Machine Learning Systems Complexity

Dieudonne Ouedraogo

8/14/2019

# I-INTRODUCTION

In all domain of the modern world such as economic, political, social, scientific and technological, there is an increased complexity while trying to study, understand or predict the behavior of the systems they belong to; there are an unprecedented dynamics and uncertainty. Interdependencies, feedback loops, interconnections, intractable external factors render those systems highly unpredictable. Even though we can collect extensive data from the system, it is not apparent which features or variable are useful in determining the future behavior of a system. Machine learning armed with sophisticated algorithms had promised to tackle those issues and provide insights, but increasingly predictions generated through machine learning are getting less and less reliable. Even worst machine learning carry at itself system-level complexities that make hard to deal with, understand, and put it into great use.

It is a new very complex world where business and organizations must be equipped with the right tools and approaches to deal with those issues. We made significant advance algorithmically and technologically, and tools, which are exceptional a the parts level, but at the system level, not much effort has been made. Systems thinking can deliver what is missing in machine learning.

Peter Senge(1994) argued that being able to see the forest and the trees is a vital skill in systems thinking.

In this paper, we outline critical steps built on systems thinking to help machine learning researcher, and practitioners reduce complexity by tackling problems in a holistic view rather than as a part.

# II- COMPLEXITY

Physicists Bohr and Heisenberg in their non-deterministic view of the universe reached to the conclusion that the universe cannot be perfectly understood. The view was a sharp departure from Einstein and other determinists who believe for a long time that the universe was exact.

The distinction between complex and deterministic systems is fundamental and has philosophical repercussions.

For centuries eminent philosophers and scientists have believed that the world is deterministic - that it predictably behaves under natural laws and that any uncertainty of outcomes is a result of our lack of knowledge how the world works. In other words, for supporters of determinism, the world is complex only for those who do not understand it. A more plausible alternative view has been put forward recently by Prigogine. He proposes that the world is inherently complex, and it evolves irreversibly with time. Future is not given; it emerges from the interaction of billions of activities performed by constituent agents, including people, animals, plants as well as natural forces such as climate, erosion, volcanic eruptions, and solar spots.

Prigogine’s hypothesis how the world works is an essential part of the Complexity Mindset, or the Complexity Worldview, which consists of the set of assumptions, concepts, principles, and methods, which represents a useful toolset for addressing complex issues. This view is widely echoed and accepted through experiments as our world become a highly interconnected system of entangled systems.

Complexity theory is inherently referencing to systems thinking and systems science so understanding complexity theory could help researchers understand systems behavior holistically.

Complexity is a property of systems that consist of diverse, interacting components, often called Agents. Complexity is characterized by seven key features: connectivity, autonomy, emergence, nonequilibrium, nonlinearity, self-organization, and coevolution (George Rzevski)

## 1. Connectivity

Agents, components, or subsystems are interconnected. The complexity of the system increases with the number of links that connect agents. The strengths of links between agents link also affect the system complexity. Adjusting agent connectivity is an effective method for tuning complexity. Complex systems often consist of regions of high connectivity (and high complexity) interconnected by low-connectivity (and low complexity) links.

## 2. Autonomy

Agents have a certain freedom of behavior (autonomy), which is always limited by norms, rules, regulations, and laws. The increase in the autonomy of agents increases complexity, and if there are no constraints on agents behavior, the system falls from the complex into random behavior. Inversely, if the autonomy of agents is reduced (by tightening of laws and regulations), the system complexity will decrease, and in the extreme, the system will become deterministic. Complex systems have no central control.

## 3. Emergence

The behavior of complex systems emerges from the interactions of agents and is not predictable, and yet it is not random. Emergence, in general, denotes a property of a system that is evident in the system as a whole, but it is not present in any of its components.

## 4. Nonequilibrium

Complex systems are subjected to continuous changes caused by either a succession of discrete disruptive events or by slow, imperceptible drift into failure. Frequency of disruptive events varies with complexity. In systems of high complexity, disruptive events occur so frequently that the system has no time to return to stable equilibrium before the next disruption occurs. When complexity levels are very high, the system is said to be at the edge of chaos because the uncertainty of behavior is close to one

## 5. Nonlinearity

Relations between agents, components, or subsystems are non-linear. Nonlinearity may be amplified, and an insignificant disruptive event could lead to a catastrophic outcome (an extreme event) referred to as the butterfly effect. The butterfly effect increases complexity. In complex systems, outcomes are, as a rule, consequences of numerous interacting causes, and therefore, the cause-effect analysis is inappropriate.

