 2020 MXNet blog post (https://medium.com/apache-mxnet/faster-cheaper-leaner-improving-real-time-ml-inference-using-apache-mxnet-2ee245668b55)

Hardware Optimizations

Tuning the hardware backend of a model serving endpoint can improve latency, throughput and costs. While testing fifteen server types can be lengthy or barely feasible in a physical, on-premise infrastructure context, in the cloud it is as easy as launching a for loop over a list of configurations. Common hardware options for ML inference are the following:

CPU-only machines

CPU machines have the benefit of being flexible, mature, well-documented and well-known. They are commonly used for ML inference. The clock frequency of the CPU and its number of cores influence both latency, throughput and costs. CPU architecture is also an optimization lever: the Arm-based, Graviton2-equipped EC2 M6g instance for example instance runs BERT NLP inference 28% faster than the Intel-equipped EC2 M5 instance

while being 20% cheaper (*AWS re:Invent 2020: Deep dive on AWS Graviton2 processor-powered EC2 instances*). Model compilation and CPU improvement makes CPU machines relevant options for ML inference.

GPU-equipped machines

GPUs are relevant when there is a parallelism opportunity. This is typically when a single inference involves a lot of computations, for example for large deep learning models, or when there is an opportunity to run inferences in batches. GPU however suffer from two drawbacks: (1) their core frequency is generally smaller than high-end CPU frequencies and (2) they require extra data transport of data from main memory to GPU memory. Consequently, using GPUs for inference is not guaranteed to present better economics than using CPUs. Given the high price point of GPU instances, I recommend to always run tests to quantify their relevancy against CPU instances.

AI Accelerators

ML practitioners can also decide to use devices specifically designed for ML inference. Mature AI accelerators include Google TPU and AWS Inferentia. AI accelerators promise better inference economics at the cost of hardware specialization. AI accelerators may require developers to conduct the extra step of converting the model in a format supported by the accelerator. For example, the AWS Inferentia chip runs a specific runtime named the Neuron SDK, that requires models to be compiled ahead-of-time. Compilation brings several benefits, described in the next section.

Software Optimizations

Model Compilation

Compilation is a post-training step that consists of specializing the model graph and its prediction runtime into a lean, single-purpose prediction application. Compilation has several benefits: (1) it reduces the memory and computational footprint, (2) it simplifies the inference dependency stack, replacing the possibly complex machine learning development framework and its dependencies by a single compiled runtime application.

Machine learning compilers use a variety of optimizations. For example, the TVM paper (*TVM: An Automated End-to-End Optimizing Compiler for Deep Learning*, Chen et al) mention *operator fusion,* consisting of fusing multiple operations to avoid intermediary communication with memory, and *data layout transformation* that arranges computations to leverage the hardware design, for example, splitting a matrix calculation in smaller chunks to fit in a physical matrix multiplier module. TVM is the leading open-source framework for model compilation, and was co-authored by the famous machine learning scientist Tianqi Chen. TVM is compatible with MXNet, PyTorch, Keras, Caffe2, CoreML, TensorFlow, supports automatic tuning and mobile and edge hardware. In its initial publication, TVM-compiled models ran approximately 60% faster than their TensorFlow Lite equivalent on Arm A53 mobile-grade CPU. Besides, GPU inference was 40% to 85% faster than TensorFlow for a variety of models on NVIDIA Titan X GPU (ResNet-18, MobileNet, LSTM, DQN, DCGAN). TVM co-author Chen is also a co-author of XGBoost and Apache MXNet, consequently it is not surprising that TVM internally uses gradient boosting to model the benefits of optimization routines and assist with compilation choices - a technique found to be superior to random search and genetic algorithm alternatives. Tianqi Chen is now CTO and co-founder of OctoML, a promising technology company developing solutions to facilitate deployment of machine learning models. Glow is a PyTorch-only compilation stack, at this stage less popular than TVM. NVIDIA also created a compilation stack for its own hardware, TensorRT. Though a lot of the attention is currently on deep learning compilation, the concept of inference artifact specialization is also relevant for non-neural network models. In particular, the Treelite compiler was co-authored by Amazon researcher Mu Li and optimizes tree-based models such as XGBoost, decision trees or random forests. Official Treelite benchmarks indicated 2x to 6x throughput improvement. Both Treelite and TVM are dependencies of SageMaker Neo, a managed model compiler created by AWS. Neo simplifies the model compilation experience by providing a single API to compile models from popular frameworks (MXNet, TensorFlow, XGBoost, PyTorch) to popular hardware targets (Intel, NVIDIA, Arm and others). Neo uses several in-house innovations developed by AWS; many are documented in public research papers such as *Optimizing CNN Model Inference on CPUs* by Liu et al and *A Unified Optimization Approach for CNN Model Inference on Integrated GPUs* by Wang et al. Microsoft-developed ONNXRuntime is another compilation stack, open-source and with a growing popularity. In a personal test, I was amazed to see a trained 130-feature Sklearn Random Forest running 200x faster in ONNXRuntime than in Sklearn on an Amazon m5.xlarge instance, at 31 microseconds instead of 6 milliseconds.

Model server

The model server is a software component exposing the model as a web service. Popular options for ML serving include Flask (paired with Gunicorn), Multi-Model Server (MMS) and TensorFlow Serving (TFServing). Those three servers are visible in the open-source backends of the Amazon SageMaker hosting service, respectively for Scikit-Learn PyTorch and MXNet, and TensorFlow, a solid testimony of their relevancy and reliability. Other ML serving options include NVIDIA Triton Inference Server and TorchServe, an MMS-inspired serving stack co-developed by AWS and Facebook for PyTorch. Two ML server controls are particularly important:

Worker count:

Model servers generally expose a parameter to control the number of model instances available to serve prediction requests concurrently. Exposing multiple model workers helps handling concurrency, yet can cause infrastructure contention.

Request batching

Batching opportunities arise at two levels. First, some business problems lend themselves to client-side batching: it is possible for the client to group prediction requests and send them as one payload to the ML server. If you see yourself writing a loop of server requests in your application, try instead to group requests and send them as one or few grouped server calls, to minimize the communication overhead. For example, DoorDash describes that for the restaurant ranking model in their food order app, they score all eligible restaurants in real-time batched inference, and tuned the client-side batch size to minimize latency (*Meet Sibyl – DoorDash's New Prediction Service – Learn about its Ideation, Implementation and Rollout*, Cody Zeng) Furthermore, advanced deep learning servers such as TFServing, MMS, TorchServe and NVIDIA Triton feature the possibility to do server-side batching, a mechanism that has the server accumulating requests opportunistically during a user-defined temporization, and then internally send batched payloads to the inference workers. Note that in that context, server-side batching actually runs the inference asynchronously, but since the queuing time is small, clients have the illusion of synchrony. Server-side batching improves infrastructure utilization and throughput by making a developer-controlled concession on latency. It can also lead to higher concurrency capacity than fully synchronous processing of single-record batch

request. Salesforce's Einstein AI team ran a benchmark of Hugging Face's PyTorch BERT-base NLP model served over NVIDIA Triton Inference Server; queried by 100 concurrent threads, the server could deliver over 500 predictions per seconds with server-side batching activated, over 300% more than single-record inference (*Benchmarking Triton Inference Server for Transformer Models*, Nitish Shirish Keskar)

Model Optimizations

A tempting answer to the inference optimization question – be it latency improvement or cost reduction – is to revisit the model science so that it contains less computations. This path is however treacherous, since changing the logic and equations behind a machine learning model is likely to disturb its accuracy. The techniques described in this section are more intrusive than the above-mentioned hardware and software optimization and some of them require revisiting the model training procedure and making concessions on model accuracy.

