AUTOMATED END-TO-END MANAGEMENT OF THE DEEP LEARNING LIFECYCLE

A Dissertation
IN
Computer Science

Presented to the Faculty of the University
of Missouri–Kansas City in partial fulfillment of
the requirements for the degree

DOCTORATE OF PHILOSOPHY

by
GHARIB GHARIBI

M. Sc., University of Missouri-Kansas City, MO, USA, 2016

Kansas City, Missouri
2020

AUTOMATED END-TO-END MANAGEMENT OF THE DEEP LEARNING LIFECYCLE

Gharib Gharibi, Candidate for the Doctorate of Philosophy Degree

University of Missouri–Kansas City, 2020

ABSTRACT

Deep learning has improved the state-of-the-art results in an ever-growing number of domains. This success heavily relies on the development of deep learning models–an experimental, iterative process that produces tens to hundreds of models before arriving at a satisfactory result. While there has been a surge in the number of tools and frameworks that aim at facilitating deep learning, the process of managing the models and their artifacts is still surprisingly challenging and time-consuming. Existing model-management solutions are either tailored for commercial platforms or require significant code changes. Moreover, most of the existing solutions address a single phase of the modeling lifecycle, such as experiment monitoring, while ignoring other essential tasks, such as model sharing and deployment. In this dissertation, we present a software system to facilitate and accelerate the deep learning lifecycle, named ModelKB. ModelKB can *automatically* manage the modeling lifecycle end-to-end, including (1) monitoring and tracking experiments; (2) visualizing, searching for, and comparing models and experiments; (3) deploying models locally and on the cloud; and (4) sharing and publishing trained models. Our system also provides a stepping-stone for enhanced reproducibility. ModelKB currently supports TensorFlow 2.0, Keras, and PyTorch, and it can be extended to other deep learning frameworks easily. A video demo is available at `https://youtu.be/XWiJpSM_jvA`.

Moreover, we study static call graphs to form a stepping-stone to facilitate the *comprehension* of the overall lifecycle implementation (i.e., source code). Specifically, we introduce Code2Graph to facilitate the exploration and tracking of the implementation and its changes over time. Code2Graph is used to construct and visualize the call graph of a software codebase. We evaluate the functionality by analyzing and studying real software systems throughout their entire lifespan. The tool, evaluation results, and a video demo are available at `https://goo.gl/8edZ64`.

Finally, we demonstrate a software system that brings together the contributions mentioned above to build a robust, open-collaborative platform for deep learning applications in the health domain, named Medl.AI. Medl.AI enables researchers and healthcare professionals to easily and efficiently: explore, share, reuse, and discuss deep learning models specific to the medical domain. We present six illustrative deep learning medical applications using Medl.AI. We conduct an online survey to assess the feasibility and benefits of Medl.AI. The user study suggests that Medl.AI provides a promising solution to open collaborative research and applications. Our live website is currently available at `http://medl.ai`.

APPROVAL PAGE

The faculty listed below, appointed by the Dean of the School of Computing and Engineering, have examined a dissertation titled "Automated End-to-End Management of the Deep Learning Lifecycle," presented by Gharib Gharibi, candidate for the Doctorate of Philosophy degree, and hereby certify that in their opinion it is worthy of acceptance.

Supervisory Committee

Yugyung Lee, Ph.D., Committee Chair
Department of Computer Science & Electrical Engineering

Ghulam Chaudhry, Ph.D.
Department of Computer Science & Electrical Engineering

Sejun Song, Ph.D.
Department of Computer Science & Electrical Engineering

Raveen Rao, Ph.D.
Department of Computer Science & Electrical Engineering

Zhu Li, Ph.D.
Department of Computer Science & Electrical Engineering

Bryan Boots, M.B.A.
Henry W. Bloch School of Management

CONTENTS

Chapter

ILLUSTRATIONS

TABLES

Tables Page

ACKNOWLEDGEMENTS

This work would have never seen the sunlight without the help of many people. Thank you all! Mom, thank you for teaching me to be who I am today. Dad, I appreciate every single word you had ever taught me. Thank you for your continuous support; you are my hero. Brother, I am pleased to have you in my life. Keep up the excellent work. Grandma and Grandpa, I am grateful for raising me the way you did. I wish you all could be here to celebrate this with me.

Dr. Yugyung Lee, thank you for teaching me how to be a better scholar and a better person. Thank you for your continuous motivation and trust. I appreciate all the time and effort you have invested in me. Prof. Ghulam Chaudhry, thank you for the tremendous support you have given to me. Because of you, I had the opportunity to teach at college and be involved on campus.

The late Sandy Gault, I am thankful for your tremendous efforts in making UMKC a better place for international students. It was an honor to be part of your team.

Dr. Praveen Rao, Dr. Zhu Li, Sejun Song, and Bryan Boots, thank you for your time and effort in being on my Ph.D. Committee. I appreciate all of your help and support. Sirisha Rella, Vijay Walunj, Rakan Alanazi, Duy Ho, and others who worked with me and supported me, thank you for being there. I am grateful for your technical support throughout the project. I wish you a happy and successful life.

Finally, my sincere thanks go to the School of Computing and Engineering faculty, staff, and friends. Thank you all. I had amazing time at UMKC!

CHAPTER 1

INTRODUCTION

Deep learning [29, 54], a subfield of Machine Learning, has improved the state-of-the-art results in an ever-growing number of domains such as computer vision [37, 56, 76, 93], speech recognition [36, 79], and reinforcement learning [8, 89]. A *Deep learning model*, also known as a *Deep Neural Network* (DNN) can be defined as a mapping function composed of a large number of simple but nonlinear processing layers that map raw input data (e.g., images) to the desired output (e.g., classification labels) by learning hierarchical representations from the data. Unlike traditional machine learning algorithms (e.g., Support Vector Machines) that require a thorough feature engineering phase before training the model [34], deep learning is an end-to-end learning approach that can automatically learn the features from the input data [29]. While this key advantage of deep learning plays a significant role in its wide-spread and adoption, it is also considered one of the main reasons that hinder the understandability and interpretation of deep learning models. Consequently, it is not uncommon to refer to DNNs as *black box* models/functions [10, 99, 23].

Due to limited knowledge on the meaning of well-trained DNN parameters, training deep learning models is an ad-hoc, search task that goes through tens to hundreds of iterations (*experiments*) before arriving at a satisfactory result. Experiments involve exploring different architectures, data transformations, and hyperparameters (e.g., optimization algorithm and learning rate). Each experiment produces a large number of artifacts such as learned parameters (i.e., weights and biases), validation scores (e.g., loss and accuracy), and source code. Thus, the final selection of an efficient model heavily relies on comparing the experiments' artifacts since the best model is not necessarily the last trained model.

Figure 1 illustrates the typical deep learning lifecycle, which we summarize in two phases: Experiments and Deployment. The Experiments phase focuses on preparing the data, creating or reusing an existing architecture, training the model, and validating its performance. The Deployment phase focuses on deploying the model in a production environment. Not only does the deep learning lifecycle involves several phases, but it also involves several members with different backgrounds and technical expertise, such as data scientists, machine learning engineers, and software developers. For example, given a prediction task with a well-defined objective, the data scientist collects and prepares the data, the machine learning engineer focuses on developing and training the model, and the software engineer works on deploying the trained model in a production environment (e.g., software system, cloud service, edge device).

Therefore, in addition to the common development challenges known to the software engineering community, e.g., code review and debugging [27], the heuristic and iterative nature of deep learning presents a new set of development challenges–particularly **managing the large number of experiments and artifacts throughout their lifecycle** [86, 61, 49, 84, 99]. The management of the deep learning lifecycle involves tracking,

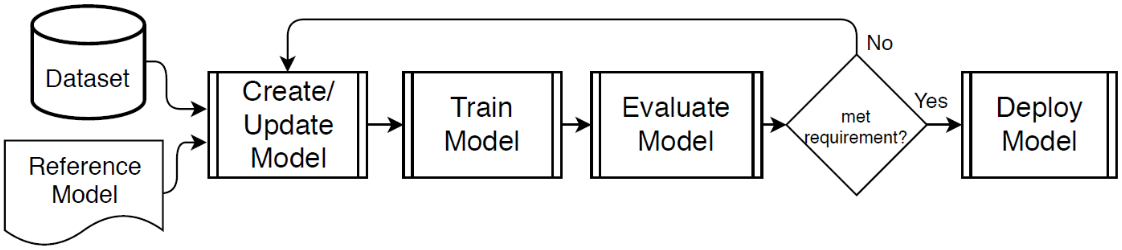
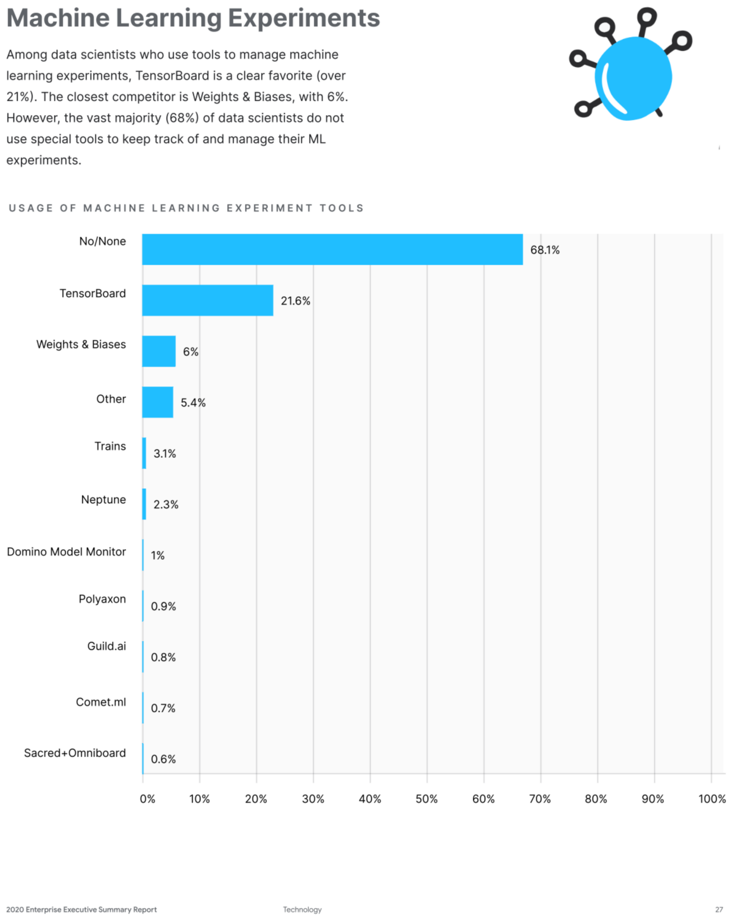


Figure 1: Deep Learning Development Lifecycle. Adopted from [61].

organizing, visualizing, and facilitating the essential tasks of the lifecycle (i.e., training, evaluation, and deployment) and then transferring this data into information for sharing, analysis, and reproducibility.

The recent surge in deep learning research and applications has led to the development of several systems that focus on training deep learning models, such as Theano [6], TensorFlow by Google [57], CNTK by Microsoft [87], PyTorch by Facebook [70], and Keras [13]. Note that Keras–one of the most popular libraries for building neural networks–is a set of high-level APIs, written in Python, capable of running on top of TensorFlow, CNTK, or Theano. These systems focus on the Experiment phase (i.e., training and validation) while largely ignore model management issues. Without a proper approach and tool support to address the lifecycle management challenges, deep learning practitioners spend lots of time and effort to track their experiments, log the used parameters, version their models, and visualize their results–to name a few. Not only is manual management expensive, time-consuming, and inefficient, but it also hinders subsequent tasks–especially model sharing and reproducibility. The recent Kaggle survey on State of Machine Learning and Data Science 2020, reveals that more than 68% of machine



Machine Learning Experiments

Among data scientists who use tools to manage machine learning experiments, TensorBoard is a clear favorite (over 21%). The closest competitor is Weights & Biases, with 6%. However, the vast majority (68%) of data scientists do not use special tools to keep track of and manage their ML experiments.