## 6. Self-organization

Complex systems have the proprieties to react to disruptive events by autonomously self-organizing to eliminate or, at least, reducing consequences of the disruption. This property is called adaptation. Self-organization may also be caused by a propensity to improve its performance, the property called creativity or innovation. A system is intelligent if it can initiate and performs adaptive and creative activities. Adaptation, intelligence, and creativity are emergent properties exclusive to complex systems; their levels increase with complexity. Artificial Intelligence (AI) found in sophisticated adaptive software is typically referred to as Emergent Intelligence [6].

## 7. Coevolution

Complex systems change over time as the environments change and, in turn, they affect their environments. Coevolution is irreversible.

# III-Mitigating complexity

Usually, complex systems cannot be controlled in the sense in which deterministic systems are controlled. However, they can be managed.

There are two aspects of managing complexity:

Coping with external complexity from the environment

Mitigating the internal complexity of systems.

## 1. Coping with External Complexity

We rarely have control over our environment, and therefore, we must be prepared to deal with and accept its complexity.

The best strategy for coping with the complex environment is to develop a capacity for adaptation. To be Adaptive means to be capable of achieving desired goals under conditions of the frequent occurrence of unpredictable, disruptive events. Adaptability is achieved by self-organizing in reaction to a disruptive event to eliminate or, at least, reducing consequences of the current disruption before the next one occurs.

## 2. mitigating internal complexity

The level of complexity of systems or organizations, which we design or own, can be mitigated by adjusting the autonomy and connectivity of agents in the system. Rather than just optimizing each part separately, an overview of the whole system can lead to optimal performance. This is mostly a trial-and-error process, informed by experience and a holistic view designing the system.

# IV-Complex Systems in machine learning

With the explosion of the internet where data collection is relatively easy than before, businesses and organization are making use of machine learning systems to handle their operation and drive decision making. Fueled by the growth of open-source software, machine learning systems can be quickly developed and deployed. However, with ease comes unique risks of technical debt in machine learning systems, using the metaphor framed by Ward Cunnigham in 1992 as “technical debt” to describe the long term costs incurred by moving quickly in software engineering, Similar to financial debt, there are often sound strategic reasons to take on technical debt. However, all debt needs must be resolved. Technical debt may be paid down by refactoring code, improving unit tests, deleting dead code, reducing dependencies, tightening APIs, and improving documentation [8]. The goal is not to add new functionality, but to enable future improvements, reduce errors, and improve maintainability. Deferring such payments results in compounding costs. Hidden debt is dangerous because it compounds silently. Unlike traditional software-based system where debt mainly comes from the code level, machine learning systems behavior depend not only on codes but also on data and inherently the external world. The debts coming from the system level can evade traditional testing used in software-based systems framework and can change or affect the behavior of a machine learning system. The goal in this paper is to highlights those system-level issues and to propose systems thinking’s approach to reduce or avoid the complexity.

## 1-Complex pipelines

Modular design and encapsulation are seen as best practices in software engineering, they allow changes and improvements to be implemented efficiently, they also allow solid logical consistency, however in machine learning inputs and outputs could change with the data and the external world, precisely the reason why we use machine learning where the behavior of a systems cannot be adequately expressed in software logic without external data[8].

The real world does not fit into tidy encapsulation. Below are issues encountered in machine learning pipelines.

### a. Input data dependencies.

Machine learning dwells on input data and mix signals from features of the data, entangling them, and making isolated improved challenging. Consider an example of a system that uses features X\_{1},…X\_{n} in a model. If we change the input distribution of values in x1, the importance, weights, or use of the remaining (n − 1)features may all change. Adding a new feature Xn+1 can cause similar changes, as can removing any feature Xj. No features are ever really independent. W describe this as CACE principle: Changing Anything Changes Everything. CACE applies not only to input signals, but also to hyper-parameters, learning settings, sampling methods, convergence thresholds, data selection, and virtually every other possible tweak in machine learning systems.