Model Compression

Model compression refers to reducing the number of weights of a model. There are two well-known model compression techniques: pruning and distillation. Model pruning consists of removing from a model the weights that can be removed without significantly altering the model accuracy. In *Pruning machine learning models with Amazon SageMaker Debugger and Amazon SageMaker Experiments*, Rauschmayr et al. prune a PyTorch ResNet using SageMaker Debugger and reduce parameter count by 67% while reducing accuracy only by 1.7 percentage points. Distillation consists of training a small model to mimic the predictions of a large model. Pruning and distillation often appear in enterprise and academic research, for example in *DistilBERT, a distilled version of BERT: smaller,faster, cheaper and lighter* by Sanh et al. Production use of pruning and distillation is still rare and I am not aware of mature, broadly-used tooling supporting it. The open-source AutoML package AutoGluon is the first use-case agnostic managed distillation tool I am aware of. Thanks to its `distill` method, AutoGluon transforms complex ensembles obtained after training into lean, fast predictors.

Architecture change

Some machine learning problems are so common that researchers invested time inventing ad-hoc architecture optimized for speed and costs. For example, lightweight architectures have been designed for image classification, such as SqueezeNet (*SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size*, Howard et al) and MobileNet (*MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*, Howard et al). According to GluonCV model zoo, `MobileNet_v3_large` runs 6.4x faster than VGG19, while being more accurate on the ImageNet dataset and 6x smaller in memory. The problem of nearest neighbor search is permanently tackled by researchers and new entries frequently appear in the benchmark maintained by Erik Bernhardsson. For example, the two popular optimized kNN libraries SCANN (*Announcing ScaNN: Efficient Vector Similarity Search*, Philip Sun) and HNSW (*Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs*, Malkov et Yashunin) run respectively 375x and 88x faster than exact neighbor search, with 95% recall.

System architecture considerations

Serverless inference

If your Model can run on CPU and receives medium traffic – typically in the single-digit request per second or less, AWS Lambda can reduce infrastructure costs while avoiding the need to manager servers.

Caching

In certain use-cases such as search or recommendation, it may happen that several prediction requests contain the same input. For example, in a search engine, the query *"best restaurant in Lyon"* is likely submitted several times a day. For those use-cases, instead of repeatedly re-running the model over the same input, it may be relevant to store input-predictions pairs in a cache to avoid re-soliciting the model with an input for which we recently saw its results. Fast key-value stores such as Amazon DynamoDB or Redis are relevant options for a caching layer. Advanced ML serving systems sometimes feature a built-in cache functionality, such as the open-source Clipper server (*Clipper: A Low-Latency Online Prediction Serving System*, Crankshaw et al), which was unfortunately recently archived.

ML inference architecture needs you

Like model training, model serving can be framed as a multi-objective optimization problem. One wants to minimize cost and latency while maximizing throughput and accuracy. To optimize those targets, ML practitioners need to browse a wide search space - model architectures, compilers, hardware and server settings – a search today mostly done manually, using empirical knowledge and intuition. One of my hopes for ML in the next decade is that the community will come up with AutoML tools that will navigate the configuration space on our behalf and learn optimized serving architectures. This is a domain where we can and should improve.

IV.9 Causal Inference

Why does it matter for the ML practitioner?

In business, it often happens that the power to predict is less valuable than the power to determine causal relationships. For example, instead of having models predicting customer churn, business leaders may prefer knowing the specific drivers that cause churn in order to fix them. Unfortunately, current ML models are almost always designed to excel at prediction only. They find correlations between an input and an output. Correlations may provide great predictive power but they do not imply causation. For example, the number of firefighters may be a good predictor of the number of fires in a city, but that does not mean that firefighters cause fires, nor that removing firefighters will reduce fires. Other example: marketing budget may be a strong predictor of sales, but that does not mean that increasing marketing will increase sales. It could just be that the marketing team has been doing a great job at allocating marketing support to product that would sell well. Awareness of causal inference methods help ML practitioners estimate the impact of initiatives and guide decision makers. It enables the estimation of quantities often believed impossible to estimate. Among

all the concepts described in this book, causal inference is by far the most rarely seen among ML practitioners, while being valuable in data-driven organizations.

Several books have been written on causal inference alone. This book does not teach causal inference but provides directions of research for the curious reader to explore.

The randomized experiment, also called A/B testing in the digital world, is the gold standard to estimate the causal impact of a treatment on an outcome of interest. However, numerous questions cannot be answered by A/B test, either for practical, ethical or legal reasons. For example, how do you estimate the impact of a service disruption on churn? Would you really want to force service disruption for the sole purpose of measuring its impact? How do you estimate the average impact of doing a PhD or MBA on employee salary? It is obviously not possible to measure this with A/B testing: you cannot force some people to follow a specific career path, and some others not to in order to design your test and control groups. Equally challenging: how do you estimate the impact of a broken leg on school results? No matter the country you live in, you likely won't be allowed to collect students and break their legs to run a randomized experiments on this topic… Social sciences and economics are rich with such questions that cannot be answered with randomized experiments, for example measuring the impact of government policies and of life events.
Economists have been thinking about this fundamental problem for decades and numerous methods have been proposed to estimate causal impacts from non-randomized, observational data. While these methods have their roots in biostatistics, econometrics and social sciences, top technology companies use them at a growing frequency. Uber and Netflix regularly publish about the topic (*Using Causal Inference to Improve the Uber User Experience*, Harinen et Li; *Computational Causal Inference at Netflix*, Wong et McFarland), and Amazon is active in the domain with several publications and a recruitment page dedicated to the hiring of economists. Several techniques regularly appear in public causal inference research. I provide in the following paragraphs a high-level conceptual overview of them. Note that many come with specific assumptions and weaknesses, so I encourage you to read further research when applying them to your causality problems.

Propensity Score Modelling (PSM)

PSM consists of building a classifier to predict the likelihood to be treated, so that one can compare units that had similar selection bias. In order to build that classifier, one needs to

gather and use pre-treatment features assumed to be predictive of the treatment assignment. In that regard, one significant disadvantage of PSM is that it requires availability of rich, granular pre-treatment data and of a data processing platform enabling the joining of multitude of features. PSM is a well-established and popular technique for causal inference on observational data. It is mentioned in causal inference publications of Uber, Netflix (*Causality without Headaches*, Benoît Rostykus) and Cisco (*The Big Three: A Methodology to Increase Data Science ROI by Answering the Questions Companies Care About*, Daniel K. Griffin)

Synthetic Control

Synthetic Control consists of building a synthetic twin of a treated unit of interest, estimating what would have happened without treatment. The synthetic twin is built as a linear combination of non-treated units that has similar pre-treatment behavior as the treated unit. Synthetic Control has two notable benefits: (1) it enables impact estimation on individual units, and (2) it is frugal in data, using only the outcome variable and no covariates. For example, Synthetic Control was applied to estimate the causal impact of the tobacco control program Proposition 99 on tobacco sales in California, an analysis described in *Synthetic Control Methods for Comparative Case Studies: Estimating the Effect of California's Tobacco Control Program* by Abadie et al.

Regression Discontinuity Design (RDD)

RDD is a design applied to the edge case where the treatment is given based on a metric value against a threshold. Units across each side of the threshold are expected to be similar in all points except that some are treated and some are not. For example, if a software company wants to reduce churn rate by providing "Free 24/7 White Glove Technical Support" to customers spending more than $500K a year, then comparing the post-treatment churn rates of customers spending slightly less than $500K and customers spending slightly more than $500K one will may estimate the causal impact of that initiative on churn.