USAGE OF MACHINE LEARNING EXPERIMENT TOOLS

2020 Enterprise Executive Summary Report Technology 27

Figure 2: Usage of Machine Learning experiment tools [42].

learning practitioners still do not use tracking tools [42] (see Figure 2).

As these challenges became more evident recently, several systems from academia and industry emerged to address different aspects of the management challenges. Examples from academia include ModelDB [100], ModelHub [60], ProvDB [59], [85], Ground [38], and [50]. While most of the work in academia is often made available as open-source systems, these systems either address a single phase of the modeling lifecycle, e.g., tracking experiments, or require significant code changes to instrument the source code, which adds extra overhead in the modeling lifecycle.

The industry has also realized the importance of machine learning management systems, which led to the emergence of several systems. Famous systems include FBLearner Flow for PyTorch [21], TensorBoard for TensorFlow [30], Digits by Nvidia [69], and Michelangelo by Uber [96]. However, these systems restrict the user to a specific framework, processing pipelines, and deployment options. Other management systems include software systems that recently emerged from industry startups that realized the critical need for management systems, such as CometML (`www.comet.ml`), Weights and Biases (`www.wandb.com`), Neptune (`www.neptune.ml`), and MLFlow [108]. Overall, these systems often address a particular phase of the modeling lifecycle or require code changes to manage the modeling lifecycle.

To this end, we present a unified approach for managing the deep learning lifecycle end-to-end. We equip our approach with a software system for automated model management, named ModelKB (Model Knowledge Base). Our overarching goal is to facilitate and accelerate the modeling lifecycle with minimal user intervention. The novelty of our approach crystallizes in unifying the modeling lifecycle in a single system and utilizing the automatically extracted metadata to automate subsequent tasks, including deployment, sharing, and reproducibility. The main objectives are to monitor models and track their evolution, facilitate reproducibility, accelerate deployment, and enhance the collaboration process, i.e., model sharing and publishing. This allows data scientists to focus on the modeling process without having to worry about managing their experiments or be restricted to a specific modeling framework.

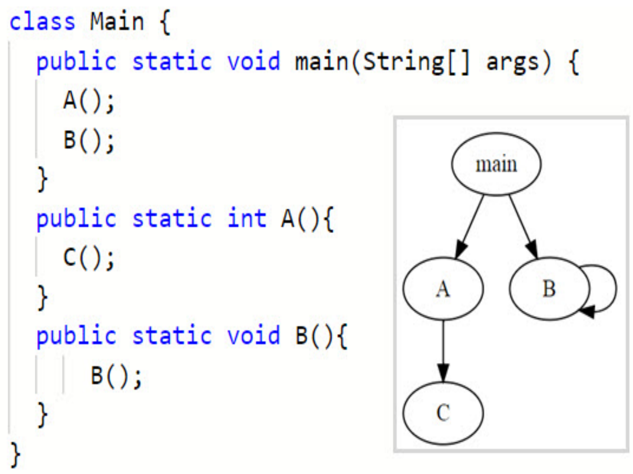
Our main contribution is a unified approach to manage the deep learning lifecycle equipped with a software system–ModelKB. ModelKB can (1) automatically extract and monitor the model's metadata and experiments' artifacts; (2) visualize, query, and compare experiments; (3) semi-automatically deploy models locally and on the cloud for casual inference tasks; and (4) reproduce models when needed through model sharing and publishing. ModelKB is a stand-alone Python library. ModelKB also aims at providing a cloud-based repository for publishing and exploring trained models, similar to sharing source code at GitHub. Note that ModelKB in itself is not a modeling framework, rather a complementary system that can automatically manage experiments in their native framework, including TensorFlow 2.0, Keras, and PyTorch. Moreover, we provide a toolkit, made of two software tools Code2Graph and GraphEvo, that aim to facilitate the comprehension of the experiments' codebase and their evolution.

Automatic extraction and tracking of the metadata is achieved using Callbacks. We provide customized Callbacks to collect metadata about each experiment. Metadata refers to all data that governs the learning task, including hyperparameters (e.g., learning rate), parameters (e.g., weights), and context data (e.g., required libraries). Once the metadata is automatically extracted, we use this data to automatically generate source code for deployment, sharing, and reproducibility. We also provide a GUI to monitor the progress throughout the lifecycle and allow users to explore and analyze their experiments. This is also beneficial to provide insights for future training tasks (e.g., what optimization algorithms work best for voice data). Moreover, our approach provides the means to run causal inference tasks using the deployed models. This allows deep learning practitioners to test their models on-the-fly using a web-based interface without having to code the prediction function. Thus, our system can facilitate the overall modeling lifecycle from training to deployment and sharing, which are otherwise expensive and time-consuming tasks.

To assess the usefulness of ModelKB in managing the overall modeling lifecycle, we conducted a user study followed by a survey to collect user's feedback. In all cases, ModelKB was used to manage the modeling lifecycle, including monitoring the experiments through their evolution, deploy particular trained models locally and remotely, share models and reproduce them, and run casual inference tasks on the deployed models. We also assessed the overhead in execution time using ModelKB, which is negligible compared to the long periods required for training a deep learning model.

In order to facilitate and enhance the comprehension of the overall system implementation and its evolution, we developed several tools that utilize static call-graphs to visualize and quantify software structure and its evolution. The goal is to assess software developers and data scientists to easily understand the overall structure of the software system and its evolution. We study software evolution based on its call graphs in a general terms, but our ultimate goal in this dissertation is to facilitate the understanding of the deep learning codebase among different experiments.

Static call graphs (see Figure 3) have been proven to assist software developers in understanding the overall system interactions and structure. For example, the tool in [101] can detect and visualize the static interactions between the classes of software systems



```
class Main {
  public static void main(String[] args) {
    A();
    B();
  }
  public static int A(){
    C();
  }
  public static void B(){
      B();
  }
}
```

Figure 3: An example of a Java code snippet and its corresponding call graph

written in Java. Similar to the existing methods and tools in this domain (see Chapter 4), our work aims at constructing and visualizing call graphs. However, unlike existing tools, our work uses *static call graphs to visualize software evolution in a monolithic color-coded graph and graph metrics to quantify software change*. Our user study prove that visualizing code evolution using colored graphs can enhance the comprehension of the source code and its evolution.

Finally, we put together our aforementioned contributions and demonstrate how a real software system can benefit from our platform. Specifically, we introduce a novel web-based collaborative platform, named *Medl.AI*, that allows physicians to easily explore, share, and reuse deep learning models specific for medical applications. In addition, the platform allows running online predictions using the hosted models and provides a forum-like service to share and discuss prediction results to encourage further studies and applications. Medl.AI is equipped with automated vulnerability tests to evaluate the security and privacy of the trained models before sharing them publicly. Our overarching goal is to foster networked science, research, and applications, which focus on the exchange and reuse of deep learning models built for medical applications with a strong emphasis on security and privacy.

1.0.1 Dissertation structure

The rest of this dissertation is organized in Chapters. Chapter 2 introduces the background and challenges in the field of deep-learning lifecycle management. Chapter 3 explains in detail the underlying methodology of our approach and tool, ModelKB, and its evaluation. Then, we introduce our approach towards program comprehension in Chapter 4. Chapter 5 wraps together our contributions in a real software system and demonstrates the direct and indirect benefits of our platform. We finally conclude our dissertation and discuss our future work in Chapter 6.

CHAPTER 2

BACKGROUND AND CHALLENGES

This section introduces the context of our research work, including a brief introduction to deep learning, its modeling lifecycle, and discusses the modeling challenges that motivate our work–the absence of a proper model management system.

2.1 Deep Learning

Deep learning can be defined as a mapping function composed of simple but non-linear processing layers that map raw input data (e.g., images) to the desired output (e.g., classification labels) by learning hierarchical representations of the data in each layer. The mapping process takes place in particular transformation layers, called *hidden layers*, which receive weighted input, from the *input layer*, transform it using nonlinear activation functions (e.g., ReLU), and then pass these values to the next hidden layers until the last layer in the structure, i.e. *output layer*, which outputs the final results. The organization of these layers and their connections are referred to as *Deep Neural Network (DNN) Architecture*. DNNs often consist of tens to hundreds of transformation layers, hence the name *deep learning*. A deep learning model refers to the combination of network architecture, the learned parameters (e.g., weights), and the configuration hyperparameters (e.g., learning rate). A key technical advantage of deep learning is that it can automatically learn the features of the input data and map them to the desired output using large amounts of

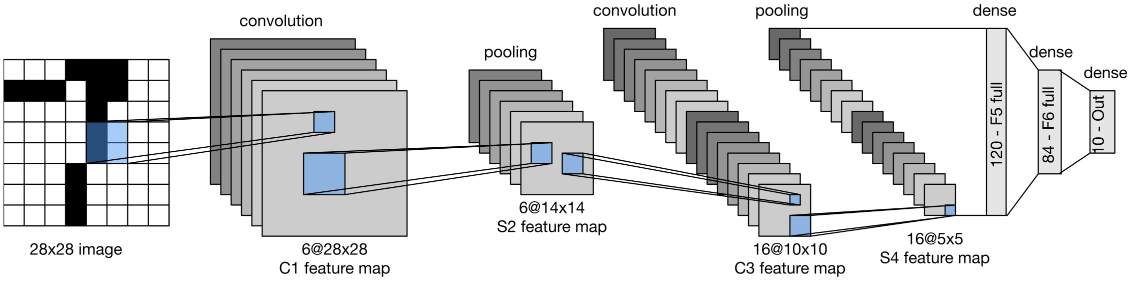


Figure 4: Example of a convolutional DNN, LeNet-5, [53]. The input is an image of hand-written digit (0-9), the output is a probability over 10 possible outcomes (classes), which predicts the digit written on the image. Diagram credit: [109].

data without human intervention (*feature engineering* [34]), which makes deep learning an end-to-end learning approach that requires less domain knowledge. Nevertheless, understanding and interpreting the learned model is still a challenging comprehension task, and it is an active research field [65, 95, 12].

Figure 4 illustrates a sample of a classical convolutional neural network (CNN), called LeNet, which was developed by Yann LeCun to recognize hand-written digit images [55, 53]. CNNs are mostly used in computer vision applications [52, 47, 43, 107]. In addition to the typical layers in the DNN, CNN has special types of layers, such as *convolutional* and *pooling* layers. These layers can be organized in different architectures while each of the layers can have a distinct set of parameters (e.g., activation function, size). For example, LeNet has two convolutional layers, each followed by a pooling layer, and two fully-connected layers, Figure 4 illustrates the architecture of LeNet and its configuration. The number of layers, their configuration, the learning rate [83], and other parameters that drive the overall training process are known as *hyperparameters*. Optimizing the hyperparameters plays a critical success factor in training the model. For example, the famous

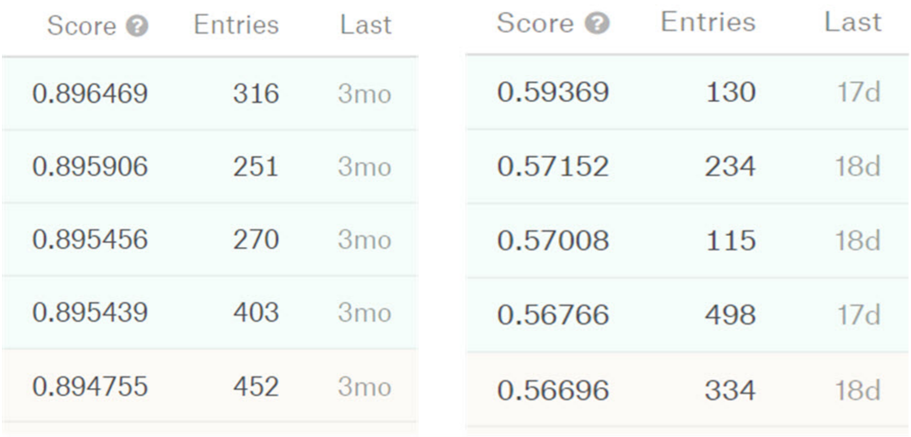
CNNs that won the ImageNet ILSVRC competition [80] each had a distinct set of hyperparameters and a unique architecture, including ResNet [37], AlexNet [47], VGG [90], and LeNet [55].