A mitigation strategy will be to use ensembles; ensembles are more robust to features changes when deployed in machine learning systems. So a slight change in the data could still lead to predictable behavior of the system rather than using single models.

A second strategy is to continue monitoring the predictions outputs in order to detect any change in behavior. One such method was proposed in [12], in which a high-dimensional visualization tool was used to allow researchers to see effects across many dimensions and slicings quickly. Metrics that operate on a slice-by-slice basis may also be handy.

### b. Correction models in cascade

In some situations a model MA exist for a problem A and we need a solution to a slightly different problem A’, and a model MA’ is built using MA as input and small correction is done to solve the problem. However, this correction model has generated a new system dependency on MA, making it expensive and risky to analyze and maintain the new model in the future. The cost and risk increase as the number of correction models increase in cascades as it creates an interdependent system of systems leading to improvement issues in the long term.

The solution here will be to learn the correction directly on the first model by adding more feature rather than building dependents models. Another possibility is to create a separate model for A.’

## 2. Undeclared Consumers

Frequently, the predictions of a machine learning model are consumed by other systems. Without access controls on those predictions, some of these consumers may be undeclared. In more traditional software engineering terms, these issues are visibility debt [13]. Those undeclare consumers build dependency between the model and other parts of the stack. Changes to the model are likely to impact others parts leading to potential complexity, unintended, or poorly understood parts.

In practice, this tight coupling can radically increase the cost and difficulty of making any changes to the model at all, even if they are improvements. Furthermore, undeclared consumers may create hidden feedback loops. Undeclared consumers may be hard to detect unless the system is specifically built to guard against this case, for example, with access restrictions or strict service-level agreements (SLAs).

## 3. Data Dependencies

In traditional software engineering, a critical part of complexity arises from codes dependencies; however, in machine learning systems, data dependencies carry the most significant weight. Code dependencies can be identified via static analysis by compilers and linkers. Without tools to detect data dependencies, it can be easy to build large data dependency chains that can be hard to untangle, which rise the complexity.

### a- Unstable Data Dependencies

It is sometimes convenient to consume signals(data) as input features that are produced by other systems in order to move quickly. However, some input signals are unstable; they change behavior over time. This can be implicit, when the input signal comes from another machine learning model itself that updates over time, or a data-dependent lookup table, such as for computing TF/IDF scores or semantic mappings. It can also happen explicitly when the engineering ownership of the input signal is separated from the engineering ownership of the model that consumes it. So, updates to the input signal may be made at any time. This is dangerous because even “improvements” to input signals may have arbitrary detrimental effects in the consuming system that are costly to diagnose and address. For example, consider the case in which an input signal was previously miscalibrated. The model consuming it likely fit to this miscalibration, and a silent update that corrects the signal will have an immediate effect for the model. A solution to this problem is to create versioned copies until an updated version has been fully vetted. However, versioning can carry some complexity, such as the maintenance of multiple versions.

### b- Underutilized Data Dependencies

In traditional software-based systems, underutilized dependencies are packages that are mostly unneeded [13]. In machine learning, underutilized data dependencies are input signals that provide little incremental modeling benefit. These can make an ML system unnecessarily slow and vulnerable to change. As an example, suppose that during the transition from an old product scheme numbers to new product scheme, we leave all features in the system leading to maintenance complexity. Underutilized data dependencies can creep into a model in several ways.

### c- Legacy Features

The most common cause is that a feature F is included in a model early in its development. Over time, F is made redundant by new features, but this goes undetected. Bundled Features. Sometimes, a group of features is evaluated and found to be beneficial. Because of deadline pressures or similar effects, all the features in the bundle are added to the model together, possibly including features that add little or no value. E-Features. Often, it is tempting to improve model accuracy; however, accuracy gain could be minimal but raise complexity substantially. Correlated Features. Often two features are strongly correlated, but one is more directly causal. Many ML methods have difficulty detecting this and credit the two features equally, or may even pick the non-causal one. Underutilized dependencies can be detected via exhaustive leave-one-feature-out evaluations. These should be run regularly to identify and remove unnecessary features.

