DeepForge: An Open Source, Collaborative Environment for Reproducible Deep Learning

Brian Broll
Digital Reasoning
Franklin, TN 37067 USA
brian.broll-ctr@digitalreasoning.com

Jimmy Whitaker
Digital Reasoning
Franklin, TN 37067 USA
jimmy.whitaker@digitalreasoning.com

Abstract

In this paper we introduce DeepForge, a platform for deep learning designed to lower the barrier to entry and facilitate the rapid development of deep learning models while maintaining a high degree of transparency and interoperability. Utilizing a cloud-based infrastructure, DeepForge facilitates rapid development by promoting reproducibility, collaboration and remote execution of machine learning pipelines. DeepForge represents an interdisciplinary approach to facilitating deep learning development as it leverages the strengths of Model Integrated Computing to provide a powerful hybrid textual-visual programming platform beneficial to both novices and researchers.

1 Introduction

Deep learning has proven to be a very powerful machine learning approach in a variety of domains from image classification (9) to audio speech recognition (2). A significant contributor to the success of deep neural networks is their ability to model very complex functions; this enables neural network architectures to be applied to various domains and the models to be confirmed empirically. Due to the emphasis on empirical validation, it is very important that researchers and practitioners are equipped with the appropriate tooling to allow them to iterate quickly, work together and easily integrate the latest advancements in research into their own projects. The foundation of DeepForge, shown in Figure 1, is based on two main computing technologies, Torch 7 and Model Integrated Computing using WebGME.

Torch 7 is a scientific computing framework, written in Lua, which provides both a high degree of flexibility and highly optimized linear algebraic operations on both CPU and GPU (7). Along with providing flexibility and high performance, Torch provides an extensible layer based neural network library which supports a simple interface for creating custom layer definitions. This extensibility, flexibility and high performance makes Torch a valuable tool for developing neural network models including prototyping new neural network layers. Torch also has developed an active community which contributes many layer implementations and extra utilities to the Torch ecosystem. However, there is still a steep learning curve as well as a barrier to entry with deep learning development, making it difficult to quickly move from idea to prototype. Additionally, with increased flexibility for research, there is no standardization between experiments, making code difficult to reproduce and share.

Model Integrated Computing (MIC) is the technique of using models, or domain specific abstractions, for developing systems or applications (23) and was developed to aid in the rapid design and implementation of complex applications and systems. The Generic Modeling Environment (GME) is an open source MIC tool developed for creating domain specific modeling environments and has been effectively applied to a number of domains including embedded systems and mechatronics (8; 24; 20; 25; 5; 22; 11).

l0.75

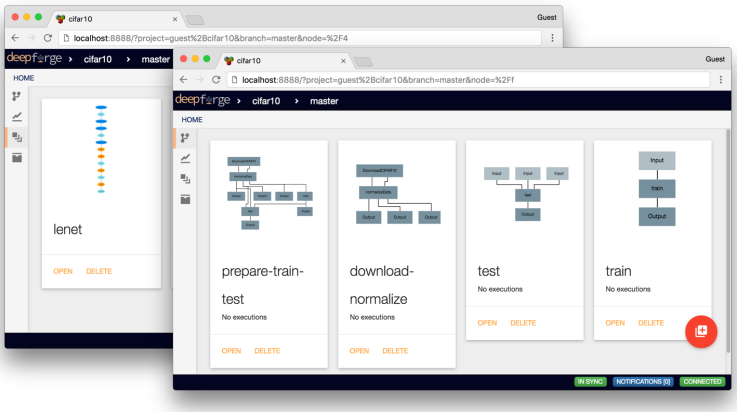

Figure 1: An example project in DeepForge

WebGME, the successor to GME, leverages a number of modern features such as a cloud-based infrastructure, integrated version control and real-time collaborative editing (16). WebGME also introduces a number of powerful modeling abstractions, such as prototypal inheritance and mixins, to improve its ability to model complex systems (15). WebGME has been used to improve development in a variety of domains from medical capsule robotics to space radiation (14; 4; 3; 18).

