Deep Learning an Approach in Software Engineering

Nivedita Tiwari, Dr. Lalji Prasad:
1 - Computer Science and Engineering, SAGE University, Indore, INDIA, nivedita.8584@gmail.com
2 - Computer Science and Engineering, SAGE University, Indore, INDIA, lalji.research@gmail.com

ABSTRACT

Code components are major keys of Software Engineering (SE) tasks, such as clone detection, impact analysis, refactoring, when identifiers are used to represent the code, that Deep Learning (DL) can replace manual feature engineering for the task of clone detection. An orthogonal view of the same code fragment use to enable a more reliable detection of similarities in code. The goal of this research to examines applications of deep learning to SE tasks and hence identifies main challenges by applying an interpretive research approach and main SE challenges with building systems with DL components. The challenges identified in this paper can be used to guide future research by the software engineering and DL component.

KEYWORDS: Software Engineering task, Neural Network, Deep Learning Architecture.

1. Introduction:

Software Engineering (SE) researchers are also starting to explore the application of DL in traditional SE problems and areas, such documentation, defect prediction, and testing Deep Learning (DL) is a subfield of Machine Learning (ML) that relies on multiple layers of Neural Networks (NN) to model high level representations [8]. Source code identifiers and comments, Abstract Syntax Trees (ASTs), Control-Flow Graphs (CFGs), data flow, byte code, etc. These distinct representations of code provide different levels of abstraction, which create explicit and implicit relationships among source code elements. Programs that have a well defined syntax can be represented by ASTs, which, in turn, can be successfully used to capture programming patterns and similarities [10, 17, 18]. "Hardcoded" algorithms rooted in the underlying properties of the specific code representation they use, which in order to work properly also need to be adequately configured [20, 21, 29]. Distinct identifiers are typically used as features in concept location approaches [19], APIs (a reserved subset of identifiers) are used as features in approaches for identifying similar applications [39, 40], automatically extract useful information from the disproportionate amount of unlabeled software data. The value in circumventing conventional feature engineering when designing ML-based approaches to SE tasks is two-fold Firstly, learning transformations of the data drastically reduces the cost of modeling code, since software store a lot of data to improve the effectiveness of learning. Secondly, performance is increased because generally, learning algorithms are more efficient than humans at discovering correlations in high dimensional space for example, these approaches are designed to automatically induce feature hierarchies [9], which may drastically reduce the cost of modeling software artifacts and improve the performance since machines are relatively efficient at discovering correlations in high-dimensional spaces.

1.1 Machine Learning and Software Engineering

A key difference between ML systems and non-ML systems is that data partly replaces code in a ML system, and a learning algorithm is used to automatically identify patterns in the data instead of writing
hard coded rules. This suggests that data should be tested just as thoroughly as code.

1.2 Big Data and Deep Learning

Big data is often loosely defined by the three Vs: Volume, Variety, and Velocity [11]. It is not only the volume of data that may require big data techniques, but also the variety of different types and formats of data. This is strength of DL, which can use representation learning to combine different modalities such as images, audio, text, and tabular data [28], [43]. It is not only the process of extraction, transforming and loading (ETL) data but also that novel distributed training algorithms may need to be used. Especially DL techniques are not trivially parallelized and again need special supporting infrastructure [44], [45].

Data dependencies have been found to build similar debt as code dependencies. Few tools currently exist for evaluating and analyzing data dependencies, especially when compared to static analysis of code using compilers and build systems. One example of a data dependency tool is described in [12], where data sources and features can be annotated.

Deep Learning also makes it possible to compose complex models from a set of sub models and potentially reuse pertained parameters with so called “transfer learning” techniques. This not only adds additional dependencies on the data, but also on external models that may be trained separately. External models add additional dependency debt and cost of ownership for ML systems. Dependency debt is noted as one of the key contributors to technical debt in SE [33]. An ML system is usually dependent on many different pipelines that may also be implemented in different languages, formats, and infrastructure systems. It is not uncommon for pipelines to change, add and remove fields, or become deprecated. Keeping a deployed production-ready ML system up to date with all these changes require a significant amount of work, and requires supporting monitoring and logging systems to be able to detect when these changes occur, dead flags in traditional software [33]. Over time, these accumulated code paths create a growing debt that will be difficult to maintain [46].

Another technical debt problem of ML systems is configuration management. Configuration of ML systems is frequently not given the same level of quality control as software code, even though it can both have a higher impact on the performance of the model and also be large in size [14].