2.2 Deep Learning Modeling Lifecycle

Unlike traditional software development lifecycle that aims at meeting a set of well-defined functional and non-functional requirements, the process of training an efficient deep learning model aims at optimizing a specific metric (e.g., minimizing a loss function). Table 1 illustrates some of the core differences between traditional software development and machine learning development, which motivates the need for novel software tools that facilitate the machine learning lifecycle.

Table 1: Brief comparison between traditional software development and Machine Learning

	Traditional Software Development	**Machine Learning Development**
Goal	To meet a set of functional requirements	To optimize a metric (e.g., maximize accuracy)
Quality	Depends on written code (i.e., developers expertise)	Depends on data and used hyperparameters
Tools	Each entity, e.g., industry, uses a specific software stack	Involves experimenting with a wide range of libraries, platforms, and algorithms
Maintenance	Less frequent based on version basis	Requires continuous monitoring and maintenance (training on new samples)
Management	Uses mature tools, e.g., git	Often done manually using spreadsheets due to the lack of mature management tools
Deployment	Done by software developers who are familiar with the system	Real-world models are deployed in a collaborative manner among ML engineers and software engineers
Sharing	Uses mature tools, e.g., GitHub	Done manually or using git-like systems that are not built for model sharing
Reproducibility	Easy to achieve	Very challenging due to limited tools that track all involved hyperparameters and requirements



Score	Entries	Last
0.896469	316	3mo
0.895906	251	3mo
0.895456	270	3mo
0.895439	403	3mo
0.894755	452	3mo

Score	Entries	Last
0.59369	130	17d
0.57152	234	18d
0.57008	115	18d
0.56766	498	17d
0.56696	334	18d

Figure 5: An example illustrating the large number of models (*Entries*) submitted by top three winners of three randomly selected Kaggle competitions.

Additionally, due to limited knowledge of the meaning of well-trained models, the optimization process is carried out in an ad-hoc, trial-and-error approach by experimenting with different sets of hyperparameters, data transformations, and even different software libraries. For example, Figure 5 illustrates a real-world scenario of the large number of models produced by competitors solving real-world problems with deep learning at *www.kaggle.com*. Note that competitors upload the best model they produce only. Nevertheless, we notice that top competitors submit an average of 140 models before the deadline for a competition.

Therefore, we first need to understand modeling lifecycle in deep learning, which typically goes through the following phases (refer to Figure 1):

- *Data Preparation:* Collecting and preparing data in a form consumable by the model (i.e., applying data transformation and augmentation).

- *Model Search:* Searching for a model with a similar goal for reusability through the transfer learning process, rather than creating a new model from scratch. However, if no such model exists, a new model needs to be created.

- *Training and Evaluation:* Training and evaluating the model are the most expensive and time-consuming tasks. As mentioned before, training a deep learning model can be defined as an iterative search problem that goes through tens to hundreds of iterations before arriving at a satisfactory result.

- *Model Deployment:* After training a model, it is necessary to deploy it in a production environment (e.g., a software system) or expose its functionality through REST APIs.

- *Maintenance:* Deployed models still require continuous monitoring to identify defects and continuously maintain their performance. Here, the maintenance process often involves retraining the model on new data instances that were not included in the previous training dataset.

Therefore, the iterative nature and the large number of involved parameters and artifacts in deep learning projects introduce a new set of challenges–particularly managing the modeling lifecycle–which we present in the next subsection. Moreover, with the large number of produced experiments, the codebase becomes much larger and difficult to track manually over time. Thus, we introduce a new approach towards visualizing code evolution using static call-graphs (see Chapter 5).

2.3 Challenges of Deep Learning Modeling Lifecycle

The deep learning lifecycle involves experimenting with several neural architectures, datasets, learning algorithms, and configuration hyperparameters over hundreds of experiments. Metadata about these experiments and their artifacts play significant roles in informing the next set of experiments and identify best performing models. However, without a proper management system, experiments and their metadata are lost, and valuable time and resources are wasted–sometimes even reproducing the same results. Subsequently, the lack of an adequate management system for deep learning has exposed several challenges that hinder its overall lifecycle, including the following tasks:

- *Monitoring Experiments:* Training a deep learning model is a trial-and-error approach experimenting with different sets of hyperparameters, data transformations, and even different software libraries. Each experiment generates a rich set of artifacts that can be used to analyze, explore, and derive insights about future experiments (e.g., *what hyperparameters work best with similar datasets*). These artifacts play a critical role in comparing trained models when identifying the best-produced model. Currently, it is challenging to track each experiment, model evaluation, analyze abnormal behaviors, identify data provenance, and reason for the produced results. A large number of involved metadata elements makes it challenging to track experiments manually, and it's expensive and time-consuming to build efficient automated tracking systems. Monitoring experiments is at the core of addressing other deep learning management challenges.

- *Reproducibility:* Reproducing the same results across different experiments is a challenging task due to several factors, including random initialization of weights, dataset shuffling, and variations in the underlying framework (e.g., swapping between Theano and Tensorflow backends in Keras). Therefore, reproducing a specific model requires the availability of metadata about the training phase, including data transformations, architecture, and hyperparameters. Examples of real-world scenarios include reproducing production models that were initially trained offline in the lab, reproducing an older version of a model that is not available, and reproducing models that are published in research papers without their implementation.

- *Deployment:* Once a model is trained, it is often passed to a different team (e.g., software engineers) to integrate it into a software system or make it accessible through APIs. However, there is still no standard approach to transfer models from the training phase to a production system. Models could be deployed as REST services, Flask-based systems, mobile applications, or cloud services. Moreover, in order to run predictions using the deployed model (i.e., inference), ingested data need to go through the same preprocessing steps that were used on the training data. Deployed models also require continuous monitoring, evaluation, and retraining. Without a proper management system that facilitates experiment monitoring and reproducibility, the deployment process becomes even more challenging.

- Sharing: The lack of a centralized repository for trained models obstructs the sharing, reusability, and evolution of deep learning models. Currently, well-known models are often shared on the developer's website, or popular frameworks' websites such as pretrained PyTorch models [71], or raw files via traditional file repositories such as Model Zoo on GitHub [64]. While some repositories have recently emerged, such as ModelHub.AI [63], these repositories are manually curated by the owners or require significant manual efforts to share the trained models.

2.4 Related Work

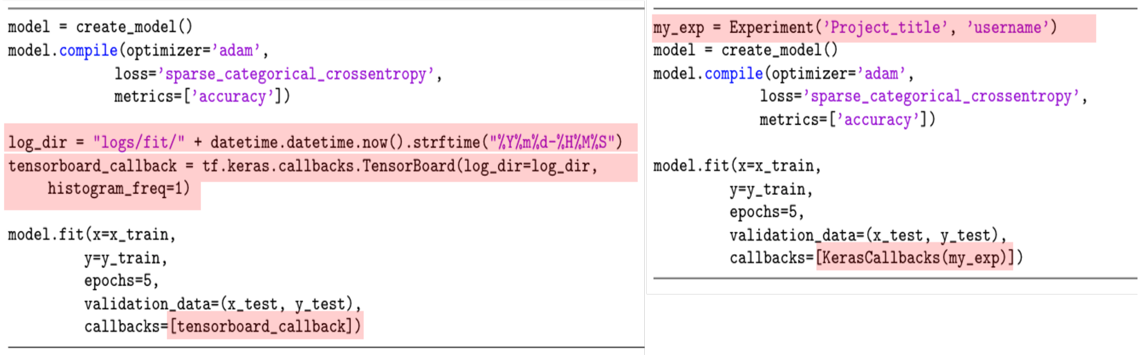
There are two main approaches to manage the modeling lifecycle. First, using a graphical workflow management system that provides user-friendly drag-and-drop features to build and train the model, which is mainly adopted by industry systems such as Microsoft Machine Learning Studio MlStudioAzure, Digits DigitsNV, and Deep Cognition deepcog. Second, using a set of predefined functions to instrument the code in its native framework. While the first approach provides an easy to use workflow management system, it might limit the users to a specific set of libraries and services. Moreover, the interviews conducted by Vartak et al. vartak2018infrastructure show that data scientists prefer not to change their favorite frameworks to a GUI management system.

The recent advances in deep learning and its applications have led to the development of several deep learning frameworks, such as TensorFlow abadi2016tensorflow, Caffee jia2014caffe, and PyTorch paszke2017automatic. These frameworks focus on the development, training aspects, and evaluation aspects. However, these frameworks have largely ignored the challenges of the modeling lifecycle, until the management challenges started floating at the surface of the overall development lifecycle.

Therefore, platform providers started developing solutions to accompany their deep learning platforms. For example, FBLearner Flow for PyTorch FBLearner, TensorBoard for TensorFlow TensorBoard, Digits by Nvidia DigitsNV, Michelangelo by Uber Michelangelo, Amazon's SageMaker SageMaker a fully-loaded machine learning training and deployment system. Additionally, other frame-work independent tools include CometML (`www.comet.ml`), Weights and Biases (`https://www.wandb.com`), Neptune (`https://neptune.ml`). However, these systems restrict the user to a specific framework, processing pipelines, and deployment options.

One of the most popular tools in this realm is TensorBoard, which focuses on a single phase of the lifecycle, i.e., experiment monitoring, compared to ModelKB, which covers the lifecycle end-to-end. Yet, using ModelKB is much easier than using TensorBoard, which requires significant code instrumentation. Figure 6 illustrates side-by-side the code changes required to track experiments in (A) TensorBoard and (B) ModelKB. Additionally, using TensorBoard becomes much more challenging for other deep learning platforms such as PyTorch. Another very popular tool is MLFlow zaharia2018accelerating, an open source platform for the machine learning lifecycle. MLFlow comes very close to our system design and goals. And it has been going through tremendous updates and new features are being added frequently. However, similar to TensorBoard, MLFlow requires significant code changes to track and monitor experiments.

Several other systems from both academia and industry emerged to facilitate the lifecycle management challenges. ModelDB miao2017modelhub was one of the first systems that aimed at addressing model management issues in machine learning. It



```
model = create_model()
model.compile(optimizer=’adam’,
             loss=’sparse_categorical_crossentropy’,
             metrics=[’accuracy’])

log_dir = "logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir,
     histogram_freq=1)

model.fit(x=x_train,
         y=y_train,
         epochs=5,
         validation_data=(x_test, y_test),
         callbacks=[tensorboard_callback])
```

```
my_exp = Experiment(’Project_title’, ’username’)
model = create_model()
model.compile(optimizer=’adam’,
             loss=’sparse_categorical_crossentropy’,
             metrics=[’accuracy’])

model.fit(x=x_train,
         y=y_train,
         epochs=5,
         validation_data=(x_test, y_test),
         callbacks=[KerasCallbacks(my_exp)])
```

(A) Tracking experiments using TensorBoard

(B) Tracking experiments using ModelKB

Figure 6: Comparison between the code changes required to track experiments in TensorBoard vs. ModelKB. The code highlighted in red is specific to each platform to initiate experiment tracking and monitoring.

comes very close to our solution in its functionality and providing a local interactive interface. However, ModelDB is tailored for machine learning models built in *scikit-learn* and *spark.ml*, which provides limited support for deep neural networks. ModelHub miao2017towards is a high-profile deep learning management system that proposes a domain-specific language to allow easy exploration of models, a model versioning system, and a deep-learning-specific storage system. It also provides a cloud-based repository. However, model sharing and deployment via ModelHub requires manual effort and significant code changes. Moreover, ModelHub does not support automatic generation of inference functions. Schelter et al. schelter2018declarative present a light-weight tool to manage the metadata and lineage of common artifacts in machine learning. While their system is capable of monitoring experiments and providing visual means for search and comparison, they lack the support of other lifecycle phases, such as deployment.

