Algebraic Convergence to Software-Knowledge: Deep Software Learning (TSE)

Iaakov Exman and Assaf B. Spanier
Software Engineering Department
The Jerusalem College of Engineering – JCE - Azrieli
Jerusalem, Israel
iaakov@jce.ac.il, shpanier@jce.ac.il

***Abstract*— It is an empirical observation that Software Engineering and Knowledge Engineering seem to converge to a single discipline which may be suitably called *Software-Knowledge*. However, mere empirical observations are not satisfactory. These should be justified by plausible arguments. There are three convergence aspects, semantic, algebraic and topological, and this paper focuses on the algebraic aspect. Linear algebra is the basis for Linear Software Models, a rigorous theory of software systems composition from sub-systems, recently developed. Linear algebra, with added non-linearity, is also the basis for Deep Learning, a successful Artificial Intelligence domain. This work suggests and analyzes *Deep Software Learning*, i.e. Deep Learning specific to Software development problems. We then conjecture on deep reasons for Software-Knowledge convergence.**

Keywords: Linear Software Models; Software Composition; Laplacian Matrix; Deep Software Learning; Software-Knowledge Convergence; RNN; LSTM; Sequential and Structured Data.

I. Introduction

It has been observed empirically that the Software Engineering and Knowledge Engineering disciplines seem to converge along their relatively short histories. Convergence is interesting theoretically and in practice. This paper analyzes Deep Software Learning as a mutual Software and Knowledge interaction for Software Development problems.

This work ultimate goal is to point out concrete directions to the rationale of Software-Knowledge convergence, starting from plausible conjectures. It offers a discussion roadmap for the Theory of Software Engineering special session on Software-Knowledge convergence, within the SEKE'2019 conference.

A. Concise Historical Overview

Software as a discipline starts in 1956 with Backus' Fortran a *high-level* programming language. Software Engineering itself was coined only in the celebrated NATO 1968 conference. Software history is since then a continuous increase in abstraction level of languages and design techniques. From structured programming, to object-oriented languages, as Java, to modeling languages, as UML, and model-driven-engineering up to ontologies as conceptual refinement of classes and inheritance by subclasses.

Knowledge, an Artificial Intelligence (AI) field, started in 1950, with Shannon's [15], and Turing's [16] pioneer papers. Next appear classical AI expert-systems, e.g. Dendral to resolve chemical structural formulas. These systems separated inference engines from knowledge bases. Ontologies resulted from this research thread.

An early algebraic learning theory is the 1969 Perceptrons by Minsky and Papert [13]. A neural networks sub-field developed, remaining in research laboratories, for the lack of computing power. They were renamed "Deep Learning" with the industrial applications surge, due to added computing power (e.g. GPU), big data sets, and algorithmic improvements.

B. Aspects of Software-Knowledge Convergence

This paper is motivated by the following conjecture:

<u>Software-Knowledge Conjecture</u>
Software Engineering and Knowledge Engineering are converging to a single discipline which we call ***Software-Knowledge***.

Software-Knowledge convergence consists of three aspects:

1) ***Semantic*** – the importance of concepts and ontologies in both software and knowledge fields;
2) ***Algebraic*** – mostly linear algebra as the basis of software composition theory and Deep Learning in diverse knowledge domains; this paper's focus;
3) ***Topological*** –graphs (planar or upon manifolds), with meaningful entities in nodes linked by edges.

Semantics got prominence within Software Engineering with the claim by Frederick Brooks in his books [1] [2] that "Conceptual Integrity is the most important consideration for software system design". This has been followed by recent research, e.g. by Jackson [8] and Exman. Semantics within Knowledge Engineering is prevalent since classical AI research, somewhat eclipsed by Deep Learning. Ontologies (e.g. the Protégé tool) are an important facet of it.

DOI: 10.18293/SEKE2019-213

The Algebraic Software Engineering aspect, recognizing the importance of Software mathematical theory, e.g. that Linear Software Models [3], [4] gradually gains traction. The Algebraic Knowledge aspect, recognizing Deep Learning's applied surge.

II. Related Work

A. Deep Learning for/by Software Engineering

Relevant neural networks are Recurrent Neural Networks (RNN), proposed in 1986 by Rumelhart, Hinton and Williams [14], and Long Short-Term Memory (LSTM) a special kind of RNN, proposed by Hochreiter and Schmidhuber [7] in 1997.