DeepForge is a development environment with deeply integrated domain specific modeling aspects created with WebGME. This enables DeepForge to leverage the strengths of model integrated computing to facilitate the rapid development of deep learning models and lower the barrier to entry for deep learning while also promoting the reproducibility of experiments. In doing so, we provide a powerful open source platform for deep learning, with aims to build a community around the platform in which users can contribute custom DeepForge extensions as well as seamlessly share and collaborate with one another.

The structure of the paper is as follows. In Section 2, we will present the conceptual framework for creating and executing machine learning tasks. Section 3 presents the design of the DeepForge platform. Section 4 provides a machine learning example in DeepForge and Section 5 provides a comparison with existing tools including Nvidia's DIGITS (19) and Google's TensorBoard (1).

2 Core Concepts

DeepForge presents four main concepts for testing and training machine learning models: *Pipelines*, *Operations*, *Executions* and *Jobs*.

Operations are atomic functions which accept one or more named inputs and return one or more named outputs. Both the input and output values are typed. Operation *attributes* can be defined for an operation at design time, providing adjustable parameters for the operation. At runtime, an operation's attributes are provided as constants to the operation. Operations can also define *references* which, like attributes, are specified at design time. Unlike attributes, a reference is a pointer to another artifact, such as a neural network architecture.

A *Pipeline* represents a machine learning task, such as model training, testing or data preprocessing composed of operations. Operations are composed by directing an output of a single operation, represented as a port, into an input of one or more other operations. That is, these operations are composed into acyclic data flow graphs. Pipelines can also contain *Input* and *Output* operations for representing input data to the pipeline and output data from the pipeline. That is, the *Input* operation is an initial node in a pipeline and *Output* operation is a terminal node.

Executing pipelines results in the creation of *Executions*. A pipeline's execution is an acyclic data flow graph that is isomorphic with the graph of the originating pipeline. This graph is created by converting

each of the pipeline's operations into a *Job* that corresponds to the original operation combined with run status and running metadata (such as images or plots generated during the execution).

As DeepForge focuses on training (deep) neural networks, it also utilizes two additional concepts: *Architectures* and *Layers*. As expected, an architecture represents a neural network architecture while a layer represents a neural network layer. The layers are connected in a directed acyclic graph forming the architecture. Similar to operations, layers can have attributes which are set at design time[1]. Unlike operation attributes, layer attributes can also be set to another layer or sequence of layers.

3 Platform

Leveraging the concepts presented in Section 2, DeepForge provides a web-based platform for deep learning with three principle design goals: development simplification, facilitating rapid development and reproducibility of experiments. DeepForge utilizes techniques in MIC along with intuitive, domain specific interfaces to lower the barrier to entry for deep learning. Rapid development is facilitated through promoting collaboration and interoperability with Torch7 as well as supporting the entire development cycle from the first iteration to the last execution. Finally, DeepForge leverages WebGME's version control system to provide deeply integrated versioning of both the code and the data throughout the entire development process. In this section we will discuss the DeepForge platform in more detail with an emphasis on how the design supports the design goals.

3.1 Development Simplification

DeepForge uses WebGME, a framework for creating domain specific modeling environments, to create a domain specific modeling language for neural networks and the concepts described in Section 2. This enables DeepForge to leverage the strengths of MIC, such as enforcing semantics of the domain and easily generating artifacts of different formats from the models, while also facilitating the development of other modern features including real-time collaborative editing and a deeply integrated version control system. DeepForge provides a hybrid textual-visual development environment for developing deep learning models. This allows users to leverage the strengths of a domain specific modeling environment when working at high levels of abstraction, such as designing pipelines, and utilize the precision of a textual programming environment when working at low levels of abstractions, such as implementing custom operations. This visual editor is not simply a diagram generated from the code but is an executable domain model and is always guaranteed to provide an up-to-date visual representation of the given pipeline or architecture.

Providing a visual interface for designing pipelines and architectures not only lowers the barrier to entry for developing deep learning models but also provides a more easily understandable diagram of the given structure. Lowering the barrier to entry reduces the amount of time it takes for a new user to be able to start developing and training neural network models. Designing pipelines and architectures with a visual interface can simplify complex machine learning tasks as the interface provides a diagram of the given pipeline or architecture.