A set of the existing platforms that aim at fostering and facilitating scientific collaboration and model sharing include OpenML van2013openml,hines2004modeldb and Google SeebBank googleSeed. However, these platforms mostly focused on sharing the models in their native formats, along with their results and documentation. They usually require the user to populate this information manually. These platforms lack the fundamental tasks of extracting the metadata from the experiments and, therefore, cannot be used for analysis or automation purposes.

Overall, the existing management systems still face two main challenges: they are either limited to a specific deep learning framework or support part of the modeling lifecycle, such as monitoring experiments, while ignoring other essential parts of the overall lifecycle, such as model deployment and publishing. In contrast, ModelKB aims at automating the model management end-to-end across all lifecycle phases with minimal user intervention.

CHAPTER 3

AUTOMATED MANAGEMENT OF THE MODELING LIFECYCLE

Figure 7 illustrates an overview of ModelKB approach, which spreads over two parts: a local client and a cloud-based server. *ModelKB Client*, mostly referred to as ModelKB, is a local software library that automates the overall modeling lifecycle. The Server side is responsible for providing services for remote sharing and deployment. The management tasks, e.g., data visualization and model deployment, are automated/semi-automated by ModelKB. Python, an object-oriented programming language, is used for the implementation tasks, and it is the language supported by our approach. The technical contributions of ModelKB include unifying the entire lifecycle management in a single system using a wide range of software methods and tools, including the use of Python *Callbacks* for extracting metadata; *Abstract Syntax Trees* (AST) AST for tracking data province, and *Jinja* Jinja template language for generating code automatically–to name a few. Additionally, ModelKB facilitates its usage by providing the user with an intuitive GUI and a set of abstract APIs that can be used in any integrated development environment (IDE). The Server is mainly used to deploy trained models on the cloud and provides a set of REST APIs for the inference tasks. It is also used to facilitate publishing and sharing models. *Publishing* refers to the process of sharing a model publicly (akin to sharing source code on GitHub) while *Sharing* refers to the process of sharing a model between users.

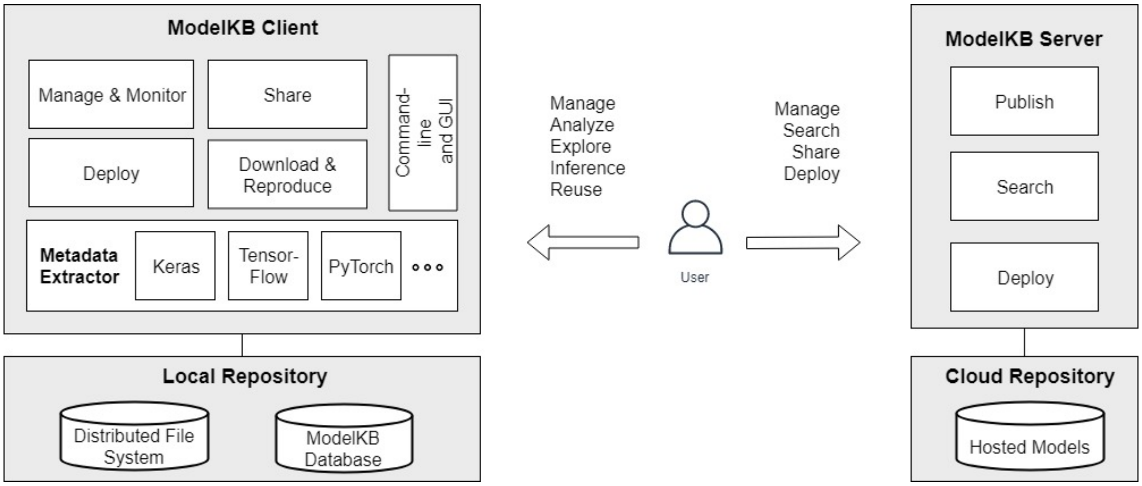


Figure 7: ModelKB System Architecture

As shown in Figure 7, ModelKB consists of the following components: *Metadata Extractor* for automatically extracting the metadata from the source code of different deep learning frameworks; *Local Repository* is the local storage unit for the metadata and other artifacts implemented in two parts, a distributed file system and an SQLite database SQL organized by the *Data Storage* component; *Visualization* is used to visualize projects, experiments, and their metadata; *Upload and Download* is responsible for sharing models among users. Moreover, our system can automatically generate a template for predict function that exposes the trained model for local inference requests. The software features of ModelKB are accessible through command-line promotes and a web-based *Graphical User Interface*, which is used to visualize, explore, compare, and test experiments. We further explain each software component as follows.

Table 2: A brief list of the types of metadata extracted from each experiment

Type	Dataset	Model	Context
Metadata	- Batch size - Pointer to dataset location - Data type (e.g., images, tabular) - Preprocessing steps	- Architecture and its configuration - Parameters (weights and biases) - Hyperparameters (learning rate, epochs, input shape, output shape, optimization algorithm, loss function, etc.) - Accuracy and Loss	- Project info. - User - Implementation - Environment - Dependencies

3.1 Automatic Metadata Extraction

The availability of metadata constitutes a concrete stepping-stone on which the rest of our system's tasks are built upon. We use the term *Metadata* to refer to all types of data and parameters that govern the modeling lifecycle, including hyperparameters, parameters, and context metadata. *Hyperparameters* refer to variables that govern the learning algorithm and are set by the user, such as optimization algorithm, learning rate, loss function, and a number of epochs. *Parameters* refer to the variables learned by the algorithm–specifically weights and biases. In addition, we use the term *Context* metadata to refer to additional data about the experiment, including the project title, user, environment, and framework. Context metadata plays a critical role in setting up the target environment for reproducibility, sharing, and deployment. Table 2 illustrates a shortened list of metadata that our approach extracts from each experiment. Figure 8 illustrates a modeified version of our metadata schema .

In order to automatically extract the metadata mentioned above, we developed our Metadata Extractor based on Callbacks and ASTs. Callbacks are functions that can be passed as an argument to another function to be called back (i.e., executed) at

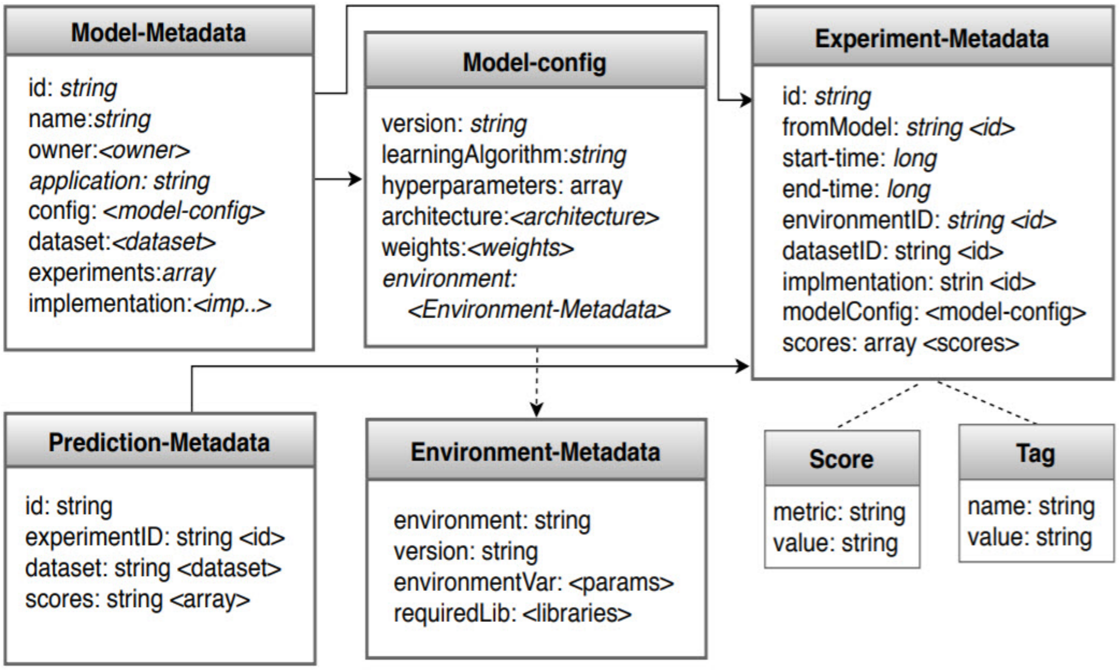


Figure 8: A simplified version of the metadata schema.

a given time. The execution of Callbacks can be initiated in several points of time–before, during, and after–the execution of the caller function. Specifically, we developed Callbacks that will initiate the metadata extraction process at the beginning of each training cycle automatically. Our callbacks are functions that can be applied at different stages of the training phase and are passed as an argument to the `.fit()` and `.fit_generator()` functions, which are used to initiate the training phase in both Keras and TensorFlow 2.0. Listing 3.1 illustrates a snippet of our callbacks implementation for Keras.

1

```
Class KerasCallback(keras.callbacks.Callback):
    def __init__(self, experiment):
        self.project_title = experiment.project_title
        self.metadata_experiment = dict()
        self.accuracy_test = list()
        self.loss_validation = list()
        # rest of the function...

    def on_training_begin(self, seed=None):
        random.seed = seed
        self.start_time = get_current_date()
        self.metadata_data[’batch_size’] = self.params.get(’bz’)
        # rest of the function...

    def on_epoch_end(self, epoch):
        self.accuracy_test.append(logs.get(’acc’))
        self.loss_test.append(logs.get(’loss’))
        self.accuracy_validation.append(logs.get(’val_acc’))
        self.loss_validation.append(logs.get(’val_loss’))
        # rest of the function...

    def on_training_end(self)
        self._get_experiment()
        # rest of the function...

    # rest of the functions...
```

Listing 3.1: A snippest of the Keras callback

One of the main technical contributions and advantages of ModelKB over other similar systems is the minimal code changes required to invoke the entire management process. Listing 3.2 illustrates how to use ModelKB to manage the modeling experiments in the case of Keras, which only requires importing ModelKB library and then passing our pre-defined Callbacks. From Listing 3.2, we notice the minimal amount of code required to invoke the model management process: the user needs only to import our modules, i.e., lines 1 and 2, define a new experiment, line 4, and then add the Callbacks with the defined experiment as an argument to the named parameter `callbacks` of the `fit` function, line 7. This process is identical in the case of TensorFlow 2.0; the only change needed is to use the proper Callback name, which is `TensorflowCallbacks` and importing its library.

```
1 from modelkb import Experiment
2 from modelkb.keras import KerasCallbacks
3 import keras

4 my_exp = Experiment('Project_title', 'username')
5 (x_train, y_train), (x_test, y_test) = mnist.load_data()
# data processing and preparation...
6 model = Sequential()
# model building and compiling...
7 model.fit(x_train, y_train, callbacks=[KerasCallbacks(my_exp)])
# rest of the code
```

Listing 3.2: A code snippet illustrating how to use ModelKB in Keras

Another contribution of our work is a hierarchical organization of deep learning projects. Each project is made up of several experiments, often tens to hundreds. A project is created by the user when declaring the experiment object by specifying a project name, as shown in line 4 of Listing 3.2. This project is different from the project created by the IDE. We use Projects to group experiments that belong to the same modeling task. Once a project is created, we assign it a unique identifier, a timestamp, and a username who created it. Then, as long as the project name is not changed, all experiments created under that project will be grouped together. This holds true in real practice since each project involves running tens to hundreds of experiments, where each experiment involves exploring different hyperparameters until reaching a satisfying result. In our approach, users do not need to specify names for each experiment, which further reduces the modeling overhead. We automatically assign every experiment a unique ID and name it using a combination of its ID and training start time.