### d- Static Analysis of Data Dependencies.

For traditional software-based systems, compilers and build systems perform static analysis of dependency graphs. However, Tools for static analysis of data dependencies are scarce. One tool available to tackle the issue is the automated feature management system described in [12], which enables data sources and features to be annotated. Automated checks can then be run to ensure that all dependencies have the appropriate annotations, and dependency trees can be fully resolved.

## 4 Feedback Loops

One of the critical features of automated live machine learning systems is that they often end up influencing their behavior if they update over time. This generates a form of analysis problems, in which it is difficult to predict the behavior of a given model before it is released. These feedback loops can take different forms, but they are all more challenging to detect and address if they occur gradually over time, as may be the case when models are updated infrequently.

### a- Direct Feedback Loops

A model may directly influence the selection of its future training data. It is common practice to use standard supervised algorithms, although the theoretically correct solution would be to use bandit algorithms. The problem here is those bandit algorithms (such as contextual bandits [9]) do not necessarily scale well to the size of action spaces typically required for real-world problems. It is possible to mitigate these effects by using some amount of randomization [3], or by isolating specific parts of data from being influenced by a given model.

### b- Hidden Feedback Loops

Direct feedback loops are costly to analyze, but at least they pose a statistical challenge that researchers may find easy to investigate [3]. A more complicated situation is hidden feedback loops, in which two systems influence each other indirectly. Suppose that two systems independently determine facets of a web page, such as one selecting products to show and other selecting related reviews. Improving one system may lead to changes in behavior in the other, as users begin clicking more or less on the other components in reaction to the changes. Very often in the real world, hidden loops exist between completely disjoint systems in the case of stock-market prediction models from two different investment companies. Improvements (or, more scarily, bugs) in one may influence the bidding and buying behavior of the other.

## 5- Complex Anti-Patterns designs

In the real-world very often, only a tiny fraction of the code in many machine learning systems is devoted to learning or prediction – see Figure 1. In the language of Lin and Ryaboy, much of the remainder may be described as “plumbing” [11]. It is, unfortunately, typical for systems that incorporate machine learning methods to end up with intricate design patterns. Below are system-design anti-patterns [4] that can make machine learning systems unnecessarily complex

### a- Glue Code

Researchers often develop general-purpose solutions as generic packages. Using generic packages generate a glue code system design pattern, in which a massive amount of supporting code is written to get data into and out. Glue code is costly in the long term because it tends to freeze a system to the peculiarities of a specific package; testing alternatives may become prohibitively expensive. In this way, using a generic package can inhibit improvements, because it makes it harder to take advantage of domain-specific properties or to tweak the objective function to achieve a domain-specific goal. Because a sophisticated system might end up being (at most) 5% machine learning code and (at least) 95% glue code, it may be less costly to create a clean native solution rather than re-use a generic package. An essential strategy for combating glue-code is to wrap black-box packages into standard API’s. It helps dependent infrastructure to be more reusable and reduces issues or complexity in changing packages.

### b- Pipeline Jungles

As a particular case of glue code, pipeline jungles often appear in data preparation. These can evolve organically, as new signals are identified and new information sources added incrementally. Without care, the resulting system for preparing data in a machine learning-friendly format may become a jungle of scrapes, joins, and sampling steps, often with intermediate files output. Managing these pipelines, detecting errors, and recovering from failures are all problematic and costly [1]. Testing such pipelines often requires expensive end-to-end integration tests. All of this adds to technical debt of a system and makes further innovation more costly. Pipeline jungles can only be avoided by thinking holistically about data collection and feature extraction. The clean-slate approach of scrapping a pipeline jungle and redesigning from the ground up is indeed a significant investment of engineering effort, but one that can dramatically reduce ongoing costs and enhance innovation. Glue code and pipeline jungles are signs of integration issues that may have a root cause in overly separated “research” and “engineering” roles. Environnement, where engineers and researchers are embedded together on the same teams (and indeed, are often the same people), can help reduce this source of complexity significantly [16].