2. Background and related work

There has been some preliminary work on using DL to replace manual feature engineering for the task of clone detection [41]. However, we are not aware of any other learning-based approach that operates on code representations other than identifiers and comments. In this section, we review some of the recent and representative papers that rely on DL in the context of SE tasks.

White et al. [41] used DL and, in particular, representation learning via recursive auto encoder, for the purpose of code clones detection. They represent each source code fragment as a stream of identifiers and literal types (from the AST leaf nodes).

Lam et al. [42] focuses on bug localization using both DL and IR techniques. Rather than manually defining features, the IR technique revised Vector Space Models (rVSM) collects these features which capture textual similarity. After that, a deep neural network learns how to relate terms in bug reports to tokens within source code.

Allamanis et al. [27] proposed an approach for suggesting meaningful class and method names. To accomplish this, identifiers in code are assigned a continuous vector, which considers the local context as well as long range dependencies by a neural log-bilinear context model. Then, identifiers which have similar vectors or embeddings will also appear to have similar contexts which require configuration and
manipulation when integrated with the model. To build upon their previous work, Allamanis et al. [38] used an Attentional Neural Network (ANN) combined with convolution on the input tokens for assigning descriptive method names. Their approach allows for automatic learning of translation in variant features.

Gu et al. [35] used DL, to avoid feature engineering, in order to learn API usage scenarios given a natural language query. The approach encodes a query or annotation of the code into a fixed length context vector. This vector helps to decode an API sequence which should correlate with the query. Therefore, once the model is trained, a natural language query will result in the correct API usage scenario for the context given in the query.

Wang et al. [36] used a Deep Belief Network (DBN) to learn semantic features from token vectors extracted from ASTs to perform defect prediction. Similar to our work, they learn a code representation and apply it to a SE task.

Hindle et al. [15] demonstrated that language models over software corpora emit a “naturalness” in the sense that real programs written by real people have useful statistical properties encapsulated in statistical language models, that can leverage SE tasks. This work was an important first step in applying natural language abstractions to software corpora, but n-grams are simple approaches that do not have the capacity to learn representations that reliably generalize beyond the explicit features in a training corpus [34]. Furthermore, these models build limited domain abstractions, and they are quickly overwhelmed by the curse of dimensionality [15], [22]–[25]. The expectation in software language modeling research is that performance at SE tasks will improve with models more sophisticated than n-grams [15].

Allamanis and Sutton [38] estimated an n-gram from a software corpus with more than one billion tokens, but we regard the massive scale as an organic smoothing technique. The model’s effectiveness is still subject to token distances in the corpus, where clues behind the n-gram’s relatively short prefix (or “history”) are elided from the model’s context [22], [25], [26]. Moreover, the massive scale does not truly solve the problem of considering tokens’ semantic similarity [22], [25].

3. Approach and uniqueness

In Deep Software Language Model, recurrent neural network (RNN) augments two-layer feed-forward architecture with a memory by recurring the hidden layer, which encapsulates the network’s state, for an arbitrary number of levels of context. The directed cycle provides context for the current prediction, and this continuous-valued state vector is fundamentally different than a discrete token in an n-gram’s history. The purpose of the depth in this architecture is to reliably model temporal dependencies. A deep architecture for software language modeling is specified in Eq. (1)–(3).

\[
x_i(t) = (w(t), z(t-1))_i \\
z_j(t) = \text{sigmoid}(\alpha_{ji}, x_i(t)) \\
y_k(t) = \text{softmax}(\beta_{kj}, z_j(t))
\]

Where \( w \) is a token in the vocabulary, \( z \) is the hidden layer, \( x \) is the input layer, \( \alpha \) and \( \beta \) are linear maps, \( y \) is the output layer, and we omit the bias terms. The model prediction is the depth of a RNN is evident when you unfold the recurrence in time and measure the path from any unit in the deepest state vector to any output unit. In practice, we use several heuristics to control the complexity with learning one of these models from massive repositories. Indeed, these architectures entail a number of hype parameters spanning a voluminous design space with incredible research potential in many different SE contexts.

Deep IR Model, A restricted Boltzmann machine (RBM) is a bipartite network of binary visible unit \( v \) and stochastic binary hidden unit \( h \), computing the posterior distribution under this topology is simply
Learning involves updating the weights $w$ so the energy of a joint configuration $p(v,h)$ is minimized. Once the model is trained, the hidden units can detect latent features of the input. The power comes with stacking these generative models, and if the number of hidden units in each layer is strictly less than the number of “visible” units, then the network is forced to learn a compressed representation of the input, i.e., the network encodes the input using a series of nonlinear transformations. Our research examines application of these codes to clustering artifacts in software repositories.