Numerous other techniques exist, yet I particularly like the 3 aforementioned which I find intuitive. If this introduction to causal inference triggered an interest, the reader should not hesitate to explore additional concepts such as *Difference-In-Differences*, *Doubly Robust* and *Instrumental Variables*. The broad adoption of causal inference techniques by top technology companies drives the creation of new tools lowering barriers to entry. Notably, Microsoft

released the open-source EconML library that implements state-of-the-art causal inference techniques, and the ML software company Dataiku announced in June 2021 being working on a causal inference toolkit.

Further reading

Scientists Susan Athey, Judea Pearl and Guido Imbens produce several quality contributions to the field of causal inference and its intersection with machine learning. I recommend browsing their publications to find research relevant to your knowledge gaps. They also authored notable books, for example:

- *Causal Inference in Statistics: A Primer,* by Pearl et al.
- *Causal Inference for Statistics, Social, and Biomedical Sciences*, Imbens et Rubin

Next Steps

ML is a fast-developing field where practitioners are rewarded with the excitement of a permanent learning curve. In this book, I provide key pillars of knowledge for individual and organization to become efficient at applying machine learning to real-world business problems. The book is not a standalone reading, but rather the beginning of your journey towards pragmatic, impactful ML. Numerous pointers are provided throughout the book, and I encourage curious readers to follow those treads, probe their relevancy to their personal journeys and shape their unique learning path. I furthermore provide extra resources and technical details in the following appendixes. I always enjoy discussing about ML, so do not hesitate to reach out for comments, questions and ideas!

Olivier

Appendix A1 – 19 Algorithms Often seen in the Field

A1.1 Classification and regression on tabular data

A large share of ML science is dedicated to tabular data, sometimes called structured data. In 2021 3 algorithms handle most of those use-cases: linear models, variants of Gradient Boosting and Random Forests. Other paradigms exist, such as Support Vector Machines (SVM), k-Nearest Neighbor (kNNs), or multi-layer perceptron (MLP) but they are much less seen than the three aforementioned.

*Example of ML over structured data: fictional **regression** problem, where we try to predict a property selling price based on 4 property variables.*

Size	*Bedrooms*	*Property Type*	*City*	*Target: Price*
250	*12*	*House*	*Paris*	*$5MM*
80	*2*	*Apartment*	*Lyon*	*$0.5MM*
…	*…*	*…*	*…*	*…*
45	*1*	*Apartment*	*Lyon*	***?***

*Example of ML over structured data: fictional **classification** problem, where we try to predict if a shipping order will be delivered on time or late*

Shipping carrier	*Day of expedition*	*Item weight*	*Historical delay rate*	*Distance to recipient*	*Target: Is late?*
Express	*Monday*	*0.3 Kg*	*0.5%*	*8 Km*	*No*
ThePost	*Monday*	*0.8 Kg*	*30%*	*23 Km*	*Yes*
…	*…*	*…*	*…*	*…*	*…*
ThePost	*Friday*	*5 Kg*	*30%*	*35 Km*	***?***

1. **Linear regression** and its classification counterpart the **logistic regression**. It is the simplest model - a weighted sum of its inputs – taught in high school in France. I like to use this model to illustrate that AI has been in our lives for decades and that many of us learned AI as kids without noticing. Thanks to its simplicity the linear model has three high-valued benefits: (1) it is fast, (2) its predictions are easy to explain, (3) it is easy to port across environments. Consequently, it is popular for high-stakes situations that require speed, transparency, reliability and good economics at scale. Its main drawback is its low expressive power, that limits it ability to model complex datasets. To compensate for this weakness, developers spend a significant amount of time – usually much more than the time spent developing the model - manually crafting high-quality datasets to facilitate the job of the model.
    - ★ How to use it? Linear models are often used in Python via the `statsmodels` library and via Scikit-learn's `LinearRegression` and `LogisticRegression`. Since 2017 the ML platform Amazon SageMaker library features a *Linear Learner* container that enables extreme-scale training of linear models with minibatch Stochastic Gradient Descent (SGD) on clusters of GPUs.
    - ★ Examples:
        - In 2017, the food delivery app **DoorDash** mentioned using a logistic regression to estimate the probabilities of consumer ordering from a given store. Those scores then power real-time search ranking. (*Powering Search & Recommendations at DoorDash*, Manasawala & Koch)

2. **XGBoost** (*XGBoost: A Scalable Tree Boosting System*, Chen et Guestrin) is arguably the most popular algorithm for tabular data. In 2015 alone, XGBoost was used by 59% of Kaggle data science competition winners and by every winning entry in the top-10 of KDDCup. The winners of KDD 2016 and ACM Recsys 2017 also used XGBoost. XGBoost is a *boosting* algorithm, meaning that it is a sequence of algorithms where each algorithm learns to correct the error of the precedent. It has been co-invented by Tianqi Chen and developed by the DMLC community, a productive pair that also produced Apache MXNet and the TVM deep learning compiler. XGBoost also runs on GPU, where it can diminish training costs by up to 98% according to a benchmark from Capital One engineers (*Dask & RAPIDS: The Next Big Thing for Data Science &*

ML, McEntee and McCarty). Also based on gradient boosting, Catboost (created by Yandex researchers Prokorenkova et al) and LightGBM (Ke et al.) see their popularity growing among ML practitioners.

- ★ How to use it? XGBoost is implemented it the eponymous `xgboost` Python library and as a container in Amazon SageMaker.
- ★ Examples:
  - At AWS re:Invent 2019, the gaming company **Rovio** (authors of the *Angry Birds* blockbuster) mentioned using XGBoost to predict game level difficulty (*How Rovio teaches Angry Birds to fly in the cloud using ML*)
  - The French insurer **AXA** (rank 46th on Fortune Global 500 2019) mentions using LightGBM gradient boosting for financial product pricing (*Predictive Underwriting in Commercial Lines*)
  - The French unicorn ride-sharing marketplace startup **BlaBlaCar** mentions "mainly using XGBoost" to build their models (*Thinking Before Building: XGBoost Parallelization*, Romain)
  - **Formula 1** uses XGBoost in SageMaker to run real-time race outcome predictions (*Accelerating innovation: How serverless machine learning on AWS powers F1 Insights*, Luuk Figdor and Andrey Syschikov)

3. **Random Forest** (*Random Forest*, Leo Breiman). Data scientists like the random forest because it is easy to parallelize and can model non-linearities well. The random forest uses *bagging*: it is an average of multiple decision trees each fit on a random fraction of the rows and the columns of the initial dataset. With the advent of above-mentioned recent gradient boosting methods, it is less and less seen in the field nonetheless.
   - ★ How to use it? The random forest is present in Scikit-learn as a classifier (`sklearn.ensemble.RandomForestClassifier`) and as a regressor (`sklearn.ensemble.RandomForestRegressor`). It is pleasant to use and its default hyperparameters generally give good baseline results.
   - ★ Examples: At AWS re:Invent 2018, the gaming company **Rovio** (authors of the *Angry Birds* blockbuster) mentioned using Random Forests in two projects: (1)

to predict user lifetime value and (2) predict game install likelihood. (*How Rovio Uses ML to Acquire, Retain, and Monetize Users*)

A.1.2 Computer vision

Computer vision is developing fast, driven by image-intensive use-cases like self-driving, aerial imagery analysis, healthcare and video protection. Classification, detection and segmentation are important tools that must be in the ML practitioner's toolbox.