Textual interfaces are provided for some of the more advanced features of DeepForge such as implementing custom operations and custom layers. The textual interface for implementing operations allows the user to utilize the power and flexibility of Torch and the ecosystem of supporting libraries and frameworks, such as Torchnet.

3.2 Rapid Development

Along with lowering the barrier to entry, DeepForge provides a number of modern features to facilitate rapid development of deep learning models. This is done primarily through promoting collaboration, reproducibility and easy execution in a distributed environment.

DeepForge provides a number of features promoting collaboration. Google Docs-style real-time collaborative editing allows users to simultaneously work together on a project. Version control is deeply integrated into DeepForge and user actions are automatically committed. This allows users to not only see the recent changes made by collaborators but also enables them to leverage branches to manage collaboration in large projects.

[1]Layer attributes are analogous to layer arguments in textual neural network implementations

Torch interoperability was another priority influencing the design of DeepForge. Promoting interoperability allows users to more easily incorporate existing Torch work into DeepForge as well as easily transition existing research using Torch into DeepForge. Maintaining interoperability and transparency enables the environment to be familiar to existing Torch users; this also makes skills learned in DeepForge, such as defining custom layers, transferable to vanilla Torch.

This interoperability with Torch enables DeepForge to easily integrate the existing cutting edge research and projects. Supporting generic Torch allows users to leverage the large number of frameworks and tools built for Torch, such as Torchnet (6). This includes not only supporting arbitrary Torch code in the operation but also supports automatically importing neural network architectures written in Torch. Importing architectures allows users to convert the Torch architecture definitions into editable models of the given architecture. As architectures can be exported to vanilla Torch, this interoperability greatly simplifies the task of incorporating and understanding complex architectures. To promote easily deploying pipelines to production, DeepForge also supports exporting pipelines to arbitrary formats by creating custom extensions; this includes exporting for use in production environments.

Supporting the complete development cycle of creating machine learning pipelines was paramount in DeepForge; supporting both the creation and execution of the machine learning pipelines not only greatly simplifies the user experience but also allows the environment to support developer iteration, multi-tasking and record the results of the given task. DeepForge supports the execution of machine learning pipelines on a distributed environment. This includes not only the ability to execute pipelines in a distributed environment but also powerful utilities for monitoring and improving upon existing pipelines. DeepForge provides real-time feedback from running jobs, including creating plots, viewing images or simply viewing the job's console output. To promote rapid development, DeepForge also enables the user to restart individual jobs within an execution and caches intermediate job results in an execution; this allows the user to reuse the outputs of unchanged jobs and avoid redundant computation.

3.3 Reproducibility

The final design goal in DeepForge is to provide easy reproducibility of experiments. This is achieved through the use of automatic versioning during development and as well as versioning both the code and the binary artifacts (such as data and trained models) for the given experiments. Automatic version control ensures that all changes and edits will be recorded in the history of the project and are reproducible. Versioning both the code and all associated binary artifacts, DeepForge is able to ensure that not only the historical versions of the code are reproducible for a given experiment but also any associated data and models.

Leveraging the version control API's provided by the WebGME framework, DeepForge is able to provide an automatic version control system specific to the deep learning domain. This includes automatically tagging commits when experiments are run, creating commits automatically complete with meaningful commit messages and committing intermediate results from machine learning pipelines to promote reproducibility and rapid iteration.

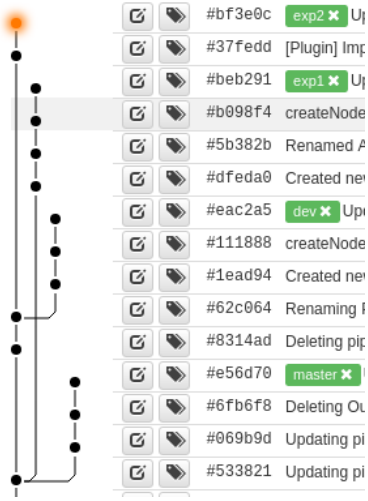

Figure 2: Viewing the commit history of a project multiple branches

4 Example

This section provides a simple, working example of how to train a deep convolutional network on the CIFAR-10 (13) dataset using DeepForge. In this example, we will retrieve training and test sets from AWS and then we will normalize the data. Next, we will train a deep neural network on the training set and then test it on the test set. Finally, we will store the trained neural network model and the test scores. We will first present the machine learning pipeline and the operations. Then we will present

the associated neural network architecture in the context of the architecture editor. Finally, we will examine an execution of the given pipeline and one of the corresponding jobs.