For example, given an object detection task, the developer might declare her experiment as `my_exp = Experiment('FaceDetector', 'Alice')`. Then, she experiments with different hyperparameters, such as different optimization algorithms (SGD, Adam) and fine-tunes other hyperparameters. Every time Alice runs an experiment, she does not change the name of the project. She only passes our Callbacks to her `fit` function on the first experiment run and then continues her modeling tasks as usual. The management tasks will be carried out automatically by ModelKB without any further intervention from Alice.

Similarly, users of PyTorch can use ModelKB to manage their experiments automatically. However, unlike Keras and TensorFlow, PyTorch does not provide a predefined function to train the model and evaluate its performance. Instead, users must implement the training steps using high-level functions predefined in the framework `torch`. For example, a training cycle in PyTorch consists of the following steps: After preparing the dataset and the neural network, `model`, the user passes the data through the neural network in a process known as feedforward step using the defined model, then she computes the loss, `loss = criterion(pred, target)`, with respect to the generated predictions, and then updates the weights of the model `optimizer.step()`. These functions are then wrapped inside a *for* loop, `for e in epochs`, to repeat each training cycle (i.e., epoch) a certain number of times. Therefore, in order to facilitate the usage of ModelKB withing PyTorch we first implemented a high-level function wrapper, *fit* and then developed the necessary Callbacks to enable metadata extraction.

While not a direct goal of this work, but a direct technical contribution is the development of high-level abstract functions, `fit` and `fit_generator` for PyTorch, inspired by Keras implementation. Refer to Section 4 for more implementation details. One the one hand, we found that these functions can significantly reduce the overhead in implementing the training cycle, as explained above. Other well-known frameworks, specifically *fast.ai* (www.fast.ai), have already done this–provided high-level APIs to use PyTorch. In this work, we did not reinvent the wheel, rather developed the two functions necessary for our approach. Thus, reducing the overhead in implementing those abstract functions is a positive side-effect of our approach. On the other hand, these functions drastically simplified the process of extracting the metadata using Callbacks. We needed to implement both `fit` and `fit_generator` similar to the implementation in Keras to facilitate reusing/extending our already-developed Callbacks for PyTorch.

Listing 3.3 illustrates a code snippet of using ModelKB in PyTorch. The user needs first to import the necessary libraries, define an Experiment object and initialize its required parameters, and finally pass the `PytorchCallbacks` function with the experiment object as a parameter to the `fit` function, which is also provided by ModelKB.

```
from modelkb import Experiment
from modelkb.torch import Model, PytorchCallbacks
import torch

my_exp = Experiment(’Project_title’, ’username’)

train_loader = DataLoader(train_set,  transform=transform)
# define the neural network, NET
model = Net()
optimizer = optim.SGD(model.parameters(), lr=lr)
criterion = nn.NLLLoss()
model.fit(train_loader, callbacks=[PytorchCallbacks(my_exp)])
# rest of the code...
```

Listing 3.3: A code snippet illustrating how to use ModelKB in PyTorch using our customized *fit* function

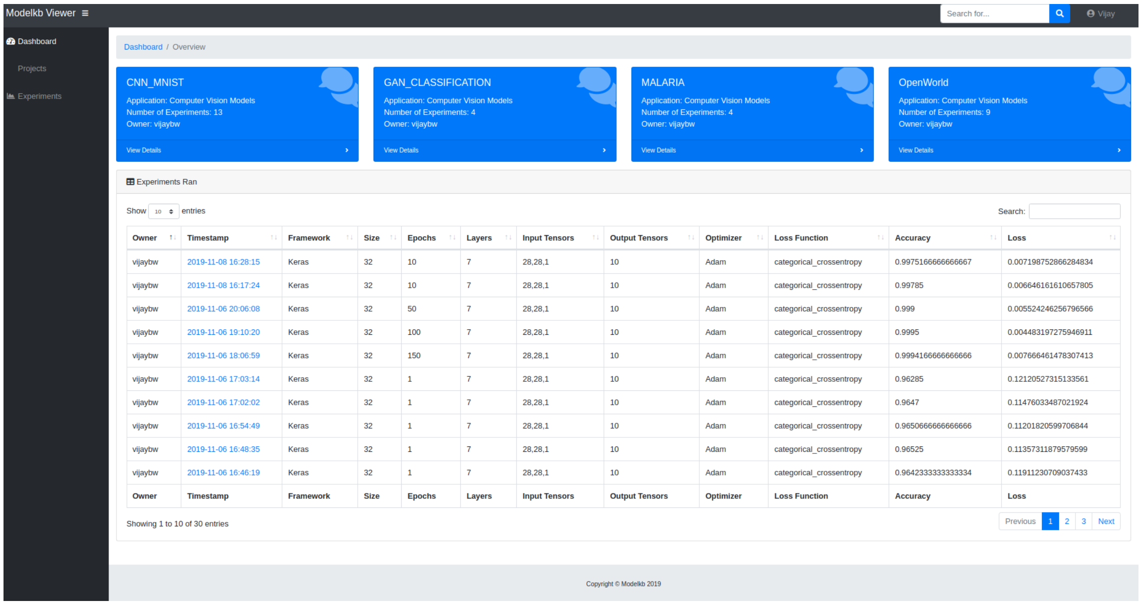
An essential feature of our approach is the ability to unify the seed value among experiments optionally. It is challenging to automatically obtain the seed value from the underlying system during the training phase. The seed value–if known and managed correctly–can drastically improve the reproducibility of experiments, since training deep neural networks depends on several stochastic techniques and involves several random variables. However, extracting the seed value from the system is not an easy task, and sometimes that value is not accessible. Therefore, we allow the user to manually set and unify a seed value across all experiments to facilitate reproducibility. Note that several deep learning practitioners already follow this training convention, but lack proper tools to keep track of the seed values across different experiments.

At the end of the training cycle, we organize the extracted metadata into dictionaries and store them in a local database along with the model file (i.e., model architecture and parameters) and implementation files (source code). The model file is then parsed to visualize its architecture and per-layer configurations. Then, the deployment functionality is invoked to generate a prediction function for casual inference automatically. After that, a web-based dashboard is automatically created and populated with the experiment metadata and artifacts, and its URL is sent to the user's IDE.

3.2 Data Visualization

ModelKB provides a local web-based visualization that allows users to explore, analyze, compare, and run casual inference tasks. Specifically, ModelKB provides three main views: Dashboard, Project view, and Experiment view. The GUI of ModelKB is built on top of *Flask* (`https://palletsprojects.com/p/flask/`), a set of open-source tools and libraries to build web applications. We selected Flask because it is a lightweight framework, easy to implement, and it requires little to no external dependencies. Moreover, Flask is popular and has an outstanding support community, which makes it easy to extend and debug grinberg2018flask.

After each experiment, we print the local URL address of our flask-based server, where ModelKB is hosted. The user can click on the active URL address to visit the landing page of ModelKB–the Dashboard. The Dashboard (see Figure 9) provides a high-level view of all projects stored in the system and a tabular view of the recent experiments sorted by date. Each of the projects is represented in a blue box (the user can select different color templates) that shows the project's title, owner, application (e.g., stock prediction), and the total number of experiments inside that project. Clicking on a



CNN_MNIST
Application: Computer Vision Models
Number of Experiments: 13
Owner: vijaybw

GAN_CLASSIFICATION
Application: Computer Vision Models
Number of Experiments: 4
Owner: vijaybw

MALARIA
Application: Computer Vision Models
Number of Experiments: 4
Owner: vijaybw

OpenWorld
Application: Computer Vision Models
Number of Experiments: 9
Owner: vijaybw

Experiments Ran

Show 10 entries

Search:

Owner	Timestamp	Framework	Size	Epochs	Layers	Input Tensors	Output Tensors	Optimizer	Loss Function	Accuracy	Loss
vijaybw	2019-11-08 16:28:15	Keras	32	10	7	28,28,1	10	Adam	categorical_crossentropy	0.9975166666666667	0.007198752866284834
vijaybw	2019-11-08 16:17:24	Keras	32	10	7	28,28,1	10	Adam	categorical_crossentropy	0.99785	0.006646161610657805
vijaybw	2019-11-06 20:06:08	Keras	32	50	7	28,28,1	10	Adam	categorical_crossentropy	0.999	0.005524246256796566
vijaybw	2019-11-06 19:10:20	Keras	32	100	7	28,28,1	10	Adam	categorical_crossentropy	0.9995	0.004483197275946911
vijaybw	2019-11-06 18:06:59	Keras	32	150	7	28,28,1	10	Adam	categorical_crossentropy	0.9994166666666666	0.007666461478307413
vijaybw	2019-11-06 17:03:14	Keras	32	1	7	28,28,1	10	Adam	categorical_crossentropy	0.96285	0.12120527315133561
vijaybw	2019-11-06 17:02:02	Keras	32	1	7	28,28,1	10	Adam	categorical_crossentropy	0.9647	0.11476033487021924
vijaybw	2019-11-06 16:54:49	Keras	32	1	7	28,28,1	10	Adam	categorical_crossentropy	0.9650666666666666	0.11201820599706844
vijaybw	2019-11-06 16:48:35	Keras	32	1	7	28,28,1	10	Adam	categorical_crossentropy	0.96525	0.11357311879579599
vijaybw	2019-11-06 16:46:19	Keras	32	1	7	28,28,1	10	Adam	categorical_crossentropy	0.9642333333333334	0.11911230709037433
Owner	Timestamp	Framework	Size	Epochs	Layers	Input Tensors	Output Tensors	Optimizer	Loss Function	Accuracy	Loss

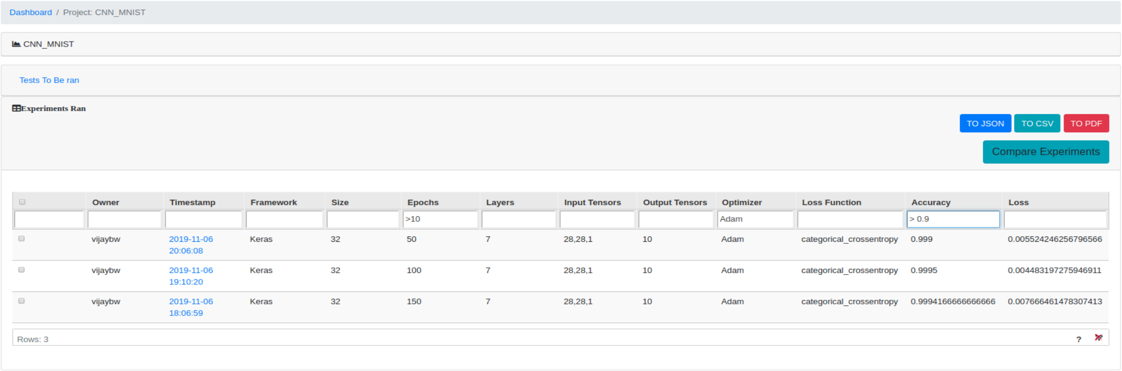
Showing 1 to 10 of 30 entries

Copyright © Modelkb 2019

Figure 9: The Dashboard of ModelKB (landing page)

particular project will lead to the Project view, which we explain as follows.