### c- Dead Experimental Codepaths

A common consequence of glue code or pipeline jungles is that it becomes increasingly attractive in the short term to perform experiments with alternative methods by implementing experimental codepaths as conditional branches within the main production code. For any individual change, the cost of experimenting in this manner is relatively low—none of the surrounding infrastructures needs to be reworked. However, over time, these accumulated codepaths can create complex issues due to the increasing difficulties of maintaining backward compatibility and an exponential increase in cyclomatic complexity. Testing all possible interactions between codepaths becomes difficult or impossible. A famous example of the dangers here was Knight Capital’s system losing $465 million in 45 minutes, apparently because of unexpected behavior from obsolete experimental codepaths [15]. As with the case of dead flags in traditional software [13], it is often beneficial to periodically re-examinee experimental branches to see what can be removed. Often only a small subset of the possible branches is used; many others may have been tested once and abandoned.

### d- Lack of Abstractions

One of the biggest challenges faced by machine learning systems is the lack of substantial abstractions to support those systems. Zheng made a compelling comparison of the state machine learning abstractions to the state of database technology [17], making the point that nothing in the machine learning literature comes close to the success of the relational database as a necessary abstraction. What is the right interface to describe a stream of data, or a model, or a prediction?

For distributed learning, in particular, there remains a lack of widely accepted abstractions. It could be argued that the widespread use of Map-Reduce in machine learning was driven by the void of strong distributed learning abstractions. Indeed, one of the few areas of broad agreement in recent years appears to be that Map-Reduce is a poor abstraction for iterative machine learning algorithms. The parameter-server abstraction seems much more robust, but there are various competing specifications of this basic idea [5, 10]. The lack of standard abstractions makes it all too easy to blur the lines between components.

### e- Common Smells

In software engineering, a design smell may indicate an underlying problem in a component or system [7]. Similarly, there are machine learning systems smells. Plain-Old-Data Type Smell.  
‘The rich information used and produced by ML systems is all to often encoded with plain data types like raw floats and integers. In a robust system, a model parameter should know if it is a log-odds multiplier or a decision threshold, and a prediction should know various pieces of information about the model that produced it and how it should be consumed.’

### f- Multiple-Language Smell

It is often tempting to write a particular piece of a system in a given language, especially when that language has a convenient library or syntax for the task at hand. However, using multiple languages in the same system often increases the cost of adequate testing and can increase the difficulty of transferring ownership to other individuals.

### - Prototype Smell

It is convenient to test new ideas in small scale via prototypes. However, regularly relying on a prototyping environment may be an indicator that the full-scale system is brittle, difficult to change, or could benefit from improved abstractions and interfaces. Maintaining a prototyping environment carries its own cost, and there is a significant danger that time pressures may encourage a prototyping system to be used as a production solution. Additionally, results found at a small scale rarely reflect the reality at full scale.

## 6- Complex Configurations

Another potential area where problems and complexity can accumulate is in the configuration stage of machine learning systems. Any extensive system has a wide range of configurable options, including which features are used, how data is selected, a wide variety of algorithm-specific learning settings, potential pre- or post-processing, verification methods. Often engineers and researcher do not pay much attention to configuration; however, with sophisticated systems, the number of lines of configuration can far exceed the number of lines of the traditional code. So attention to the configuration is crucial to avoid potential mistakes those mistakes in configuration can be costly, leading to a severe loss of time, waste of computing resources, or production issues.

Mitigation strategies to avoid complex configuration are :

It should be easy to specify a configuration as a small change from a previous configuration.

It should be hard to make manual errors, omissions, or oversights.

It should be easy to see, visually, the difference in configuration between the two models.

It should be easy to automatically assert and verify basic facts about the configuration: the number of features used and transitive closure of data dependencies.

It should be possible to detect unused or redundant settings.

Configurations should undergo a full code review and be checked into a repository.

## 7- A complex world, changes in the external world

Machine learning systems are used to tackle real-world problems; the world itself is a precarious system, that generates a need to maintain machine learning systems continuously.

### a- Fixed Thresholds in Dynamic Systems

It is often necessary to pick a decision threshold for a given model to perform some action: to predict true or false, to mark an email as spam or not spam, to show or not show a given ad. One classic approach in machine learning is to choose a threshold from a set of possible thresholds, in order to get the right tradeoffs on specific metrics, such as precision and recall. However, such thresholds are often manually set. Thus if a model updates on new data, the old manually set threshold may be invalid. Manually updating many thresholds across many models is time-consuming and brittle. One mitigation strategy for this kind of problem appears in [14], in which thresholds are learned via simple evaluation of holdout validation data.