4. **Image Classification** consists of associating a class to an image, for example recognizing the brand of a car, the category of a product, if a cart is empty or if a person wears sunglasses. The most popular model for classification is **ResNet** (*Deep Residual Learning for Image Recognition*, He et al). ResNet has implementations written in most DL frameworks and scales well: while He et al took 29h to train their ResNet (written in Caffe) on a P100 GPU to a 75.3% accuracy on the ImageNet dataset, Yamazaki et al were able to train their MXNet-written ResNet to 75.08% accuracy in 76s over a cluster of 2,048 V100 GPUs (*Yet Another Accelerated SGD: ResNet-50 Trainingon ImageNet in 74.7 seconds*, Yamazaki et al)
    - ★ How to use it? To get started with efficient yet accessible implementations of ResNet I recommend the use of Amazon SageMaker Image Classification container (proprietary) or `gluoncv` (open-source).
    - ★ Example: **Airbnb** trained a TensorFlow ResNet-50 model on an Amazon EC2 P2.8xl GPU instance to categorize listing photos by room type such as bathroom, kitchen, etc. (*Categorizing Listing Photos at Airbnb*, Shijing Yao)

5. **Video Classification** consists of associating a class to a video, for example recognizing the action performed in a video or the type of motion done by an object. I rarely encountered it in real-world use-case: analyzing individual frames in a video is often enough to understand the content of the video. For example, classifying if a video is shot indoor or outdoor does not require analysis of motion. Knowledge of motion sometimes matters, for example to estimate the speed of a car or differentiate a hug from a fight. The **Two-Stream Inflated 3D Convnet (I3D)** (*Quo Vadis, Action Recognition? A New Model and the Kinetics Dataset*, Carreira et Zisserman) is a video

classification model providing a good balance between inference speed and accuracy, according to the gluoncv model zoo.

- ★ How to use it? The open-source package GluonCV implements I3D models
- ★ Example: in the AWS ML Blog *Predicting soccer goals in near real time using computer vision*, Zhen Liu et al. present an I3D video classification model predicting soccer goals in near real time. Sports data leader Sportradar experiments this approach to score the intensity of soccer games.

6. **Object Detection** consists of drawing bounding boxes around instances of classes of interest. It is a useful model to count instances of objects in images and to crop areas of interest. **SSD** (*SSD: Single-Shot Multibox Detector*, Liu et al) offers a good balance between speed and accuracy and is available in Amazon SageMaker Object Detector container and in the open-source Python library `gluoncv`. Instead of the SSD neural network architecture you can also consider **FasterRCNN** (Ren et al) if your use case prioritizes accuracy and **YoloV3 (**Redmon et Farhadi) if speed matters more.
   - ★ Example: **carsales.com.au**, Australia's number one car sales website mentions using SageMaker SSD Object Detector to detect license plates on car images for catalog quality automation. (*How we used "Mandarins" to Enhance AWS Sagemaker Object Detection*, Yuxuan Lin). In the AWS ML blog post *Helmet detection error analysis in football videos using Amazon SageMaker* (Huddleston et Ghosh) National Football League presents a PyTorch FasterRCNN object detection model trained to analyze player helmet trajectories.

7. **Semantic Segmentation** consists of classifying each individual pixel in an image. Typical use-case include image background detection and area measurement. **DeeplabV3** (*Rethinking Atrous Convolutions for Semantic Image Segmentation*, Chen et al) performs strongly in benchmarks.
   - ★ How to use it? Top segmentation algorithms DeeplabV3, PSPNet and FCN are available in both Amazon SageMaker and `gluoncv`.
   - ★ Examples: **Eagleview**, a geospatial software and analytics company, describes in the AWS ML Blog *Using deep learning on AWS to lower property damage losses from natural disasters* a segmentation model written in the Apache

MXNet framework estimating the natural disaster property damage from aerial imagery.

Going further: Several advanced image analysis tasks have their own model architectures, for example:

- *Metric learning*, sometimes called *embedding learning*, is a learning paradigm in which the model does not learn labels but instead learns a dimensionality reduction transformation in which instances that have the same label are geometrically close to one another. Learning a transformation instead of a label has many advantages: (1) the model is scalable – its size does not depend on the number of labels, (2) the model can do inference on classes unseen at training (a problem called *open-set recognition*), (3) the prediction consists of finding nearest neighbors of the query record, which nearest-neighbor search (k-NN) libraries do fast. (see *A.1.5* below). Because of those 3 properties metric learning algorithms are popular for identification tasks with large and changing label space, for example logo recognition or facial identification. ArcFace is a well-known facial recognition model (*ArcFace: Additive Angular Margin Loss for Deep Face Recognition*, Deng et al). The Open Neural Network eXchange (ONNX) hosts a pre-trained Arface model.

- *Depth perception* consists of learning the depth of every pixel in an image. Depth models can be monocular and binocular, and supervised or unsupervised. A promising application of depth perception models is to replace expensive depth sensors in autonomous driving. The open-source computer vision library GluonCV has several depth perception tutorials and pre-trained models.

- *Super-resolution* models learn to increase the resolution of images or videos. Super-resolution has promising applications in video traffic bandwidth reduction and image compression.

- *Visual change detection* models detect change across pairs of images while being insensitive to irrelevant background change.

- *Crowd counting* models count people in pictures of human crowds, with possibly high occlusion and scale variation.

A.1.3 Natural Language Processing

8. **fastText** and **Amazon BlazingText classifiers** should be your first choices on text classification problems: they reach good modeling ability by combining n-grams and embeddings. fastText is an open-source library created by Facebook for both token embedding and text classification, and BlazingText is a GPU-compatible, optimized re-implementation of fastText by Amazon. They are shallow neural networks that train quickly (Training on the 400k-sentence DBPedia dataset in a minute)
    - ★ How to use them? `fastText` is an easy-to-use and well documented Python library. The NLP Python library `gensim` also has a fastText implementation (`FastText`). BlazingText is available as a proprietary Docker container in the Amazon SageMaker ML platform. BlazingText-trained artifacts can be loaded off-cloud with open-source fastText readers.
    - ★ Example: **Ibotta**, a technology company providing cash back experiences via a mobile app, describes in the AWS ML Blog post *Powering a search engine with Amazon SageMaker* (Evan Harris) an ML-powered search system. In this system, SageMaker BlazingText classifies search queries in real time to expand query metadata and enhance content retrieval. Another interesting BlazingText use is by the active-learning data annotation Amazon SageMaker Ground Truth, which uses BlazingText to learn to self-label text annotations.

9. **BERT** and its variants (RoBERTa, ALBERT, DistillBERT) are the state-of-the-art for text classification and several other text NLP tasks. Use a BERT variant If your priority is accuracy.
    - ★ How to use it? BERT models can easily be used thanks to open-source NLP libraries `Gluonnlp` and Hugging Face `transformers`.
    - ★ Example: According to Google, BERT-like models power both Google search ranking and Google featured search (*Understanding searches better than ever before*, Pandu Nayak). Microsoft Bing also indicates using BERT models for search (*Bing delivers its largest improvement in search experience using Azure GPUs*, Jeffrey Zhu)

A.1.4 Representation learning

10. **Word2vec** is among the most popular neural networks, along with previously-mentioned ResNet (computer vision) and BERT (NLP). Initially invented in the context of natural language, Word2vec name unfortunately misleads people to think it NLP-specific. It's not! Word2vec works on any natural sequence of tokens, for example walks in graphs and networks, ecommerce carts, movie watchlists or click streams. Word2vec learns numerical representations of tokens that predict neighboring tokens. The representation learned by Word2vec can power recommender systems, anomaly detection, duplicate detection and sequence continuation.
    - ★ How to use it? Word2vec is open-sourced in Python packages `gensim` and `fastText`. SageMaker BlazingText provides a proprietary, scalable and easy-to-use Word2vec container.
    - ★ Examples: Most of the actual use-cases of Word2vec I know are not in NLP, but in token sequence processing for recommendation and representation learning:
        - **Instagram** mentions training Word2vec on sequence of visited accounts to recommend accounts to follow (*Powered by AI: Instagram's Explore recommender system* by Medvedev et Wu).
        - **Anghami**, the popular music streaming service, released a blog post demonstrating Word2vec use for music recommendation (*Using Word2vec for Music Recommendations*, Ramzi Karam)
        - **Tinder** mentioned training Word2vec on sequences of swiped `user_id` to learn user embedding predictive of who they often are swiped with (*Personalized Recommendations at Tinder The TinVec Approach*, Steve Liu)
        - **Yahoo** trains Word2vec in the joint URL-query string space to find the URL best associated to a given search query (*Network-Efficient Distributed Word2vec Training System for Large Vocabularies*, Ordentlich et al)
        - **Uber** trains Word2vec on sequences of cities to learn vector representation of locations and better identify anomalous ride patterns (*Exploring New Machine Learning Models for Account Security*, Evans and Ramasamy)