The Project view, illustrated in Figure 10, lists a summary of all experiments related to the open project. The user can customize the information shown in the table using a drop-down menu on top of the table. The user can select to show the number of layers in each experiment, the accuracy score, and the learning rate. Moreover, the user can also select different filters to query specific results. For example, the user can search for experiments with a specific learning rate and an accuracy score over a given value that was created between two specific dates. For example, in Figure 10, we query models with `epochs > 10 && Optimizer=Adam && Accuracy >0.9`. Another important feature of this view is the ability to compare two models. The user can select two models and click the Compare button. This will lead to a side-by-side comparison for the two models based on the data that the user selects to view (e.g., architecture, accuracy, used



Experiments Ran

□	Owner	Timestamp	Framework	Size	Epochs	Layers	Input Tensors	Output Tensors	Optimizer	Loss Function	Accuracy	Loss
					>10				Adam		> 0.9	
□	vijaybw	2019-11-06 20:06:08	Keras	32	50	7	28,28,1	10	Adam	categorical_crossentropy	0.999	0.005524246256796566
□	vijaybw	2019-11-06 19:10:20	Keras	32	100	7	28,28,1	10	Adam	categorical_crossentropy	0.9995	0.004483197275946911
□	vijaybw	2019-11-06 18:06:59	Keras	32	150	7	28,28,1	10	Adam	categorical_crossentropy	0.9994166666666666	0.007666461478307413

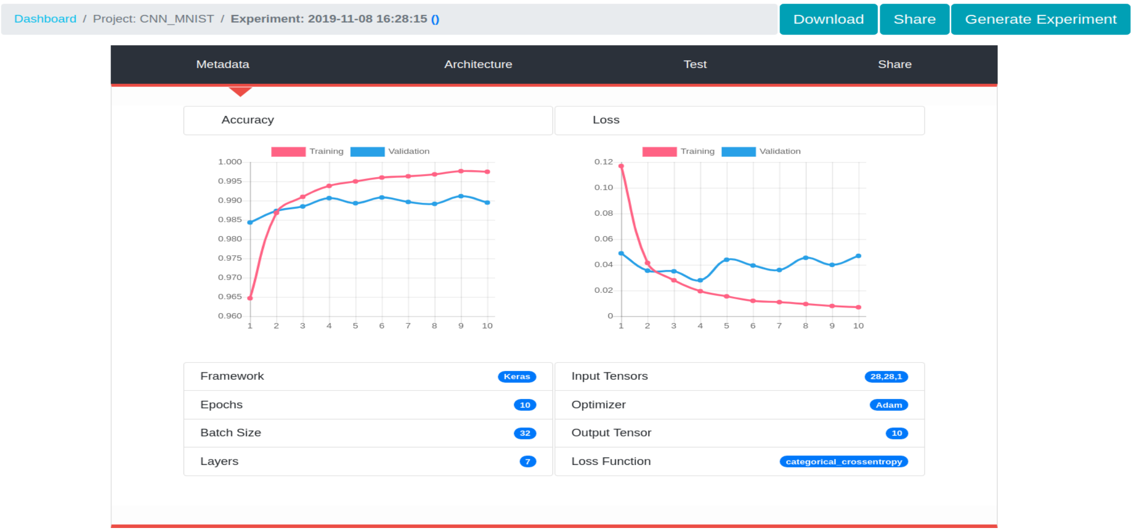
Rows: 3

Figure 10: The Project view lists the experiments and their metadata related to a single project

dataset). Clicking on any of the experiments will lead to the experiment view. Note that users can download those tables in several different formats, including *json*, *csv*, *pdf*, and we currently working on supporting LaTeX, in order to facilitate extracting the metadata from the GUI in different formats, rather than using copy-and-paste.

The Experiment view focuses on visualizing one experiment at a time, and it includes four tabs: *Metadata*, *Architecture*, *Inference*, and *Share*. The Metadata tab lists all the metadata that the user selects to view, including plots for the accuracy and loss scores over time for both training and validation datasets–if available. Figure 11 illustrates the Metadata tab in the Experiment view.

The Architecture tab visualizes the architecture of the model in an interactive interface, where the user can click any layer in the architecture to view its configuration. This functionality is provided by an open-source tool called *Netron* netron, which we integrated into ModelKB. Figure 12 illustrates the Architecture of the current open Experiment (i.e., model). Note that clicking on any layer will open a side panel showing the configuration of the corresponding layer. This feature is beneficial for debugging

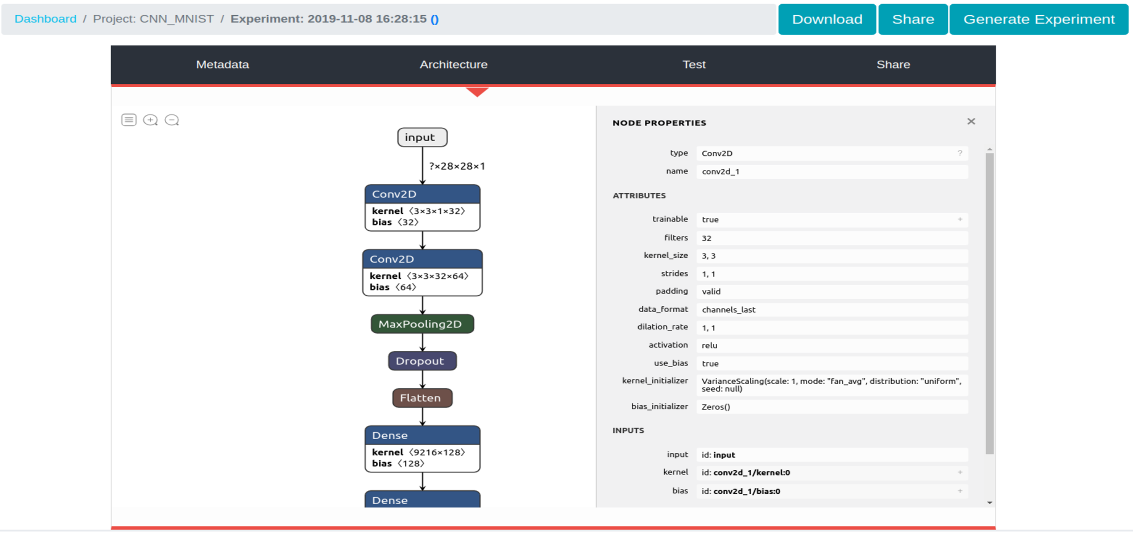


Framework	Keras	Input Tensors	28,28,1
Epochs	10	Optimizer	Adam
Batch Size	32	Output Tensor	10
Layers	7	Loss Function	categorical_crossentropy

Figure 11: The Experiment-Metadata view visualizes the selected metadata, including for loss and accuracy.

and reusing trained neural networks, where the user needs to reach some configuration parameters.

The Inference tab allows the user to run casual inferences on the given model. For example, the user can upload an image and predict its label directly from the GUI without having to manually deploy the model. This feature allows users to test their models against specific inputs without having to rewrite the prediction function. It is enabled through a semi-automated deployment feature, which we will explain in the following subsection. For example, figure 13 illustrates a casual inference task on the given model. The underlying model used for this visualization was trained to predict the hand-written digits using the MNIST dataset (i.e., the ”Hello, World!” example for deep learning). The figure illustrates the input image and its corresponding output generated by the model. The implementation (i.e., source code) of this inference function was generated entirely



NODE PROPERTIES

type	Conv2D
name	conv2d_1

ATTRIBUTES

trainable	true
filters	32
kernel_size	3, 3
strides	1, 1
padding	valid
data_format	channels_last
dilation_rate	1, 1
activation	relu
use_bias	true
kernel_initializer	VarianceScaling(scale: 1, mode: "fan_avg", distribution: "uniform", seed: null)
bias_initializer	Zeros()

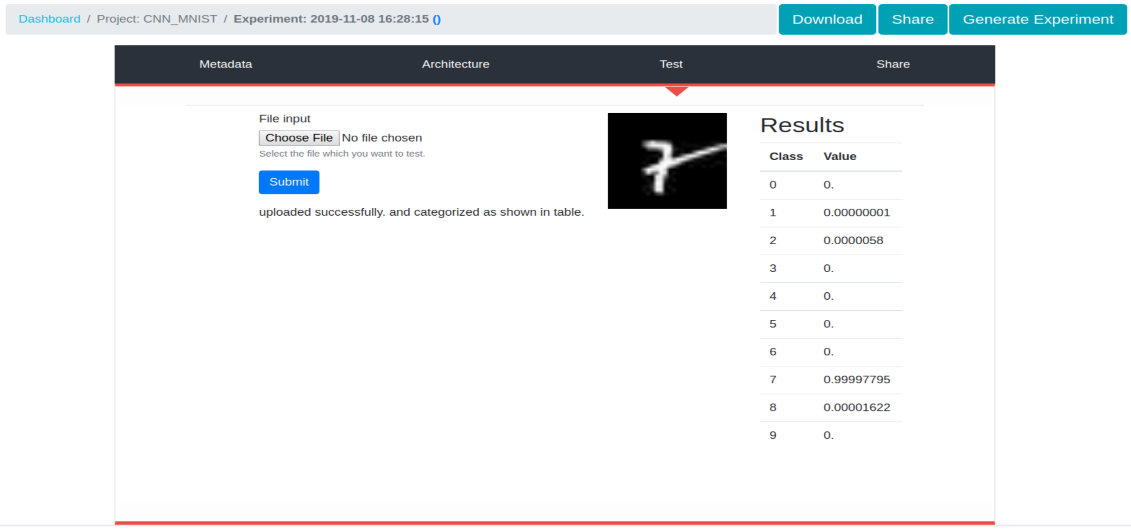
INPUTS

input	id: **input**
kernel	id: **conv2d_1/kernel:0**
bias	id: **conv2d_1/bias:0**

Figure 12: The Architecture view visualizes the neural network architecture and per-layer configurations.

by ModelKB. Moreover, casual inference can assist users in model selection tasks, and can be utilized in the online repository to explore and run predictions before cloning the model to the local environment.

The Share view allows the user to share their experiments remotely by deploying the model on the cloud and then providing the user with two REST APIs: one for remote inference tasks and the other for downloading the model. In order to use the Share feature, the user needs to first have an account on the cloud ModelKB service.



File input

No file chosen

Select the file which you want to test.

uploaded successfully. and categorized as shown in table.

Results

Class	Value
0	0.
1	0.00000001
2	0.0000058
3	0.
4	0.
5	0.
6	0.
7	0.99997795
8	0.00001622
9	0.

Figure 13: The Experiment-Inference feature allows the user to run casual inference tasks on the given model. In this example, the given model is trained to classify hand-written digits using the MNIST dataset. The user selects a picture, as shown above, and the prediction results are then generated by the model and visualized accordingly.

3.3 Model Deployment

Model deployment is one of the most challenging and time-consuming tasks in the overall modeling lifecycle, especially when deploying models on the cloud. Implementing models using a new cloud service and integrating it with an existing system requires significant amounts of time and effort. For example, given a well-trained model, a graduate student with no prior knowledge on cloud deployment services would spend five to seven days to expose a trained model online, not to mention the need for creating a simple GUI or a set of APIs to use the deployed model. Therefore, one of the main features of ModelKB is to facilitate automatic deployment of trained models locally, and semi-automatic deployment on the could with ”on-click” directly from the GUI.