The entire pipeline for training this deep neural network model is shown in Figure 3. As discussed in Section 2, a *pipeline* is a collection of operations that are connected in a directed acyclic graph of *operations*. In Figure 3, the grey boxes represent the operations in the given pipeline and the connections represent the data flow through the given pipeline. In this case, there are six operations: *DownloadCIFAR10*, *NormalizeData*, *Train*, *Test* and two *Output* operations. *DownloadCIFAR10* downloads the CIFAR-10 testing and training datasets. After downloading the data, the two datasets are passed into the *NormalizeData* operation which normalizes the data.

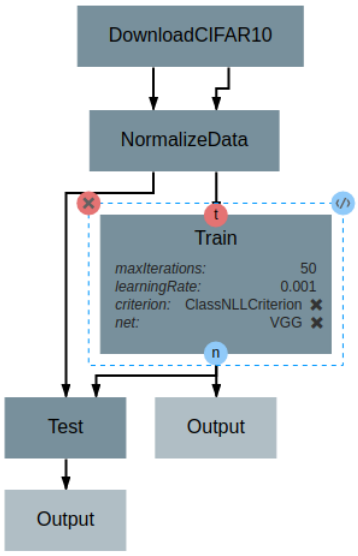

Figure 3: Pipeline to download and normalize the CIFAR-10 dataset then train and test a neural network model

Next, the normalized training data is passed to the *Train* operation. In Figure 3, the *Train* operation is selected; this results in the operation being expanded and shows the attributes and references defined for the given operation. The *Train* operation has two attributes, *maxIterations* and *learningRate*, and two references, *criterion* and *net*, defined. In this example, the learning rate has been set to 0.001 and the maximum number of iterations is set to 50. This operation is using the popular architecture *VGG* (21); this is specified by setting *net* to reference *VGG*. In this example, the *criterion* is set to *ClassNLLCriterion* provided in the neural network torch package.

After the *Train* operation, the trained model is then passed to the *Output* operation and the *Test* operation. The *Output* operation is a default operation which simply saves the data back to DeepForge for easy export and reuse as the input to other pipelines. The *Test* operation accepts testing data and a trained model and returns the classification accuracy for each of the provided classes. These accuracies are then passed to another *Output* operation, allowing them to be easily accessible within DeepForge.

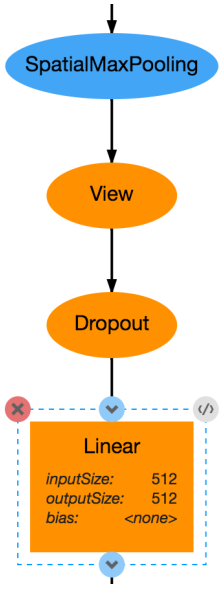

Figure 4: Editing the VGG architecture from Figure 3

Figure 4 shows a section of the *VGG* architecture used in the pipeline. In this section, there are four layers shown: *SpatialMaxPooling*, *View*, *Dropout* and *Linear*. These layers are all provided in the neural network package provided by Torch. In Figure 4, the linear layer is selected and is showing its configurable attributes: *inputSize*, *outputSize* and *bias*. This particular linear layer has 512 input neurons and output neurons with no bias. As the linear layer is provided by the neural network package, the implementation of this layer is not editable; visually, this is represented by the disabling of the button in the top right of the layer. When a custom layer is selected, the icon in the top right is light blue (similar to the selected operations) and allows the user to edit the Torch implementation of the layer or operation.