Software Engineering applications of Deep Learning for higher abstraction levels include: program generation from user intention (Lili Mou et al. [12]); program comprehension, to generate comments to Java code. (Xing Hu et al. [20]); API functions extraction from annotated code snippets collected from GitHub (Xiaodong Gu et al. [19]); and software modeling for various tasks (Hoa Khanh Dam et al. [6]).

Practical tools deal with software Traceability (Jin Guo et al. [11]), and fixing of C program errors (Rahul Gupta et al. [5]). Wei Fu and Tim Menzies [17] combine classical AI with Deep Learning to shorten training tasks.

B. Algebraic Software Theory: Linear Software Models

Software composition algebraic theory formalizes Brooks' Conceptual Integrity idea. Software is a hierarchical system, where each level is represented by a Modularity Matrix [3], [4]. Matrix columns stand for structural units, object-oriented *classes*, and matrix rows for functional units, i.e. class *methods*.

Brooks' principles translated into linear algebra demand that all matrix column vectors be *linearly independent* and similarly all the row vectors be linearly independent, obtaining a square matrix. If vector subsets are disjoint to other subsets, the matrix displays a block-diagonal form, i.e. the modules are *orthogonal*.

Modularity matrices may have outliers coupling between modules. Spectral methods for the Modularity Matrix [3], or the respective Laplacian Matrix [4], resolve couplings. A Laplacian obtains the same modules as the Modularity Matrix. The Fiedler vector, fitting the lowest Laplacian non-zero eigenvalue, allows locating outliers and splitting of too sparse modules.

III. Deep Software Learning: The Problems

Deep Software Learning has to assume that software is a collection of diverse assets: requirements, class diagrams, statecharts for design, a variety of graphs, models and code.

A. Software Problems to be Solved

Software problems dealt with by Deep Learning can be classified by their abstraction levels (Fig. 1). Higher abstraction activities, such as API Extraction, Program Generation and Program Comprehension depend on a suitable underlying ***software modeling***. Modeling is high-level abstraction, since many activities involve ***translation between models***.

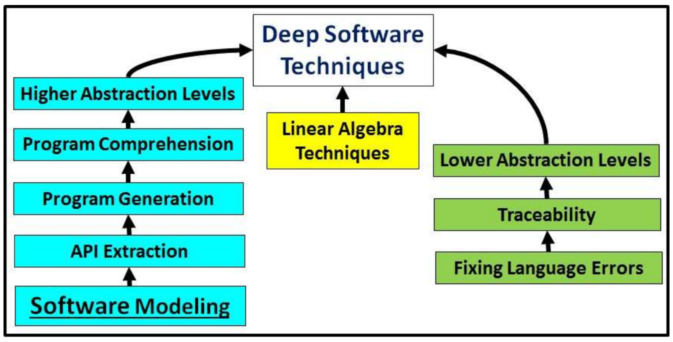

Figure 1. Deep Learning to Software Engineering applications, classified by abstraction levels. In between, the essential Linear Algebra Techniques.

Even lower abstraction level activities, such as correcting program errors, as done in DeepFix [5] need modeling. Every programmer has experience with accumulated bugs that result from misinterpretation (by the compiler!) of only a few bugs.

Linear algebra techniques are essential for Deep Learning. Richard Wei et al. [18], in their Compiler Infrastructure for Deep Learning, emphasize linear algebra representation in their system, such as a first class tensor type, algebraic operators such as "dot" and "tanh" (a typical sigmoid-like activation function).

B. Software Characterization: Sequential but not Consecutive, and Structured

Often program feature pairs are sequential but not appearing in each other neighborhood. Examples are: left and right parentheses (or braces); open and close a file; Java try and catch. Code modeling is sequential, but not of consecutive tokens. Dealing with such sequences, demands specific Deep Learning (DL) networks. Software also has more complex structures such as abstract syntax trees, dependency graphs, design diagrams. Sequences are not enough for software DL.

IV. Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are dedicated to continuous data, such as text, audio and video. It reuses previous information about a word in a sentence or video frame to understand the next word or frame. RNNs handle tasks, like free text comprehension and text sequences generation from scratch. To reuse previous information to handle the next input, RNN has recourse to persistence loops. RNNs have difficulty applying prediction of long-distance natural language dependencies. Most translation, voice recognition, and image classification successes, are due to LSTM a special class of RNNs.