First, ModelKB uses a templating engine to automatically generate inference functions and then uses Docker containers to wrap those functions and their requirements in a proper configuration to be run across platforms. To generate the inference functions automatically, we developed a set of Jinja templates that are capable of parsing the collected metadata and subsequently generate proper inference functions. A proper inference function is a Python function that ”knows” the data type, size, and the preprocessing steps required by the corresponding model. It also knows the required libraries, dependencies, and the output type and shape of the model. List 3.4 illustrates a snippet of a template that is used to generate inference functions for computer vision models (i.e., models that expect images as an input). And Listing 3.5 illustrates a fully functional inference function that was automatically generated by ModelKB. Our deployment templates supports tabular, textual, and image data types.

```
#import statements
{required_import_statements}

first_arg = sys.argv[1]
second_arg = sys.argv[2]
sys.path.append(second_arg)

def xpredict(model_path, image_path):
    model = load_model(model_path)
    image_shape = {inference_data.input_shape}
    {% if inference_data.color_image %}
    img = image.load_img(image_path, target_size=image_shape)
    #Data Preprocessing Steps
    img = image.img_to_array(img)
    {% else %}
    import cv2
    img = cv2.imread(image_path)
    grayImage = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    img = cv2.resize(grayImage, image_shape[0:2])
    {% endif %}
    {% if inference_data.data_augumentation %}
    {inference_data.preprocessing_steps}
    {% endif %}
    batch = np.expand_dims(img, axis=0)
    {% if one_two %}
    batch = np.expand_dims(batch, axis=3)
    {% endif %}
    preds = model.predict(batch)
    np.set_printoptions(suppress=True)
    print (",".join([str(x) for x in preds]))

xpredict(first_arg, second_arg)
```

Listing 3.4: A code snippet illustrating a simple template for generating inference functions for computer vision models.

```
from keras.models import load_model
from keras.preprocessing import image
import numpy as np

model = load_model(’malaria.h5’)
classnames = [’Uninfected’, ’Parasitized’]
img = image.load_img(’p.png’, target_size=model.input_shape[1:])

x = image.img_to_array(img)
x = np.expand_dims(x, axis=0)
x = x/255
preds = model.predict(x)
classe = model.predict_classes(x)

for i in range(0, len(preds[0])):
    print(str(classnames[i]) + " - " + str(preds[0][i]))
```

Listing 3.5: A code snippet of an inference function that was automatically generated by ModelKB. Fully qualified paths are shortened for visualization purposes.

Once an inference function is generated, ModelKB can then parse the required configurations and dependencies, such as the deep learning platform version and build a Docker container to facilitate deploying the model either locally or on the cloud (as shown in Figure 14). Notice that our docker containers also generate a set of REST APIs, one for inference and one for downloading the entire Docker image. Docker images are also created automatically by generating a Docker file using our templating engine. For cloud deployment, the user is required to provide the authentication information to access their hosting platform.

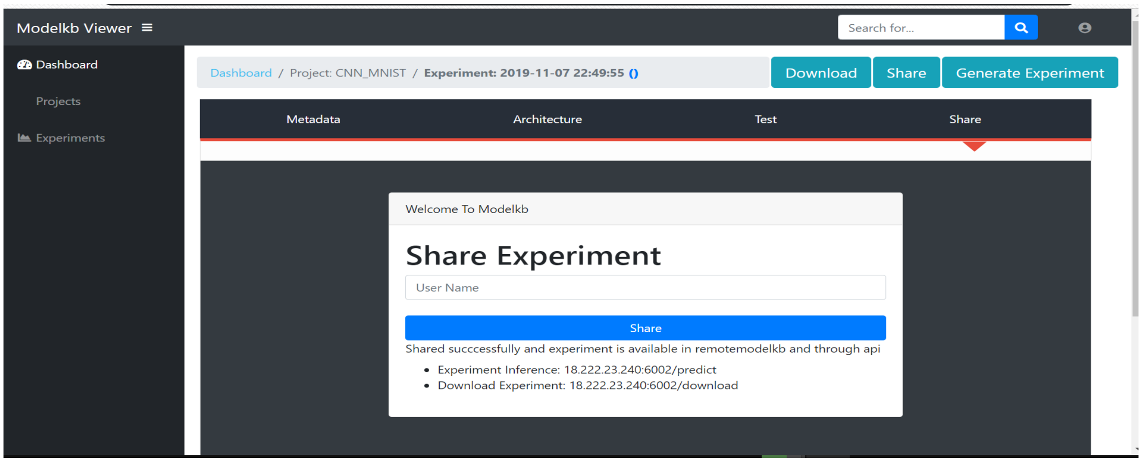
Our containers are currently served on the Kubernetes (`https://kubernetes.io`) serving system via remote procedure calls. Deep learning practitioners are not required to be familiar with the underlying serving functions; instead, ModelKB will take care of setting up the necessary containers and protocols. However, users are responsible for double-checking the correctness of the generated inference functions, which rarely might require additional information that is not present in the extracted metadata, such as class labels.

3.4 Model Sharing

ModelKB provides a simplified approach to share models among users based on whether or not the remote user uses ModelKB. Specifically, we present two approaches to share models:

- *Remote user with ModelKB:* If the receiving user already has ModelKB installed on their system, then they can use the download feature to clone a remote model to their system locally by providing the model's global ID. This will download and unpack a Docker container and create a new database entry to host the model locally.

- *Remote user without ModelKB:* A user that does not have ModelKB installed on their system can download a remote project hosted on ModelKB Server as a zipped file, which contains a Docker container to run an image locally with the inference function, the model file, the inference function as a `.py` file, and the training source code.



Shared succcessfully and experiment is available in remotemodelkb and through api

- Experiment Inference: 18.222.23.240:6002/predict
- Download Experiment: 18.222.23.240:6002/download

Figure 14: Share view allows users to share their experiments remotely, which automatically deploys the model on the cloud and exposes REST APIs to run inference tasks and download the model.

The major technical contribution here is facilitating the process of sharing a model. Specifically, to download a model that is already hosted at ModelKB, the user needs only to provide its unique ID at ModelKB Client-side, and the model download and set-up will be initiated automatically. The new cloned model will be available to view locally and to be imported into a proper IDE for further reusability. In contrast, the user can download a shared model manually, as explained above. Figure 14 illustrates the interface that allows users to host their models remotely in addition to a set of generated APIs for download and inference tasks. An additional technical advantage of automatic APIs generation is that a user can share their model functionality by exposing the API without having to share the actual model file and, therefore, preserving the privacy and intellectual property of the model.

3.5 Reproducibility

With the recent surge in the number of research papers reporting state-of-the-art results in deep learning, the challenge of reproducing a deep learning experiment has come to the forefront. Without proper tracking of the experiment's metadata and the environment set-up, it is difficult to share or reproduce a deep learning model. ModelKB addresses this issue by providing the functionality to package the deep learning model along with its metadata and artifacts into a reusable docker *image* that can be shared through ModelKB and then imported and deployed on the other end. Packaging and downloading a model can be done directly from the ModelKB interface. Using the automatically extracted metadata in addition to unifying the seed value among experiments, ModelKB can enhance the reproducibility of the trained models.

Due to the importance of model reproducibility, one of the most popular conferences in the neural networks domain, NeurIPS, has recently published a reproducibility checklist (https://www.cs.mcgill.ca/ jpineau/ReproducibilityChecklist.pdf). Authors submitting papers to the conference are required to answer every question on the list. Not only does answering the entire checklist is a time-consuming process, but our study and experiments prove that such information might be difficult to collect without using a proper management tool, such as ModelKB. In Table 3, we list the NeurIPS reproducibility checklist and show what information is collected automatically by ModelKB. This validates the usefulness and correctness of ModelKB in enhancing model reproducibility.

Table 3: A comparison between the Reproducibility Checklist requested by NeruIPS and ModelKB features that facilitate reproducibility.

The Machine Learning Reproducibility Checklist	**ModelKB Support**
A clear description of the mathematical setting, algorithm, and/or model.	Automatic extraction of the model's architecture, parameters, and hyperparameters.
An analysis of the complexity (time, space, sample size) of any algorithm	Not supported
A link to a downloadable source code, with specification of all dependencies, including external libraries.	A docker container including source code, dependency specification, and all external libraries
A complete description of the data collection process, including sample size.	Name and sample size of the dataset and all preprocessing steps are collected automatically.
A link to a downloadable version of the dataset or simulation environment.	The user can log the dataset link manually
An explanation of any data that were excluded, description of any pre-processing step.	User can provide explanations as comments. Preprocessing steps are recorded automatically
The range of hyper-parameters considered, method to select the best hyper-parameter configuration, and specification of all hyper-parameters used to generate results	Hyperparameters from all experiments are collected automatically.
An explanation of how samples were allocated for training / validation / testing.	Information about the size and metadata for each group are collected automatically. Explanations can be provided by the user as comments
The exact number of evaluation runs.	Collected automatically.
A description of how experiments were run.	Automatic collection of context metadata, environment, time, data, username, etc.
A clear definition of the specific measure or statistics used to report results	Accuracy and loss metrics are extracted automatically. Additional metrics can be logged manually by the user.
A description of results with central tendency (e.g. mean) & variation (e.g. stddev).	Not supported.
A description of the computing infrastructure used.	Automatic collection of context metadata, environment, time, data, username, etc.

3.6 Implementation

The implementation of ModelKB required a particular set of interdisciplinary skills that fall at the intersections of Software Engineering and Deep Learning. The project required the integration of a wide range of software systems and programming languages, including Python, JavaScript, database management systems, web development tools, and modeling deep learning applications using TensorFlow 2.0, Keras, and PyTorch.

Developing the metadata extractors was the most time-consuming phase, which required extensive knowledge of the underlying frameworks. The metadata extractors were developed in Python using Callbacks. In the case of Keras and TensorFlow 2.0, we developed our Callbacks by extending the Callback base classes of Keras and TensorFlow, i.e., `keras.callbacks.Callback` and `tf.keras.callbacks.Callback`, respectively. Listing 3.1 illustrates a snippiest of our Keras Callbacks code as an example. In particular, the Callbacks first define an empty dictionary, `metadata`, in the `self.__init__` function, whose keys are the metadata elements that will be populated with their corresponding values during the modeling lifecycle. We create a dictionary for each type of the extracted metadata, i.e., dataset, model, and context. We notice from the pseudocode that different types of metadata are collected during different times of the training process. For example, the accuracy and loss scores are computed at the end of each epoch, and therefore, we collect those using the Callback `on_epoch_end`. Current Callbacks include `on_train_begin`, `on_train_end`, `on_epoch_begin`, and `on_epoch_end`.

We followed the same approach mentioned above to develop the Callbacks for PyTorch. However, unlike Keras and TensorFlow 2.0, PyTorch does not provide `.fit()` or `.fit_generator()` functions to invoke model training. Therefore, we had first to develop these two functions to imitate our approach in the case of Keras and TensorFlow 2.0. We argue that developing these functions, especially given the availability of Keras open-course code, provide a much easier approach to reuse our Callbacks rather than developing a new set of Callbacks for PyTorch. This also holds true for other deep learning frameworks. Listing 3.6 illustrates a snippet of our implementation for the `fit` function in PyTorch, which is inspired by the implementation of the `fit` function in Keras.

The rest of the implementation tasks were reasonably straightforward. We used the SQLite database management system to facilitate the storage and query of the metadata. We also used a folder storage system to store the serialized models and other file-based artifacts. Developing the user interface was a straight forward process. We used Flask as the underlying server to serve the web-based interface locally. Visualizing the model's architecture and configuration was adapted from Netron netron, an open-source tool that can visualize the model's architecture using an interactive interface, where the user can click any layer to visualize its configuration. We used Jinja templating engine to write templates for automatic inference function generation. We used Docker containers to package the models for deployment and utilized Kubernetes for serving our models.

```
Class Model():
# rest of the class definition

def fit(self, x, y, validation=None, *, batch_size=batch_size,

epochs=1, steps_per_epoch=None, validation_steps=None,

batches_per_step=1, initial_epoch=1, verbose=True, callbacks=None):

    train_generator = self._dataloader_from_data((x, y),
    batch_size=batch_size)
    valid_generator = None

    if validation_data is not None:
        valid_generator = self._dataloader_from_data(validation_data,
                                           batch_size=batch_size)

    return self.fit_generator(train_generator,
                              valid_generator=valid_generator,
                              epochs=epochs,
                              steps_per_epoch=steps_per_epoch,
                              validation_steps=validation_steps,
                              batches_per_step=batches_per_step,
                              initial_epoch=initial_epoch,
                              verbose=verbose,
                              callbacks=callbacks)
```

Listing 3.6: A snippet of the fit function implementation for PyTorch

3.7 Evaluation

In order to assess the usefulness of ModelKB, we conducted a user study that consisted of six industry participants and seven academic researchers. We also assessed the performance overhead of using ModelKB by measuring its required execution time. The six industry participants included a Data Scientist from HR Block Inc., an ML Engineer intern and a Technical Client Experience Professional both working at IBM, a Data Engineer from Quest Analytics Inc., and a Product Manager and a Design Engineer both from Sprint. Additionally, the user study included seven Ph.D. researchers from the Computer Science Department at UMKC, who are working on developing deep learning solutions for real-world problems.