### b- Monitoring and Testing

Unit testing of individual components and end-to-end tests of running systems are valuable, but in the face of a changing world, such tests are not sufficient to provide evidence that a system is working as intended. Comprehensive live monitoring of system behavior in real-time combined with an automated response is critical for long-term system reliability.

The critical question is: what to monitor? Testable invariants are not always evident, given that many machine systems are intended to adapt over time. We offer the following starting points.

***Prediction Bias***

In a system that is working as intended, it should usually be the case that the distribution of predicted labels is equal to the distribution of observed labels. This is by no means a comprehensive test, as it can be met by a null model that predicts average values of label occurrences without regard to the input features. However, it is a surprisingly useful diagnostic, and changes in metrics such as this are often indicative of an issue that requires attention. For example, this method can help to detect cases in which the world behavior suddenly changes, making training distributions drawn from historical data no longer reflective of current reality. Slicing prediction bias by various dimensions isolate issues quickly, and can also be used for automated alerting.

***Action Limits***

In systems that are used to take actions in the real world, such as bidding on items or marking messages as spam, it can be useful to set and enforce action limits as a sanity check. These limits should be broad enough not to trigger spuriously. If the system hits a limit for a given action, automated alerts should fire and trigger manual intervention or investigation.

***Up-Stream Producers***

Data is often fed through to a learning system from various upstream producers. These up-stream processes should be thoroughly monitored, tested, and routinely meet a service level objective that takes the downstream machine learning system needs into account. Further, any up-stream alerts must be propagated to the control plane of a machine learning system to ensure its accuracy. Similarly, any failure of the machine learning system to meet established service level objectives be also propagated down-stream to all consumers, and directly to their control planes if at all possible. Because external changes occur in real-time, a response must also occur in real-time as well. Relying on human intervention in response to alert pages is one strategy, but can be brittle for time-sensitive issues. Creating systems that allow automated response without direct human intervention is often well worth the investment.

## 8- Complexity in Data testing

There is saying that garbage in, garbage out. Data is crucial in machine learning and determine the insights. So continuous testing on data is imperative for any sophisticated system to give valuable insights.

## 9- Complexity in reproducibility

An area of complexity encounter in machine learning is the reproducibility, very often a system needs to be tested on the ability to produce the same results, but this is made difficult by randomized algorithms and the changing external world.

## 10- Complexity in scale

Sophisticated ML systems usually have multiple or even hundreds of models running concurenlty[14,6], managing such a system could challenging, especially if there are manuals steps involved. So a mitigation strategy will be to reduce the number of manual steps as much as possible during the system design.

## 11- Scarcity of data

Some machine learning systems do not have enough data to make reliable inferences, a sector such as healthcare is one.

## 12-scrappy design

when building a prototype system, it is fine to be scrappy, but for the final product, a good clean up of codes and feature should be implemented

## 13- Lack of vetting

a robust system should be vet properly; a system built without another person or team vetting could have severe issues in the long term

## 14- lack of proper documentation

A reliable system should be well documented. New team members can maintain systems well if they are adequately documented.

## 15 reusable build features

A sound system must possess a feature engineering framework that incorporates reusable ML features. Without such a framework, it is essential to try and implement features that are reusable, transformable, interpretable, and reliable.

## 16- adding unnecessary features

Sound systems should not have unnecessary features. more future rise the complexity of the system, add more computing resources

## 17- avoid complex models

Good practice to reduce complexity in machine learning systems is to use the simplest models possible. A generally agreed rank of complexities from simple to very complex models is below:

Linear regression

Logistic regression

Collaborative filtering

Random forests

Gradient boosted decision trees

Elastic nets

LambdaMart and other learning-to-rank approaches

Neural networks

That said, the devil is in the details, and you could have a simple implementation of gradient boosted decision trees that is simpler than a complex implementation of logistic regression. Also, it is essential to remember that there is a direct interaction between model complexity and number (and type) of features. So, a complex model might not show any quality wins simply because it does not have the right features to learn.

## 18- Open-source

There many open source tools; researchers, data scientits, machine learning engineers or systems scientists should use them as much as possible and build their system on top. Usually, those open sources are fully vetted and well tested and present less risk.