A. LSTM = Long Short-Term Memory

LSTMs remember information for long periods. Forgetting must be explicitly handled. LSTMs also have a loop structure of repeating neural network units. The main difference of a typical LSTM unit from an RNN unit, is 4 layers instead of a single one. A hidden state Z is the key to LSTMs functioning. It has an input $\mathbf{z}_{t-1}$ from its predecessor, runs throughout the chain of (unrolled loop) units, affected by controlled interactions, and outputs $\mathbf{z}_t$ to its successor unit. Three gates (in Fig. 2) update the hidden state Z in each cycle, filtering the output Y parts:

- ***Forget gate*** f_t – sigmoid taking $\mathbf{y}_{t-1}$ and $\mathbf{x}_t$ and producing a number between 0 and 1 for each $\mathbf{z}_{t-1}$ value; the part of the hidden state Z to (fully or partially) discard;
- ***Input gate*** – has 2 layers: *sigmoid* $\boldsymbol{i}_t$ sets which values will be updated; *tanh* creates a new vector of values $\hat{z}_t$, multiplied by the sigmoid output giving candidate values actually added to Z: $z_t = f_t * z_{t-1} + i_t * \hat{z}_t$
- ***Output gate*** – sigmoid $\boldsymbol{o}_t$ filters what will be the output and what remains the Z output: *tanh* normalizes $\mathbf{z}_t$ values between -1 and +1, then multiplied by the sigmoid $\boldsymbol{o}_t$:
  $\mathbf{y}_t = \boldsymbol{o}_t * \tanh(\mathbf{z}_t)$.

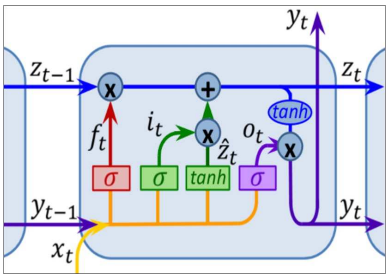

Figure 2. LSTM Deep Learning Network Schematic diagram - The upper (blue) horizontal line is the Z hidden state, with input $\mathbf{z}_{t-1}$ and hidden output $\mathbf{z}_t$. The lower horizontal line passes the output $\mathbf{y}_t$ between consecutive units. The three vertical gates are: the (red) *forget gate* $\boldsymbol{f}_t$; the (green) *input gate* $\boldsymbol{i}_t$ with two parallel layers ($\boldsymbol{\sigma}$ and ***tanh***); the (violet) output gate $\boldsymbol{o}_t$. (Color online)

A realistic LSTM test, keeping track of long-range attributes, takes a large source code and randomly concatenates it into a long file (e.g. Linux Kernel [9] about $6*10^6$ characters). The LSTM network is trained to predict special source code symbols, as whitespace, quotation marks and brackets. To predict a *close* bracket, the model must be aware of a matching *open* bracket, appearing many time steps ago. LSTM performs much better than other learning models due to an additional state feature (hidden state Z) explained above, in contrast to the standard RNN single hidden state. Results can be improved with an attention mechanism [21] or bidirectional-LSTM [10].

V. Deep Software Learning in Practice

The Deep Learning (DL) mechanism in the Xing Hu [20] work deals with sequential code, and Abstract Syntax Tree (AST) structure, translating sequence-to-sequence, code to comments. The Encoder/Decoder (both LSTMs) architecture uses one Encoder pass to traverse the AST, learning the code relevant to the comments. A second Encoder pass composes the gathered comment pieces into meaningful sentences.

Rare tokens clutter a vocabulary with single instances. Practical projects exchange numerals/strings by generic tokens <NUM> and <STR>, and rare words by “unknowns” <UNK>.

VI. Discussion

Today’s software and knowledge (DL) theories have two important characteristics: 1- they heavily involve linear algebra; 2- the algebra is totally independent of concepts’ semantics. Remaining issues are: The algebraic software theory is up to now strictly linear, while Deep Learning involves non-linearity. Will there be a convergence also in this sense? The Laplacian matrix is central to the software theory, while not so prominent in Deep Learning; will it be important for Deep Learning too?