11. **LDA (Latent Dirichlet Allocation)** is a token sequence representation algorithm that converts variable-length sequences of tokens into fixed-length numerical vectors whose coefficient sum to one. This principle is also called *topic modelling* as the model tries to describe each token sequence as a combination of distinct categories. LDA is useful to transform text into numbers for later numerical processing, for example clustering, similarity search or classification. With the advent of deep-learning based NLP, LDA is less and less commonly seen for pure NLP tasks. Like Word2vec, LDA can creatively be used on non-text token sequences.
    - ★ How to use it? LDA is easy to use via its implementation in Amazon SageMaker LDA and Amazon Comprehend Topic Modelling.
    - ★ Example: AT&T security researchers apply LDA to *source-destination* IP traffic network data in order to detect peer-to-peer communities. (*Gangs of the Internet: Towards Automatic Discovery of Peer-to-Peer Communities*, Li et al)

12. **Matrix Factorization** and its neural network variants are the de-facto solutions to learn to score affinity between two types of objects, for example:
    - *users* and *items* in a recommender system problem
    - *seller* and *buyer* in a marketplace modelling problem
    - *client* and *server* in a networking problem
    - *candidate* and *job* in a recruitment context
    - *website* and *ad* in an advertising context

    Matrix factorization learns the affinity between two objects by (1) learning intermediary vector representations of each object and (2) using those representations to form an interaction score, typically via Euclidean distance. A lovely feature of the matrix factorization and its deep derivatives is that it can be used in 2 ways:
    a. As a supervised predictor estimating the interaction between objects A and B
    b. As a representation learning model that learns dense numerical representation of objects A and B. Those representations can then be used to compare, cluster, find similarities and anomalies in each collection of objects.

The standard matrix factorization model consists of a decomposition of the interaction matrix between A and B into the product of two matrixes representing objects A and B in a low-dimension space.

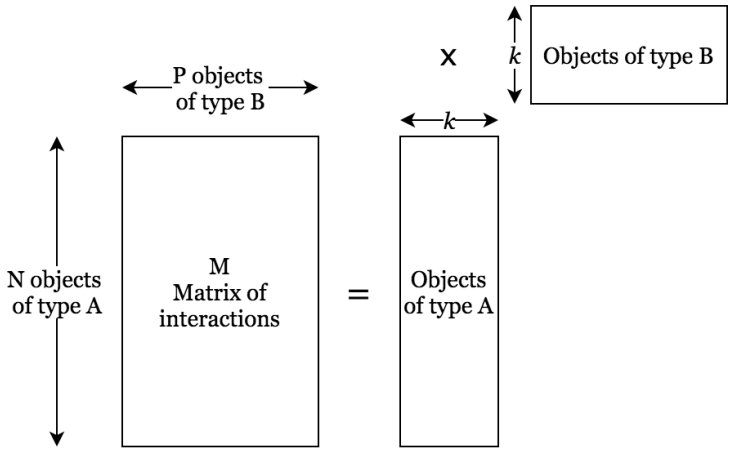

Figure 29. Standard matrix factorization

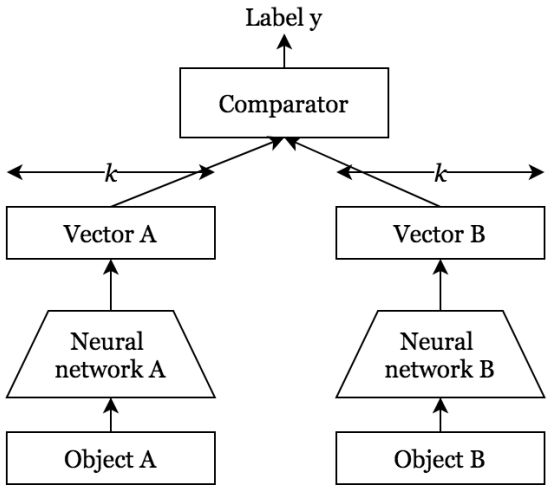

Figure 30. Matrix factorization with deep neural networks.
When the same model is used for arms A and B this is called a siamese network.

★ How to use it?

- Standard matrix factorization can be used in Scikit-learn (featuring a variant learning positive weights, `sklearn.decomposition.NMF`). The big data framework Apache Spark features a popular distributed matrix factorization model trained with Alternating Least Squares (`ALS`)

in its `spark.ml` library. Matrix factorization is such a prominent approach in recommender systems that several Python libraries were developed to improve and simplify it, such as `surprise`, `implicit` and `lightfm`.

- Deep-learning based matrix factorization is simple to implement with the Object2vec container of Amazon SageMaker, that works both on tokens and sequences of tokens. In the open-source it is surprisingly less accessible and, in my knowledge, has no well-adopted simple implementation. Re-implementing it from scratch with modern imperative frameworks such as PyTorch or MXNet takes few lines of code and is a good introductory practice to deep learning. Apache MXNet codebase contains an excellent deep matrix factorization tutorial that implements embedding arms with skip-neural connections, a technique presumably inspired from computer vision that I never saw anywhere else. I recommend the reader to view the associated 1-hour AWS re:Invent 2016 session on recommenders delivered by Leo Dirac (*Using MXNet for Recommendation Modeling at Scale*), which I find to be the best introduction to deep matrix factorization.

★ Examples: the most popular use-case of matrix factorization is undeniably recommender systems. For example:

- **Spotify** mentions using matrix factorization over 40MM users and 20MM songs to power various music recommendation features (*Logistic Matrix Factorization for Implicit Feedback Data*, Johnson et al)

A.1.5 kNN

13. **K-Nearest-neighbor (kNN)** search consists of finding the *k* most similar vectors to a query vector. The kNN has two possible uses:
    a. As a supervised algorithm: in this design, the label of the neighbors is used to infer query labels.
    b. As a search algorithm: to return the most similar vectors to the query vector.

The advent of DL made kNN immensely popular, because deep neural networks generate vector representations that need to be indexed, searched and compared.

Theoretically, kNN approaches do not require any training since distances can be computed one by one when need be. In practice it is common to pre-compute a file, called an *index,* that stores relevant pre-computations to make the later searches faster.