## 19- Complexity in multiple ways in the system

Avoid using multiple ways of doing the same thing. All options should be explored, and the optimal one should be implemented.

## 20- Conclusions

In this paper, we use systems thinking-level to highlight machine learning systems issues. We highlight the need to have a longterm vision in mind during design. While most engineers and researchers see the necessity to move fast building ML systems, keeping in mind issues that this speed can create roadblock can help reduce the long term issues that any system will encounter later.

It will not be possible to avoid all issues as the external -world is an unstable system that in which machine learning systems operate, but being aware of its dependencies help develop better mitigation strategies for problems faced by ML systems. To work on schedule and under certain constrains Almost all team in machine learning will accept to take controversial steps that need to be addressed later, it is referred as technical debt, but successful ML teams recognize, prioritizing and reward effort to integrate and design strategies to mitigate future problems. It is referred to as paying down the technical debt.

Lately, we have seen an explosion of packages and tools to handle similar problems faced by the different team while innovation is excellent as it provides choice, we believe it slows the progress on the overall machine learning community as much time is used rebuilding the same thing.

We strongly suggest focusing on improvement rather than building from scratch. In this chapter, we focus on the system-level proprieties of machine learning systems; we spend time mentioning the complexity around the data. The following chapter, we will highlight the complexity around datasets. Machine learning is worthless without data, and the data is actually where the insights are hidden, we will discuss and dissect the datasets. A good systems-thinker which we will refer in the remaining chapters as systems scientist must perceive data as a system. A solid understanding of data lead to better inferences and provides actionable insights.

## Reference

0 - The High-Interest Credit Card of Technical Debt - DocShare … <http://docshare.tips/the-high-interest-credit-card-of-technical-debt_58c498a5b6d87fc5418b5c34.html>

1- George Rzevski - On Complexity. <https://www.rzevski.net/on-complexity/>

2- MANAGING AIRCRAFT LIFECYCLE COMPLEXITY. <https://www.witpress.com/Secure/ejournals/papers/DNE110201f.pdf>

3- MANAGING AIRCRAFT LIFECYCLE COMPLEXITY. <https://www.witpress.com/Secure/ejournals/papers/DNE110201f.pdf>

4- Complexity Science - andywhiteanthropology.com. <https://www.andywhiteanthropology.com/complexity-science.html>

5- George Rzevski - On Complexity. <https://www.rzevski.net/on-complexity/>

6- Prigogine, Ilya, The End of Certainty: Time, Chaos and the new Laws of Nature, Free Press, 1997, ISBN 0-684-83705-6.

7- Prigogine, Ilya, Is Future Given?, World Scientific Publishing Co., 2003, ISBN 981-238-508-8.

8- Kaufman, S., At Home In the Universe: The Search for the Laws of Self-Organization and Complexity. Oxford Press, 1995, ISBN 0-19-511130-3.

9- Holland, J., Emergence: from Chaos to Order, Oxford University Press, 1998, ISBN 0-19-850409-8.

10- Rzevski, G., Skobelev, P., Managing Complexity, WIT Press, 2014, ISBN 978-1-84564-936-4.

11- Rzevski, G., Skobelev, P., “Emergent Intelligence in Large Scale Multi-Agent Systems.” International Journal of Education and Information Technology, Issue 2, Volume 1, 2007, pp. 64-71.

12- Rzevski, G., “Using complexity science framework and multi-agent technology in design.” In Alexiou, K., Johnson, J., Zamenopoulos, T. (eds.), Embracing Complexity in Design, Routledge, 2010, pp. 61-72, ISBN 978-0-415-49700-8.

13- Popper, Karl, Conjectures, and Refutations: The Growth of Scientific Knowledge, Routledge & Kegan Paul, Fourth Edition, 1972, ISBN 0-7100-6508-6.