After creating the pipeline, we can execute it within DeepForge. As described in Section 2, this results in an execution being created from the pipeline. Executions share the same structure as their originating pipeline but are composed of jobs rather than operations. Executing the pipeline from Figure 3 results in the execution shown in Figure 5. Figure 5 shows a job for each of the operations from the originating pipeline. Each job is color coded according to its execution status. The first two jobs, *DownloadCIFAR10* and *NormalizeData*, have completed and the *Train* job is currently running. This is illustrated by the green and yellow colors, respectively. The remaining jobs are queued and will run as soon as their required data is available. As described in Section 3, detailed real-time output of each job can

be monitored during an execution. An example viewing the output of the *Train* job is provided in Figure 6.

Figure 6 shows an example of monitoring the *Train* job. When monitoring a job, the job implementation is provided along with any of the job's outputs. In this example, the *Train* job has two types of output: the console output and a plot of the training error (using the criterion function defined in Figure 3). In Figure 6, the implementation is provided on the left and the output is shown on the right. In this example, the graph is currently visible and providing real-time feedback after each training iteration. The console output can be viewed by selecting the "Console" button beneath the graph.

One powerful aspect of this approach to job feedback lies in its flexibility; the real-time plotting API is provided in the operation implementation and enables users the flexibility to define the most appropriate feedback mechanisms for the given operation. This plotting API is also very simple and easy to incorporate into an operation's implementation. The relevant plotting logic from the *Train* operation is provided below.

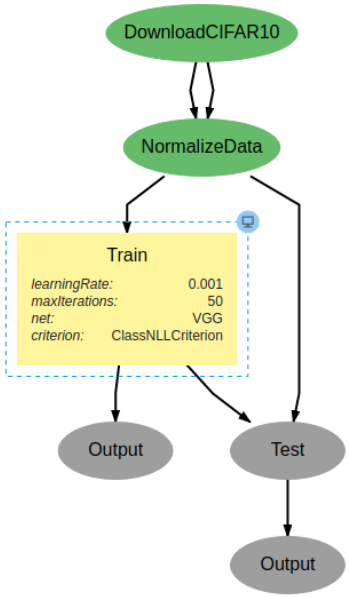

Figure 5: Execution of the pipeline in Figure 3

```
graph = DeepForge.Graph('Training Error')
errLine = graph:line('error')

trainer.hookIteration =
    function(t, iter, currentErr)
       errLine:add(iter, currentErr)
    end

trainer:train(trainset)
```

In this example, we are first creating the graph and an error line using the "DeepForge.Graph" constructor and the graph's "line" method, respectively. The *trainer* variable is the stochastic gradient descent module provided in Torch; in defining a function for the "hookIteration", we are able to add a point to the error line where the x value is the iteration and the y value is the associated error metric. Finally, the trainer invokes "train" to perform stochastic gradient descent on the neural network model and plot the error metric after each iteration.

Figure 6: Monitoring *Train* execution with real-time graph feedback

5 Comparison with Other Tools

While there are a handful of visual environments for data science developers, such as RapidMiner (17) and Knime (12), as well as more recent tools tailored to deep learning development, such as Nvidia's DIGITS (19) and Google's TensorBoard (1), DeepForge has significant differences. Unlike the existing data science developer tools, DeepForge primarily addresses the challenges of developing deep learning models while still providing the user a high degree of low-level flexibility, such as implementing custom layers and operations using Torch7. This flexibility allows DeepForge to be valuable not only to basic use cases but also to the more advanced use cases that come up in research. This enables DeepForge to be a useful platform for both researchers and industry, simplifing the task of incorporating the latest research into production environments. DeepForge provides a much higher degree of customizability and flexibility when creating neural networks than these generic data science tools.

Both DIGITS and TensorBoard provide support for the execution of machine learning tasks and provide visualization utilities to aid in the creation of the given tasks. They also both provide a web-based user interface for monitoring these executions and performing some model introspection such as visualizing weights of trained layers. Despite having visualization functionality, both tools require that the actual tasks are written in a textual language (or, for DIGITS, optionally using Caffe prototext). Both tools provide very useful capabilities for running and monitoring a machine learning task but provide little support for the actual process of creating and defining the machine learning task initially.