4. Selected Challenges

This section presents a list of concisely described challenges in the intersection between ML and SE. They have been grouped into three categories: development, production, and organizational challenges.

Development Challenges

There are fundamental differences between developing ML systems compared to traditional SE systems. One of the main differences is that data is used to program the system “automatically” instead of writing the software code manually. The performance of the system is unknown until it has been tested with given data, making it difficult to plan the project in a structured manner. In addition, the lack of model transparency, inability to understand large and complicated models, and difficulty in debugging using libraries with lazy execution makes it challenging to estimate the effort needed to complete the project.

1) Experiment Management: During the development of ML models, a large number of experiments are usually performed to identify the optimal model. Each experiment can differ from other experiments in a number of ways and it is important to ensure reproducible results for these experiments. To have reproducible results, it may be necessary to know the exact version of components such as:
   a) Hardware (e.g. GPU models primarily)
   b) Platform (e.g. operating system and installed packages)
   c) Source code (e.g. model training and pre-processing)
   d) Configuration (e.g. model configuration and preprocessing settings)
   e) Training data (e.g. input signals and target values)
   f) Model state (e.g. versions of trained models).

Version control for ML systems adds a number of challenges compared to traditional software development, especially given the high level of data dependency in ML systems. Different versions of data will yield different results, and the input data are often a conglomerate of data from multiple heterogeneous data sources. It has been argued that one of the most difficult components to keep track of is the data component, and the cost and storing of versioned data can be very high.

2) Limited Transparency: Software engineering is based on the principle of reducing complex systems into smaller, simpler blocks. Whenever possible, it is desirable to group blocks into different levels of abstraction that have a similar conceptual meaning. Although DL systems, in principle, do that automatically, it is very difficult to know exactly how it is performed or predict what the abstraction layers will look like once the model has been trained. Furthermore, it is difficult to isolate a specific functional area or obtain a semantic understanding of the model. This can only be performed with approximated methods [32].

3) Troubleshooting: A major challenge in developing DL systems is the difficulties in estimating the results before a system has been trained and tested. Furthermore, our poor understanding of the inner workings of complex neural networks makes it difficult to debug them in a traditional way. In a neural network, the structure combines the functional parts and the memory and can also be distributed across multiple machines. In addition, using libraries such as TensorFlow [13] potentially combed with big data.
frameworks such as Apache Spark [47] makes it difficult to troubleshoot and debug problems in the code. As they both have a lazy execution graph, where the code is not executed in imperative order, it can be difficult to troubleshoot bugs using traditional SE tools. Other frameworks such as PyTorch [37], do not have the lazy execution graph problem but have other issues such as lower adoption rate in the AI community.

4) Resource Limitations: Working with data that require distributed system adds another magnitude of complexity compared to single machine solutions. It is not only the volume of the data that may require a distributed solution, but also computational needs for extracting and transforming data, training and evaluating the model, and/or serving the model in production. For DL systems, it can also be limited GPU memory that requires special techniques to split the model across multiple GPUs. Working with distributed systems, both for data processing such as Apache Spark [47] and DL training such as Distributed Tensor Flow or Tensor Flow On Spark [45], [13], adds complexity in a number of dimensions. It not only requires additional knowledge and times to operate them, but also additional management and cost of associated hardware and software.

5) Testing: ML systems require testing of software used for building data pipelines, training models, and serving in production. Given the high data dependency of ML systems, data also need to be tested. However, currently only a few data testing tools exist compared to software testing. A frequent pattern seen when testing data is to make use of a small sample of the full dataset. It is challenging to provide a sample that includes all the edge cases that may exist in the full dataset. Also, as the external world is dynamic and changes over time, new edge cases will continue to appear later in time.

B. Production Challenges

Particularly for DL, it is important to take advantage of the latest hardware. This yields a significant challenge to maintain frequent updates to state-of-the-art software and managing associated dependencies. To accurately detect problems introduced by changing behavior in dependencies, including data sources that have been modified, requires careful and clever monitoring.

1) Dependency Management: Traditional SE typically builds on the assumption that hardware is at best a nonissue and at worst a static entity that has to be taken into consideration. DL systems are primarily trained on GPUs as they provide a 40-100x speedup over classic CPUs. For the past 5 years, the performance has significantly improved and new GPUs are released 1–2 times per year. The DL software platforms are continuously updated on a weekly and sometimes daily basis and the updates typically result in noticeable improvements. This works well for academic research and for developing proofs of concept but can cause considerable issues for production-ready systems. Unlike other ML methods, DL often scales directly with model size and data amount. As training times can be very long (typically a few days up to several weeks) there is a very strong motivation to maximize performance by using the latest software and hardware.