References

[1] F.P. Brooks, *The Mythical Man-Month, Essays on Software Engineering*, Anniversary Edition, Addison-Wesley, Boston, MA, USA, (1995).

[2] F. Brooks, *The Design of Design, Essays from a Computer Scientist*, Addison-Wesley, Boston, MA, USA, 2010.

[3] I. Exman, “Linear Software Models: Decoupled Modules from Modularity Eigenvectors”, Int. Journal on Software Engineering and Knowledge Engineering, vol. 25, pp. 1395-1426, October 2015. DOI: 10.1142/S0218194015500308

[4] I. Exman and R. Sakhnini, “Linear Software Models: Bipartite Isomorphism between Laplacian Eigenvectors and Modularity Matrix Eigenvectors”, Int. Journal of Software Engineering and Knowledge Engineering, Vol. 28, No 7, pp. 897-935, 2018. DOI: http://dx.doi.org/10.1142/S0218194018400107

[5] R. Gupta, S. Pal, A. Kanade and S. Shevade, “DeepFix: Fixing Common C Language Errors by Deep Learning”, Proc. 31st AAAI Conference, pp. 1345-1351, (2017).

[6] Hoa Khanh Dam, Truyen Tran and Trang Pham, “A deep language model for software code”, arXiv:1608.02715, (August 2016).

[7] S. Hochreiter and J. Schmidhuber, “Long short-term memory”, Neural Computation, Vol. 9, pp. 1735-1780, (1997).

[8] D. Jackson, “Conceptual Design of Software: A Research Agenda”, MIT-CSAIL-TR-2013-020, August 8, 2013.

[9] A.Karpathy, J. Johnson and L. Fei-Fei, “Visualizing and Understanding Recurrent Networks, https://arxiv.org/pdf/1506.02078.pdf.

[10] E. Kiperwasser and Y. Goldberg, “Simple and accurate dependency parsing using bidirectional LSTM feature representations”, Trans. Assoc. Computational Linguistics, Vol. 4, pp. 313-327, (2016).

[11] Jin Guo, Jinghui Chang and Jane Cleland-Huang, “Semantically Enhanced Software Traceability Using Deep Learning Techniques”, arXiv:1804.02438 (April 2018).

[12] Lili Mou, Rui Men, Ge Li, Lu Zhang and Zhi Jin, “On End-to-End Program Generation from User Intention by Deep Neural Networks”, arXiv:1510.07211, (October 2015).

[13] M. Minsky and S. Papert, Perceptrons, MIT Press, Cambridge, MA, USA, 1969.

[14] D.E. Rumelhart, G.E. Hinton and R.J. Williams, “Learning internal representations by error propagation”, in Rumelhart and McClelland (eds.) *Parallel Distributed Processing*, vol. 1, pp. 318-362, MIT Press, Cambridge, MA, USA, (1986).

[15] C. E. Shannon, “Programming a Computer for Playing Chess”, Phil. Mag., Series 7, Vol. 41, 18 pages (March 1950).

[16] A.M. Turing, “Computing Machinery and Intelligence”, Mind, New Series, Vol. 59, pp. 433-460, (October 1950).

[17] Wei Fu and Tim Menzies, “Easy over Hard: A Case Study on Deep Learning”, ESEC/FSE’17, arXiv:1703.00133, (June 2017). DOI: https://dx.doi.org/10.1145/3106237.31052546

[18] R. Wei, L. Schwartz and V. Adve, “DLVM: A Modern Compiler Infrastructure for Deep Learning Systems, Workshop track ICLR 2018, arXiv:1711.03016, (February 2018).

[19] Xiaodong Gu, Hongyu Zhang, Dogmei Zhang and Sunghun Kim, “Deep API Learning”, Proc. FSE’16, arXiv:1605.08545 (2017). DOI: http://dx.doi.org/10.1145/1235

[20] Xing Hu, Ge Li, Xin Xia, David Lo and Zhi Jin, “Deep Code Comment Generation”, Proc. ICPC IEEE/ACM Int. Conf. Program Comprehension, 11 pages (May 2018). DOI: https://doi.org/10.475/123_4

[21] W. Yin, H. Schutze, B. Xiang and B. Zhou “Abcnn: Attention-based convolutional network for modeling sentence pairs”, Trans. Assoc. Computational Linguistics, Vol. 4, pp. 259-272 (2016).