- ★ How to use it? Several dedicated libraries have been developed for low-latency real-time kNN search, most of them searching for approximate neighbors instead of exact neighbors. Three of them are particularly popular:
  - The Spotify-developed `annoy` works by storing records in clusters formed recursively N times (forming the N *trees* required as an input to the algorithm)
  - The Facebook-developed `faiss` can run on GPUs. FAISS is specifically known for its methods based on quantization of vectors. (*Billion-scale similarity search with GPUs*, Johnson et al)
  - `hnswlib` pre-computes a number of graphs to facilitate search at query time (*Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs*, Malkov et Yashunin). HNSW is considered state-of-the-art and performs better than Annoy and FAISS on a number of benchmarks (https://github.com/erikbern/ann-benchmarks)
  - Amazon Elasticsearch and Open Distro for Elasticsearch have a distributed kNN search powered by HNSW. At the time of this writing, this is the only production-ready distributed kNN index I am aware of. Though Annoy, FAISS and HNSW are said to be designed for scale, they produce single-file artifacts and do not feature native model-parallel implementation, making them impractical to use on their own when artifacts exceed several dozen GB size.
  - Don't force yourself to use an approximate kNN library. Several problems have scale and latency objectives that exhaustive search answers well. For example, computing the exact nearest neighbors in an array of 10MM 64-dimensional vectors take less than 20ms on GPU with algebra routines found in Apache MXNet `NDArray`
- ★ Examples:
  - Annoy k-NN documentation mentions use at Spotify for music recommendation

- In 2019 Amazon presented a new deep-learning based search system that transforms query and document in a joint embedding space and performs inference with k-NN search (*Semantic Product Search*, Nigam et al)
- Instagram uses FAISS to lookup relevant accounts to recommend in its word2vec-based account embedding recommend system (*Powered by AI: Instagram's Explore recommender system*, Medvedev et al.)

A.1.6 Clustering

Clustering algorithms are useful to (1) transform numerical data into categorical data (for example to quantize the weights of a neural network or the coefficients of a vector), (2) to arrange multi-dimensional data in buckets and (3) to find isolated points. Customer segmentation is a use-case frequently seen in theorical textbook examples but much less in the field in production. Two clustering techniques are particularly prominent:

14. **KMeans** is the most well-known clustering algorithm. It is intuitive: it tries to find N clusters and their centers, called *centroids,* in the dataset so that the distance from the points to their cluster centroid is the smallest possible.
    - ★ How to use it? The open-source ML library Scikit-learn has a KMeans implementation (https://scikit-learn.org/stable/modules/clustering.html#k-means). Amazon SageMaker has a scalable and efficient implementation of KMeans that was benchmarked an order of magnitude faster and cheaper to train than Spark MLLib on the 372Gb Gdelt dataset (*Elastic Machine Learning Algorithms in Amazon SageMaker*, Liberty et al).
    - ★ Examples
        - KMeans can be used to reduce the number of colors necessary to represent an image (scikit-learn.org)
        - KMeans is used by Redmon et Farhadi to find prior bounding box positions of YoloV3 detection model representative of the dataset. (*YoloV3: an Incremental Improvement*, Redmon et Farhadi)

15. **DBSCAN** is another popular clustering algorithm. Three notable differences of DBSCAN compared to KMeans are that (1) DBSCAN finds the number of clusters itself while KMeans needs it as an input, (2) DBSCAN is based on point-to-point distances while KMeans is based point-to-centroid distance, making DBSCAN able to cluster much more varied point cloud shapes than KMeans, (3) DBSCAN can leave points unclustered and label them as "outliers", while KMeans tries to cluster every points. Consequently, DBSCAN is often seen as an outlier detection model as well.
    - ★ How to use it? DBSCAN is implemented in Scikit-Learn (`sklearn.cluster.DBSCAN`)
    - ★ Example: The ride-hailing company Uber mentions using DBSCAN on IP traffic representations to identify anomalies and detect fraudulent behavior (*Exploring New Machine Learning Models for Account Security*, Evans and Ramasamy). DBSCAN is also used over 3D points clouds by Soonmin Hwang et al to identify regions of interest in a point cloud for 3D object detection in fused LIDAR and camera data. (*Fast Multiple Objects Detection and Tracking Fusing Color Camera and 3D LIDAR for Intelligent Vehicles*, Hwang et al.)

A.1.7 Unsupervised Anomaly detection

16. **Random Cut Forest** (RCF) detects anomalies by doing recursive random splits in the data point cloud and measuring how early a point is isolated by a split. The measure of isolation is made robust by the use of bootstrapping (the suite of splits is done on N partitions of the data) and by selecting axis to cut proportional to their variance.
    - ★ How to use it? AWS has two RCF implementations. In the Amazon Kinesis Analytics service RCF is applied to data streams, while the Docker-based proprietary RCF that is part of the SageMaker Built-in algorithms library supports batch training and deployment over cloud HTTP endpoints. In the open-source, the `rrcf` Python implementation of RCF has notable traction. (https://github.com/kLabUM/rrcf)
    - ★ Example: At Big Data Paris 2019, the French premium TV channel **Canal+** and its technology partner MFGLabs mentioned using Robust Random Cut Forest algorithm (*Robust Random Cut Forest Based Anomaly Detection On Streams*,

Guha et al) in order to detect streaming consumption anomalies (*Anomaly Detection on Streaming Data*)

17. **Autoencoders** are neural networks trained to compress their input into a lower-dimension representation and decompress it with maximum fidelity. Autoencoders are popular for dimensionality reduction. For example, Amazon researchers Ran Ding, Ramesh Nallapati and Bing Xiang apply variants of autoencoders to text to learn compact vector representations of documents. (*Coherence-Aware Topic Modelling*, Ding et al). The ability of the autoencoder to learn to compress and reproduce a given pattern depends on the pattern frequency in the dataset: rare patterns will not drive sufficient learnings and cause reconstruction errors. Therefore, autoencoders are good tools for unsupervised anomaly detection: reconstruction error scores how surprised the model was to see a given input combination.
    - ★ How to use-it? A distributed Variational Auto-Encoder (VAE) Docker container for bag-of-token sequences can be used in Amazon SageMaker Neural Topic Modelling (NTM) container. I am not aware of a popular, production-grade autoencoder abstraction in the open-source and this type of network is usually rewritten from scratch.

A.1.8 Time series forecasting

Time series forecasting consists of predicting the continuum of values a time series will take in its future iterations. A time series forecasting problem could either involve just one time-series to predict or several thousands of related timeseries. Examples of time series forecasting include:

- Predicting the energy consumption of households in a city for capacity planning
- Predicting the traffic curve of web applications for auto-scaling
- Predicting the demand of all products of an ecommerce catalog for stock management

For example, the cloud data warehouse service Amazon Redshift internally uses a time series forecasting algorithm to right-size in warm instance pool (*Automating your Amazon Forecast workflow with Lambda, Step Functions, and CloudWatch Events rule,* Ma et al.)

18. **DeepAR** is a recurrent neural network-based, use-case agnostic time-series forecasting model that was published by Amazon researchers Salinas et al. in 2017 in their paper *DeepAR: Probabilistic Forecasting with Autoregressive Recurrent Networks*. DeepAR outperformed most relevant baselines and popular multi-dimensional timeseries benchmark datasets. DeepAR has several benefits over traditional time-series forecasting approaches:
    - It is autoregressive yet it can also learn from categorical and numerical covariates
    - It returns probabilistic predictions (quantiles) and can handle several likelihood functions modeling real-life situations
    - It handles varying scales well (in retail it is common to see orders of magnitude differences in activity volumes)
    - It requires little data engineering. Calendar and lagged features are created by the model.
    - How to use it: An implementation of DeepAR is available in the open-source package `gluonts`. It also exists as a managed Docker container in Amazon SageMaker and as a managed service, with extra features such as ensembling and learning rate scheduling, in Amazon Forecast.

    ★ Examples:
        - The media group Axel Springer mentions using SageMaker DeepAR to forecast newspaper sales (*Predicting Newspaper Sales with Amazon SageMaker DeepAR*, Justin Neumann)
        - In *Resilient Neural Forecasting Systems*, Bohlke-Schneider et al. Describe the application of DeepAR forecasting for labor attendance prediction in the context of Amazon Fulfillment Center Network labor planning
        - The French water services leader Veolia described in 2020 a method to assist maintenance of water filtering membranes using the DeepAR algorithm (*Using Machine Learning for Water Filtering Membranes Maintenance*)

A.1.9 Bandits

All the previously-mentioned algorithms are trained offline in batch, then deployed as a static artifact. They suffer from couple weaknesses: (1) they require availability of historical data and (2) they do not improve nor learn additional knowledge unless retrained. Multi-arm and contextual bandits are algorithms learning from interaction and directly changing their weights anytime they receive the feedback from a past prediction. They excel in situations where an ML system interacts with its environment and influence the data generation process.