Unlike existing deep learning platforms, DeepForge also focuses on facilitating reproducibility and collaboration throughout the entire development process. By providing capabilities such as real-time collaboration and automatic version control, DeepForge supports the development process from the very beginning. DeepForge also provides a simple, yet powerful conceptual framework for the creation, execution and management of machine learning tasks. Providing not just visualization utilities but actually a visual editor for neural network architectures and machine learning pipelines lowers the barrier to entry for working with deep learning models while also providing an intuitive representation which can still benefit experienced users. As DeepForge still provides remote execution, real time feedback and visualization capabilities, it provides a platform that supports the developer from initial experimentation through to the iteration and fine tuning of the developed models.

6 Conclusion

This paper presented DeepForge, a robust development platform for deep learning with a powerful underlying conceptual model. DeepForge utilizes the strengths of model integrated computing, including model analysis and manipulation, to develop a deep learning platform that both provides an intuitive visual interface enforcing domain semantics as well as providing portable models which can be exported to various deployment formats. Furthermore, DeepForge has been designed to facilitate collaboration, reproducibility and extensibility. This includes supporting real-time collaborative editing, integrated version control and the remote execution of machine learning pipelines.

Future work includes integration of publicly available datasets such as OpenML and Kaggle as well as creating a registry for hosting operation definitions, architectures and trained models (26; 10). Adding data visualization support to the existing extension architecture is another planned feature to incorporate into DeepForge. This will enable the development and incorporation of third party custom model and data visualization utilities and will facilitate the incorporation of the latest neural network introspection and visualization techniques (27; 28). Developing an active community is particularly valuable as users can develop custom DeepForge extensions. Currently, extensions can be used to create custom export formats (simplifying deployment of machine learning pipelines); however, in the future, these will also include custom model introspection utilities as well as data visualization.

References

[1] ABADI, M., AGARWAL, A., BARHAM, P., BREVDO, E., CHEN, Z., CITRO, C., CORRADO, G. S., DAVIS, A., DEAN, J., DEVIN, M., ET AL. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467* (2016).

[2] AMODEI, D., ANUBHAI, R., BATTENBERG, E., CASE, C., CASPER, J., CATANZARO, B., CHEN, J., CHRZANOWSKI, M., COATES, A., DIAMOS, G., ELSEN, E., ENGEL, J., FAN, L., FOUGNER, C., HAN, T., HANNUN, A. Y., JUN, B., LEGRESLEY, P., LIN, L., NARANG, S., NG, A. Y., OZAIR, S., PRENGER, R., RAIMAN, J., SATHEESH, S., SEETAPUN, D., SENGUPTA, S., WANG, Y., WANG, Z., WANG, C., XIAO, B., YOGATAMA, D., ZHAN, J., AND ZHU, Z. Deep speech 2: End-to-end speech recognition in english and mandarin. *CoRR abs/1512.02595* (2015).

[3] AUSTIN, R. A. *A Radiation-Reliability Assurance Case using Goal Structuring Notation for a CubeSat Experiment*. PhD thesis, Vanderbilt University, 2016.

[4] BECCANI, M., TUNC, H., TADDESE, A., SUSILO, E., VÖLGYESI, P., LÉDECZI, A., AND VALDASTRI, P. Systematic design of medical capsule robots. *IEEE Design & Test 32*, 5 (2015), 98–108.

[5] CHILDS, A., GREENWALD, J., JUNG, G., HOOSIER, M., AND HATCLIFF, J. Calm and cadena: Metamodeling for component-based product-line development. *Computer 39*, 2 (2006), 42–50.

[6] COLLOBERT, R., DER MAATEN, L. V., AND JOULIN, A. Torchnet: An open-source platform for (deep) learning research. In *ICML Machine Learning Systems Workshop* (2016).

[7] COLLOBERT, R., KAVUKCUOGLU, K., AND FARABET, C. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop* (2011), no. EPFL-CONF-192376.

[8] CZARNECKI, K., BEDNASCH, T., UNGER, P., AND EISENECKER, U. Generative programming for embedded software: An industrial experience report. In *International Conference on Generative Programming and Component Engineering* (2002), Springer, pp. 156–172.