14- Hidden Technical Debt in Machine Learning Systems. <https://papers.nips.cc/paper/5656-hidden-technical-debt-in-machine-learning-systems.pdf>

15- The 21st century Age of Complexity: What is it and what … <http://smartorg.sg/the-21st-century-age-of-complexity-what-is-it-and-what-does-it-mean/>

[1] R. Ananthanarayanan, V. Basker, S. Das, A. Gupta, H. Jiang, T. Qiu, A. Reznichenko, D. Ryabkov, M. Singh, and S. Venkataraman. Photon: Fault-tolerant and scalable joining of continuous data streams. In SIGMOD ’13: Proceedings of the 2013 international conference on Management of data, pages 577– 588, New York, NY, USA, 2013.

[2] A. Anonymous. Machine learning: The high-interest credit card of technical debt. SE4ML: Software Engineering for Machine Learning (NIPS 2014 Workshop).

[3] L. Bottou, J. Peters, J. Quin ̃onero Candela, D. X. Charles, D. M. Chickering, E. Portugaly, D. Ray, P. Simard, and E. Snelson. Counterfactual reasoning and learning systems: The example of computational advertising. Journal of Machine Learning Research, 14(Nov), 2013.

[4] W. J. Brown, H. W. McCormick, T. J. Mowbray, and R. C. Malveau. Antipatterns: refactoring software, architectures, and projects in crisis. 1998.

[5] T. M. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In 11th USENIX Symposium on Operating Systems Design and Implementation, OSDI ’14, Broomfield, CO, USA, October 6-8, 2014., pages 571–582, 2014.

[6] B. Dalessandro, D. Chen, T. Raeder, C. Perlich, M. Han Williams, and F. Provost. Scalable hands- free transfer learning for online advertising. In Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining, pages 1573–1582. ACM, 2014.

[7] M. Fowler. Code smells. <http://http://martinfowler.com/bliki/CodeSmell.html>.

[8] M. Fowler. Refactoring: improving the design of existing code. Pearson Education, India, 1999.

[9] J. Langford and T. Zhang. The epoch-greedy algorithm for multi-armed bandits with side information. In Advances in neural information processing systems, pages 817–824, 2008.

[10] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B. Su. Scaling distributed machine learning with the parameter server. In 11th USENIX Symposium on Operating Systems Design and Implementation, OSDI ’14, Broomfield, CO, USA, October 6-8, 2014., pages 583–598, 2014.

[11] J. Lin and D. Ryaboy. Scaling big data mining infrastructure: the twitter experience. ACM SIGKDD Explorations Newsletter, 14(2):6–19, 2013.

[12] H. B. McMahan, G. Holt, D. Sculley, M. Young, D. Ebner, J. Grady, L. Nie, T. Phillips, E. Davydov, D. Golovin, S. Chikkerur, D. Liu, M. Wattenberg, A. M. Hrafnkelsson, T. Boulos, and J. Kubica. Ad click prediction: a view from the trenches. In The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2013, Chicago, IL, USA, August 11-14, 2013, 2013.

[13] J. D. Morgenthaler, M. Gridnev, R. Sauciuc, and S. Bhansali. Searching for build debt: Experiences managing technical debt at Google. In Proceedings of the Third International Workshop on Managing Technical Debt, 2012.

[14] D. Sculley, M. E. Otey, M. Pohl, B. Spitznagel, J. Hainsworth, and Y. Zhou. Detecting adversarial advertisements in the wild. In Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Diego, CA, USA, August 21-24, 2011, 2011.

[15] Securities and E.Commission.SEC Charges Knight Capital With Violations of Market Access Rule,2013.

[16] A.Spector, P.Norvig, and S.Petrov. Google’s hybrid approach to research.CommunicationsoftheACM, 55 Issue 7, 2012.

[17] A.Zheng.The challenges of building machine learning tools for the masses.SE4ML: Software Engineering for Machine Learning (NIPS 2014 Workshop).

19- Hidden Technical Debt in Machine Learning Systems. <https://papers.nips.cc/paper/5656-hidden-technical-debt-in-machine-learning-systems.pdf>

20- Technical Debt in Machine Learning - Part 1 -ml-dl. <http://ml-dl.com/technical-debt-in-machine-learning-1/>

21- Technical Debt in Machine Learning - Part 1 - ml-dl. <http://ml-dl.com/technical-debt-in-machine-learning-1/>

22- HARNESSING THE POWER OF SELF-ORGANISATION. <https://www.witpress.com/Secure/ejournals/papers/DNE110401f.pdf>