- *Multi-arm bandits* learn to choose the best option ("arm") in a list of options. One can see them as a fast-changing, dynamic A/B tests where arm weights are permanently changed based on action feedback
- *Contextual bandits* are multi-arm bandits assisting their decision with extra context features

19. **Thompson Sampling (TS)** is a popular option selection strategy for both multi-arm and contextual bandits. Thompson Sampling creates exploration by avoiding to always select the top-predicted option, and by instead sampling among all possible options with a sampling probability proportional to their chance of being the best. Consequently, TS almost always makes relevant choices, yet the randomness induced by the sampling strategy means it will give its chance to less promising options from time to time for data acquisition.
    - ★ How to use it: Thompson Sampling is conceptually simple and easy to re-implement from scratch
    - ★ Examples: Amazon mentioned using a Thompson Sampling contextual bandit to learn an optimal design of the Prime membership call-to-action page (*An Efficient Bandit Algorithm for Realtime Multivariate Optimization*, Hill et al)

Appendix A2 - Acronyms

Computer science is an acronym-heavy industry. Below are several acronyms that the reader may cross repeatedly throughout the ML journey

ABCI: AI Bridging Cloud Infrastructure. A supercomputer located in Japan that hosted benchmark-beating deep learning trainings.
API: Application Programming Interface. Code interface used to interact with a remote service or a local library.
AVX: Advanced Vector Extensions. Extensions to the instruction set of Intel CPU that improve its parallel computing abilities.
AWS: Amazon Web Services. A pioneering cloud services company. The employer of the author at the time of this writing.
BERT: Bidirectional Embedding Representation from Transformers. A popular natural language processing model architecture. Proposed in *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*, Devlin et al.
CICD: Continuous Integration, Continuous Delivery. Software development practice consisting of tools and processes facilitating the transformation of source code into customer-facing products.
CPU: Central Processing Unit. The chief chip in a computer system. The CPU orchestrates tasks and can run computations.
CSV: Comma-Separated Values. A popular file format to store tabular data.
CTR: Click-Through-Rate. The ratio of clicks divided by impressions of a digital asset. CTR is a frequent target metric for digital advertisers.
CUDA: Compute Unified Device Architecture. A programming framework created by NVIDIA to write generic programs for the GPU.
DALI: Data Loading Library, an NVIDIA library to facilitate data loading and transformation on NVIDIA GPUs.
DBSCAN: Density-Based Spatial Clustering of Applications with Noise. A popular clustering technique. By Ester et al. in *A density-based algorithm for discovering clusters in large spatial databases with noise*.

DL: Deep Learning, a variety of algorithm consisting of several layers of neural networks, giving them enough learning power so that they can be applied directly on low-level data such as text, images or speech.

EIA: Elastic Inference Accelerator, a fractioned GPU service for inference invented by AWS.

ETL: Extract, Transform and Load. Processes, usually scripted, that transfer or transform data.

FMA: Fused Multiply Add. A matrix multiplication followed by an addition.

FPGA: Field Programmable Gate Array. An integrated circuit that can be configured programmatically.

GCP: Google Cloud Platform, the cloud offering of Google.

GDPR: General Data Protection Regulation. A data protection compliance framework protecting EU citizens by governing the use of their data.

GPU: Graphical Processing Unit. Computing cards initially designed for graphical workloads that have high parallelism ability for algebra.

GTC: GPU Technology Conference, an annual event run by NVIDIA.

HIPAA: Health Insurance Portability and Accountability Act. A Federal US law governing the management of health data.

HPC: High-Performance Computing. Domain of computer science focusing on large-scale, compute intensive tasks and their optimization.

IDE: Integrated development environment. Application enabling developers to build software.

IPU: Intelligence Processing Unit. An AI chip designed by the firm Graphcore.

KNN: k-Nearest Neighbor. An ML question task consisting in finding the K records of a dataset most similar to a query record.

LAMB: Layer-wise Adaptive Moments optimizer for Batch training. A large-scale NLP training optimization technique invented by researchers You et al.

LARS: Layer-wise Adaptive Rate Scaling. A large-scale computer vision training optimization technique invented by researchers You et al.

LIDAR: LIght Detection And Ranging is a depth perception technique based on reflection of laser beams.

LSTM: Long Short-Term Memory. A neural network layer capable of modelling sequences. *Long Short-Term Memory*, Hochreiter et Schmidhuber.

MMS: Multi-Model Server. An open-source model serving stack created by AWS.

MPI: Message Passing Interface. A communication framework to write parallel programs over cluster of machines.

NCCL: NVIDIA Collective Communication Library. A multi-machine communication library developed by NVIDIA.

NLP: Natural Language Processing. Automated treatment of natural language.

ONNX: Open Neural Network eXchange. An open consortium and associated open-source tools aiming at cross-framework portability.

PCI-DSS: Payment Card Industry Data Security Standard. A compliance framework governing the handling of payment card data.

PS: Parameter Server or Parameter Store. A distributed model training method where model parameters are shared across machines via a dedicated store.

RAM: Random Access Memory. The short-term, fast-access memory of a computer, can be used only when the computer in on.

RDMA: Remote Direct Memory Access. Technology enabling data transfer across machines from memory to memory.

RNN: Recurrent Neural Network. A neural network paradigm to model sequences.

SDK: Software Development Kit. A code development framework to build applications on a given platform.

SGD: Stochastic Gradient Descent. The main neural network training method, that consists of iteratively applying small variations to model weights in the direction that goes down the error slope.

SIMD: Single Instruction Multiple Data. A parallelism design where the same operation is applied in parallel to multiple data.

SNPE: Snapdragon Neural Processing Engine. Qualcomm runtime for neural network inference.

SRAM: Static Random-Access Memory. A type of memory that is faster than the traditional DRAM used in main memory, used for on-chip memory in the Graphcore IPU.

SSD (1): Solid-State Drive. Fast storage device using Flash-based memory instead of rotating disks.

SSD (2): Single-Shot Detector. A popular computer vision model architecture for object detection. *Single-Shot Multibox Detector*, Liu et al.

SVAIL: Silicon Valley AI Lab, a research laboratory of Baidu.

TDP: Thermal Design Power. Measure of heat dissipated by a computer system.

TPC: Tensor Processing Core, the elementary compute block in the Habana Gaudi chip.

TPU: Tensor Processing Unit. An AI chip invented by Google.
TS: Thompson Sampling. An arm selection strategy for multi-arm bandit and contextual bandit algorithms.
VAE: Variational Auto-Encoder. A deep learning model trained to compress objects by learning to compress and decompress from a noisy version.
VCS: Version Control System. A system to manage the versioning of software code.
VM: Virtual Machine. A distant virtual computer, oftentimes itself a fraction of a bigger physical computer.

Appendix A3 – Reading List

The following resources help becoming an efficient ML practitioner:

Systems engineering

- *The Amazon Builder Library*.
- *Building Secure and Reliable Systems: Best Practices for Designing, Implementing, and Maintaining Systems* - Heather Adkins, Betsy Beyer, Paul Blankinship, Piotr Lewandowski, Ana Oprea and Adam Stubblefield.

ML coding

- *Python Data Science Handbook* - Jake VanderPlas.
- *Deep Learning with Python* - François Chollet.
- *Deep Learning for Coders* - Jeremy Howard and Sylvain Gugger.