[9] HE, K., ZHANG, X., REN, S., AND SUN, J. Deep residual learning for image recognition. *CoRR abs/1512.03385* (2015).

[10] KAGGLE. Kaggle Datasets. `https://www.kaggle.com/datasets`. Accessed: 2017-02-1.

[11] KE, X., AND SIERSZECKI, K. Generative programming for a component-based framework of distributed embedded systems. In *Proceedings of the 6th OOPSLA Workshop on Domain-Specific Modeling (DSM'06)* (2006), pp. 113–122.

[12] KNIME. `https://knime.org`. Accessed: 2017-02-1.

[13] KRIZHEVSKY, A., AND HINTON, G. Learning multiple layers of features from tiny images.

[14] KUMAR, P. S., EMFINGER, W., KARSAI, G., WATKINS, D., GASSER, B., AND ANILKUMAR, A. Rosmod: a toolsuite for modeling, generating, deploying, and managing distributed real-time component-based software using ros. *Electronics 5*, 3 (2016), 53.

[15] LATTMANN, Z., KECSKÉS, T., MEIJER, P., KARSAI, G., VÖLGYESI, P., AND LÉDECZI, Á. Abstractions for modeling complex systems. In *International Symposium on Leveraging Applications of Formal Methods* (2016), Springer, pp. 68–79.

[16] MARÓTI, M., KECSKÉS, T., KERESKÉNYI, R., BROLL, B., VÖLGYESI, P., JURÁCZ, L., AND LÉDECZI, Á. Next generation (meta) modeling: Web-and cloud-based collaborative tool infrastructure.

[17] MIERSWA, I., WURST, M., KLINKENBERG, R., SCHOLZ, M., AND EULER, T. Yale: Rapid prototyping for complex data mining tasks. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining* (2006), ACM, pp. 935–940.

[18] NEEMA, H., SZTIPANOVITS, J., BURNS, M., AND GRIFFOR, E. C2wt-te: A model-based open platform for integrated simulations of transactive smart grids. In *Modeling and Simulation of Cyber-Physical Energy Systems (MSCPES), 2016 Workshop on* (2016), IEEE, pp. 1–6.

[19] NVIDIA. Nvidia DIGITS: Interactive deep learning gpu training system. `https://developer.nvidia.com/digits`. Accessed: 2017-02-1.

[20] ÖZGEN, C. *Transforming Mission Space Models To Executable Simulation Models*. PhD thesis, Middle East Technical University, 2011.

[21] SIMONYAN, K., AND ZISSERMAN, A. Very deep convolutional networks for large-scale image recognition. *CoRR abs/1409.1556* (2014).

[22] SPRINKLE, J. Generative components for hybrid systems tools. *Journal of Object Technology 4*, 3 (2004), 33–38.

[23] SPRINKLE, J. Model-integrated computing. *IEEE potentials 23*, 1 (2004), 28–30.

[24] STANKOVIC, J. A., ZHU, R., POORNALINGAM, R., LU, C., YU, Z., HUMPHREY, M., AND ELLIS, B. Vest: An aspect-based composition tool for real-time systems. In *Real-Time and Embedded Technology and Applications Symposium, 2003. Proceedings. The 9th IEEE* (2003), IEEE, pp. 58–69.

[25] THRAMBOULIDIS, K. Model-integrated mechatronics-toward a new paradigm in the development of manufacturing systems. *IEEE Transactions on Industrial Informatics 1*, 1 (2005), 54–61.

[26] VAN RIJN, J. N., BISCHL, B., TORGO, L., GAO, B., UMAASHANKAR, V., FISCHER, S., WINTER, P., WISWEDEL, B., BERTHOLD, M. R., AND VANSCHOREN, J. Openml: A collaborative science platform. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases* (2013), Springer, pp. 645–649.

[27] YOSINSKI, J., CLUNE, J., NGUYEN, A., FUCHS, T., AND LIPSON, H. Understanding neural networks through deep visualization. *arXiv preprint arXiv:1506.06579* (2015).

[28] ZEILER, M. D., AND FERGUS, R. Visualizing and understanding convolutional networks. In *European conference on computer vision* (2014), Springer, pp. 818–833.