2) Monitoring and Logging: Building a toy example ML system or even an offline research prototype is easy compared to the amount of work required to build a production-ready ML system [31]. In real-world ML applications beyond toy examples, it can become difficult to cover all the edge cases that may occur once a model has been deployed in production. It is also common that people fail to recognize the effort needed to maintain a deployed ML system over time [50]. An ML system may be retrained frequently and thus change behavior autonomously. As the behavior of the external world changes, the behavior of the ML system can suddenly change without any human action in “control” of the system. In this situation, unit testing and integration tests are valuable but not sufficient to validate the performance of the system. Old thresholds that may have been manually assigned may no longer be valid given drifts in the data from the external world. Live monitoring of the system performance can help, but choosing what metrics to monitor can be challenging.
3) Unintended Feedback Loops: A ML system is, by definition, always open-ended as it is driven by external data. No matter how carefully the model is designed and tested, its final performance will always be heavily dependent on the external data [50]. Furthermore, especially in models deployed in a big data context (as ML systems often are), there is a risk of creating an unintended feedback loop where real-world systems adapt to the model rather than the other way around.

4) Glue Code and Supporting Systems: An unfortunate property of ML systems, and especially DL systems, is that only a small part of the system deals with the model. In a production-ready system, only 5% of the code may deal with the model and the rest is “glue code” that interacts with supporting systems and glues libraries and systems together [50]. Using cloud services can greatly improve development time and decrease maintenance needs. However, keeping the glue code up to date, and keeping up with external changes in cloud services, can introduce unexpected challenges in production ready systems.

C. Organizational Challenges

To put an ML model into production usually requires collaboration between many different teams with different ideas, priorities, and cultural values. This not only introduces organizational challenges from a cultural point of view, but also in being able to properly estimate the amount of effort needed by the different types of teams.

Effort Estimation: The reductionist modular SE design of a non-ML project makes it considerably easier to estimate the time and resources required to complete it. In a ML project, the goal might also be well defined but it is unclear to what extent a learned model will achieve that goal, and an unknown number of iterations will be needed before the results reach acceptable levels. This is in the nature of any research project. It is can also be difficult to decrease scope and run the project in a time-based setting, with a predefined delivery date, since it is hard to determine when, if at all, acceptable performance will be achieved.

2) Privacy and Data Safety: A lack of understanding of the internal workings of a large neural network can have serious implications for privacy and data safety. The knowledge in a neural network is stored in a distributed manner across the weights of the network. Although we know that specialization occurs in specific regions of the neural network, its exact mechanism is poorly understood. Thus, it is very difficult for designers to control where and how information is stored. It also not uncommon for companies to have terms of service agreements with their end-users that prevents them from using raw data as direct input to a ML model, and instead have to make use of anonymized and/or aggregated statistics of the user data [48]. This can not only reduce the performance on the model, but can also make tasks such as data exploration and troubleshooting problems more difficult. New regulations such as the European General Data Protection Regulation [30] go to great lengths to keep the data safe and to protect privacy concerns. However, while keeping data safe is important, it also adds significant challenges in how to develop and manage ML systems.

3) Cultural Differences: Building a production-ready ML system usually involves collaboration between people with different roles. A data scientist might be “pragmatic” about their code as long as it achieves the desired results in controlled environment, whereas members of the engineering teams care much more about maintainability and stability. Transforming an initial prototype into a production-ready system that also interacts with existing backend and frontend systems usually requires a significantly larger effort. This normally includes collaboration with, for example, backend engineers, UX designers, and product owners.

Conclusion

The main goal of this research is to identify and outline main SE challenges with building systems.
with DL components. Several projects were described to exemplify the potential for making use of the ML and specifically the DL technology. For these projects, the main problematic areas and challenges with building these systems were identified. To clarify these problematic areas in more detail challenges were identified and described in the areas of: development, production, and organizational challenges. One clear conclusion of this work is that, although the DL technology has achieved very promising results, there is still a significant need for further research into and development in how to easily and efficiently build high-quality production ready DL systems. Traditional SE has high-quality tools and practices for reviewing, writing test, and debugging code. However, they are rarely sufficient for building production ready systems containing DL components. If the SE community, together with the DL community, could make an effort in finding solutions to these challenges, the power of the DL technology could not only be made available to researchers and large technology companies, but also to the vast majority of companies around the world.

REFERENCES


42. A. Lam, A. Nguyen, H. Nguyen, and T. Nguyen. [n. d.]. Combining Deep Learning with Information Retrieval to Localize Buggy Files for Bug Reports (ASE’15)