ML science

- *Dive into Deep Learning* - Zhang et al.
- *The Elements of Statistical Learning: Data Mining, Inference, and Prediction* - Trevor Hastie, Robert Tibshirani, Jerome Friedman.
- *Pattern Recognition and Machine Learning* - Christopher M. Bishop.

ML on AWS

- *Data Science on AWS: Implementing End-to-End, Continuous AI and Machine Learning Pipelines* - Antje Barth and Chris Fregly.
- *Learn Amazon SageMaker* - Julien Simon.
- *AWS Well-Architected Framework*.

Causal Inference

- *Causal Inference in Statistics: A Primer,* by Pearl et al.
- *Causal Inference for Statistics, Social, and Biomedical Sciences*, Imbens et Rubin.

Communication

- *The Elements of Style* - William Strunk Jr. Not an ML book, but a must-read on written English, that will help the ML practitioner write effectively.

Appendix A4 – Notable AWS services for ML

I encourage the reader to be familiar with the following AWS services. Note that those are not necessarily ML services, but rather services often seen within a production ML project.

1. Amazon S3, to store datasets and model artifacts.
2. Amazon Elastic Map Reduce (EMR) to create distributed computing clusters. EMR is commonly used with Spark.
3. Amazon DynamoDB is a scalable, managed key-value store database. DynamoDB can be useful in ML use-cases to store features, to cache inference results, or also to build metadata stores.
4. Amazon Athena for managed SQL queries over S3 data with pay-per-volume pricing.
5. AWS CodeCommit for code versioning.
6. SageMaker Ground Truth for data annotation.
7. SageMaker Notebook Instances to create managed, git-connected Jupyter notebooks.

8. SageMaker Training and Tuning, to create custom remote training jobs on ephemeral infrastructure.
9. SageMaker Hosting, to deploy models on low-latency, real-time REST endpoints.
10. SageMaker Neo to compile models for low-latency, low-memory footprint inference.
11. AWS Lambda, a service to create micro-virtual machines that last up to 15min and pack up to 10G of RAM (as of May 2021). Lambda is often seen in machine learning at multiple places:
    a. To run scheduled API calls, for example to launch a model training or to shut down resources like endpoints and notebook instances.
    b. To call model endpoints and run additional steps such as feature collection, pre-processing and logging.
    c. To deploy a model. Numerous are the models and frameworks that fit within Lambda constraints. Deployment economics of AWS Lambda are excellent for small-to-medium scale ML endpoints.
    d. More rarely, to run pre-processing or training tasks.

Appendix A5 – 20 Important Python libraries for ML

By decreasing order of importance

Native Python modules

1. `multiprocessing` in particular `Pool.map` and `Pool.starmap`, that can parallelize the execution of a function over multiple inputs and harness all the power of multi-CPU hardware.
2. `os` for common discussions with the OS, such as `os.listdir()` .
3. `subprocess`, for example, to launch command line instructions from Python.
4. `collections`, in particular `Counter`, to do word or token counts.
5. `functools`, which contains the `reduce` function in Python 3.
6. `itertools`, in particular `compress` and `product`.

Third party modules

7. `pandas`, the standard library to manipulate tabular data.
8. `numpy`, the standard library for computations on vectors and matrices.
9. `scikit-learn`, the library for machine learning on structured data.
10. `seaborn`, a library for data visualization.
11. `gluoncv`, a library for computer vision built on top of MXNet.
12. Hugging Face `transformers`, a library for natural language processing that is both intuitive and state-of-the-art from a science standpoint.
13. `mxnet.ndarray`, an MXNet library for asynchronous, GPU-compatible imperative tensor computation. Can sometimes be used as a substitute of `numpy` for faster computations.
14. `boto3`, a Python module to manipulate the services of the AWS cloud.
15. `nltk`, in particular for its text cleaning abilities.
16. `pillow` to read and manipulate images.
17. `pyarrow`, in particular to read parquet files into pandas.
18. `cv2`, the Python interface of the computer vision library OpenCV, convenient for computer vision and associated manipulations, for example parsing a video into a `numpy` array.
19. `networkx`, a package to analyze networks and graphs.
20. `bokeh` for interactive data visualization. For example, useful to create visualization of 2d-PCA of embeddings of sentences, and plot sentence interactively with a hover action. The hover action allows to handle long text labels that would otherwise be difficult to visualize with a static visualization.

Appendix A6 – GitHub popularity of ML libraries

I found GitHub stars to represent rather well the popularity of libraries. I recommend to run extra due diligence on quality and stability of open-source libraries that have less than a thousand stars. Low GitHub traction could signal low maintenance and difficulties to troubleshoot. Note that tools for exploration and training (orange background) are more popular than tools for deployment (green background), reflecting my message throughout the book that there is an opportunity for organizations and individuals in developing their knowledge of deployment.

Library	Use-Case	GitHub stars (June 2021, thousands)
TensorFlow	Neural network development	156
Keras	Neural network development	51,3
PyTorch	Neural network development	48,7
Hugging Face Transformers	NLP model development	47
Scikit-Learn	ML on tabular data	46,1
Apache Spark	Data processing	30,1
Pandas	Data processing	30
fastText	NLP model development	22,6
Airflow	Workflow manager	21,7
XGBoost	ML on tabular data	21,1
fastai	Neural network development	21
Apache MXNet (incubating)	Neural network development	19,5
Numpy	Data processing	17,4
ClickHouse	Data warehousing	16,9
Detectron2	Computer vision neural networks	16,8
Ray	Distributed computing	16,2
PaddlePaddle	Neural network development	15,7
Bokeh	Data visualization	15,2
PyTorch Lightning	Neural network development	13,8
Matplotlib	Data visualization	13,7
FAISS	KNN	13,7
LightGBM	ML on tabular data	12,6
Gensim	NLP model development	12,1
Horovod	Distributed SGD	11,3
MLFlow	Experiment management	9,5
Networx	Graph analysis	9,2
Annoy	KNN	8,6
Seaborn	Data visualization	8,5
Dask	Data processing	8,4

DVC	Dataset management	8,1
Vowpal Wabbit	ML on tabular data	7,6
TVM	Inference compiler	6,8
Statsmodels	ML on tabular data	6,4
CatBoost	ML on tabular data	5,9
TensorFlow Serving	Serving	5,1
DeepSpeed	Distributed SGD	5,1
Surprise	Recommender systems	4,9
GluonCV	Computer vision neural networks	4,8
ONNXRutime	Inference compiler	4,7
Optuna	Hyperparameter optimization	4,7
LightFM	Recommender systems	3,6
AutoGluon	ML on tabular data	3,4
SimpleDet	Computer vision neural networks	2,9
BytePS	Distributed SGD	2,8
Glow	Inference compiler	2,5
NMSLIB	KNN	2,4
Implicit	Recommender systems	2,3
MindSpore	Neural network development	2,3
GluonNLP	NLP model development	2,3
Triton Inference Server	Serving	2,3
Scikit-Optimize	Hyperparameter optimization	2,1
GluonTS	Timeseries model development	1,9
TorchServe	Serving	1,8
EconML	Causal Inference	1,4
Clipper	Serving	1,3
Mesh TensorFlow	Distributed SGD	1
Multi Model Server	Serving	0,8
Treelite	Inference compiler	0,5
Neo DLR	Inference runtime	0,4

Appendix A7 – Acknowledgements

- I want to thank the persons who gave me ideas and feedback, in particular book reviewers Romain Vermeulen, Antje Barth, Ro Mullier, Fred Nowak, Hervé Nivon, Loïs Baude, Paul Dermarkar, Matthieu Bouthors, Jean-Baptiste Dugast, Alexandre Deschildre and Bruno Cruchant.
- Eternal thanks to my wife that drives me to take risks and grow.
- Warm thanks to my employer Amazon, that has been supporting me for 7 years.