None of the participants was involved in the development of ModelKB. Some of their projects include classifying EEG signals, MRI image detection, time series analysis using LSTM, building an autoencoder for textual data, and image seg-mentation for brain MRI scans. Seven of the participants used Keras, four of them used TensorFlow 2.0, and two of them used PyTorch. We asked the participants to use ModelKB to manage their deep learning projects and then answer a survey to quantify their feedback. Additionally, we collected data from their experiments on the performance overhead of using ModelKB (i.e., its execution time).

3.7.1 Objectives

The overarching goal of our user study is to assess the usefulness and applicability of our software system in managing the development lifecycle of a realistic deep learning project, i.e., *automatically/semi-automatically monitor, organize, share, deploy, and visualize the model throughout its lifecycle*. Specifically, our evaluation objectives include the following:

1. We investigate the benefits, issues, and difficulties of using ModelKB in deep learning projects. We are specifically interested in whether or not our software is capable of extracting and tracking essential metadata about each experiment and how helpful it is to visualize these metadata for practitioners, e.g., data scientists.

2. We validate the semi-automatic deployment feature locally and on the cloud, which can deploy a model that was managed using our system.

3. We illustrate the benefits and current limitations of the Sharing feature.

4. We evaluate how our system can improve the reproducibility task. This goal is particularly validated through sharing metadata among different team members working on the same project to regenerate a previous experiment. We also compare our system reproducibility features to the recent reproducibility checklist required for papers accepted at NeurIPS Conference (https://nips.cc/).

3.7.2 Methodology

To conduct our evaluation, we developed a set of tasks that applies all features of ModelKB to be carried out by the participants and then asked them to answer a Likert-Scale survey to quantify their feedback. Table 4 lists the set of questions used in our survey, which was developed and collected using Google Forms. Participants respond

Table 4: Questions of the user study questionnaire.

It is important to manage deep learning lifecycle in my work.
It was easy to install ModelKB.
It was easy to use ModelKB for tracking experiments.
ModelKB's interface is intuitive and user-friendly.
Experiments tracking and monitoring in ModelKB is useful.
The hyperparameters collected and visualized by ModelKB are informative.
ModelKB helped me to query my experiments and find their differences.
ModelKB correctly organized my experiments within projects.
The inference function generated by ModelKB worked correctly without errors.
The inference function is useful.
Model architecture visualization was useful.
ModelKB facilitates sharing models.
ModelKB facilitates reproducibility.
ModelKB saves a lot of time and effort in managing the deep learning lifecycle.
Overall, ModelKB helped me focus on the modeling tasks rather than their management.
Overall, I am satisfied with ModelKB performance (i.e., its execution time).
I have used other model management tools before, e.g., Tensorboard.
ModelKB is easier to use compared to other model management tools.
I will continue to use ModelKB in future projects.

to each item in the survey by a score from 1 to 5 (1: strongly disagree, 2: disagree, 3: neutral, 4: agree, 5: strongly agree). The tasks included the following steps:

- Step 1: Clone ModelKB from our private repository in GitHub and set it up locally.

- Step 2: Use ModelKB to track your experiments and then visualize your work evolution using its GUI dashboard.

- Step 3: Use ModelKB to select the top three performing models in a given project and find out their differences and what parameters lead to producing the best model.

- Step 4: Use ModelKB to print your model's architecture.

- Step 5: Use ModelKB to run inference tasks using your trained model. This step must be without any user intervention., i.e., the user should not implement the inference function or expose it on a GUI manually. Instead, the user should use ModelKB features to automatically generate the inference function and expose it via a GUI.

- Step 6: Deploy your model remotely from within ModelKB GUI and use the automatically generated inference API to run inferences from a different application.

- Step 7: Delete your deep learning project from ModelKB and download it from the remote Server. Alternatively, download your hosted model as a zipped file and run the inference function locally using the downloaded docker container.

These tasks were performed on different machines, different integrated development environments including PyCharm, Spider, and Jupyter Notebooks and three deep learning frameworks including Keras, TensorFlow 2.0, and PyTorch.

3.7.3 Results

Work motivation: Before discussing the validation of ModelKB, it is essential to mention here that our first question in the survey asked participants about the importance of managing the lifecycle of their work. 92.3% of the participants (i.e., 12 out of 13) *strongly agree* that managing the deep learning lifecycle is important in their work. Only one participant answered that they *agree* with this statement, which is probably due to their technical background. Similar conclusions have been drawn in previous studies that interviewed a much larger number of participants [98].

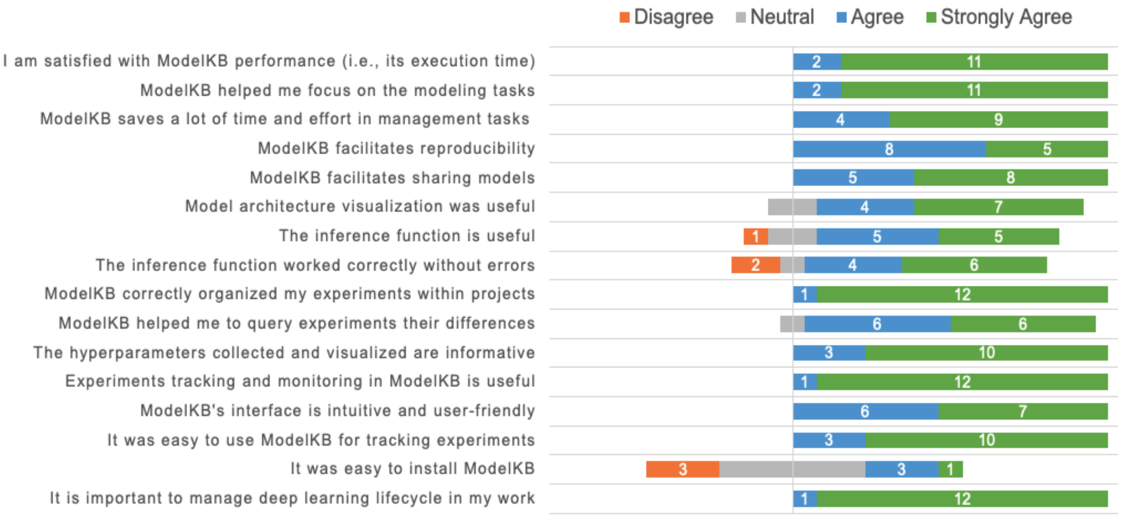


Figure 15: Likert-scale answers to the survey. Total number of participates: 13.

Installing ModelKB: It was expected before setting up the user study that ModelKB would require installing several dependencies before becoming fully-functional. ModelKB is still a research tool that requires some efforts to be installed, which is not at the core of its objectives. Nevertheless, 10 of the participants were able to install ModelKB on their own. We provided direct assistance for stalling ModelKB for 3 participants only. Therefore, we notice from the survey that 46% of the participants were neutral, 31% agree, and 23% disagree that installing ModelKB was easy. While this question, in particular, does not affect the usefulness of ModelKB, we included it in the survey to assess the easiness of its installation for future improvements.

Tracking experiments: ModelKB was able to extract the metadata from all projects, across all experiments, with minimal user intervention, and using all three frameworks. About 77% of the participants *strongly agree* that it was easy to use ModelKB to track their experiments. Users had to add two lines of source code only to their existing or new experiments to initiate ModelKB. And 12 out of 13 participants answered that they *strongly agree* on the usefulness of the experiment tracking in ModelKB. Participants also *agree* that the collected metadata is informative and helpful in providing information to lead the upcoming experiments. However, one of the open comments that were provided by a participant mentioned the lack of seed values among the collected hyperparameters. This is because seed values that are not controlled by the user are extremely challenging–if not impossible–to be extracted. However, ModelKB allows the user to set a specific seed value and unify it across experiments, which in this case, the seed value will be collected automatically.

visualization and analysis: All participants *agree* that ModelKB visualization and its interface are intuitive and user-friendly. Additionally, metadata visualization played a critical role in analyzing and comparing the experiments, which, without our system, is often done manually, and expensive runs are lost unsaved. Practitioners were able to compare their experiments and derive new insights regarding the next experiments using the visual interface. For example, it was easy to practice the *early stopping* regularization technique by looking at the accuracy graphs generated by our system to detect overfitting visually. It was also beneficial to organize the work among different members of the team by learning which member ran what experiment. Overall, 6 participants *strongly agree*, 6 *agree*, and 1 was *neutral* that ModelKB visualization helps query and compare experiments.

Automatic deployment: To validate the importance and benefits of the semi-automated deployment feature, we asked our study subjects to deploy their models on a free web serving service, such as Amazon Web Service or Kubernetes, and provide APIs for running inference tasks remotely. Only five of the thirteen participants (two academic researchers and three software developers) had previous expertise deploying models on the could. Some of the participants–who did not have a previous experience deploying models–were able to deploy their models and provide REST APIs for inference tasks in as little as three days, while other participants needed about two weeks. In contrast, ModelKB is capable of deploying the model both locally and in the cloud within seconds.

Out of the 13 participants, 1 *disagree* and 10 *agree* (out of which 5 *strongly agree*) that the automatically generated inference function is useful. However, it seemed from the survey that 2 participants had issues with the generated inference function; they *disagree* that the functionality worked correctly. Upon further investigation, we learned that one of the participants was training a time series model using voice data. While ModelKB was helpful to carry out all of its functionalities, it failed to generate a proper inference function for voice data due to special data preprocessing steps that required a large number of external dependencies, which generated syntax errors in the generated code, and hence the inference function failed to work. Overall, not only can our system deploy a given model, but it also provides a proper web interface to test the deployed model, which participants found very helpful as discussed above.

A side-positive effect of the automatically-generated inference function was noticed by researchers who build models for different domains than computer science. Specifically, it was beneficial for practitioners whose research involved collaboration with medical doctors. The physicians had no technical knowledge about deep learning, but they provided the datasets and defined the research problem. The casual inference feature, based on the deployed model, provided the physicians with an excellent method to test the developed models and assess their performance. Before ModelKB, researchers had to conduct weekly meetings to run inference examples on their machines and report their results to the physicians manually, case by case.

Reproducibility: Reproducibility is evaluated by the ability to reproduce similar results of a previous experiment. In practice, it is critical to reproduce an older version of a model that is not available or reproduce a model given its metadata, such as models reported in research papers without their implementation. Following our tasks and survey, 8 participants *agree* that ModelKB helps to facilitate model reproducibility, and 5 participants *strongly agree* with that statement. Note that it is challenging to reproduce identical results due to the high degree of randomness involved in training deep learning models. For example, every optimization algorithm (e.g., Stochastic Gradient Descent) has a dedicated technique to initialize the weights in the first step of the training phase, which is challenging to reproduce.

Our experiments illustrated that when trying to reproduce a new model, the extracted metadata helped reproduced models with very similar results, given identical datasets. In some of the experiments, we activated our seed generator to improve reproducibility further. In the worst case, when the seed generator was not used, the highest accuracy difference in a reproduced experiment for an image classification task was 3.7%. Even in this subject study, we considered such a result to be acceptable given that reproducing a model was not done heuristically; instead, it was trained using the metadata and hyperparameters provided by our tool. Hence, the Metadata Extractors are helpful and feasible to provide a sturdy stepping-stone for reproducing experiments.

Model Sharing: ModelKB provided an efficient approach to share models among participating members. To share a trained model, the developer had to first upload it to ModelKB Server, which is done automatically in one click. Once a model is published, other members were able to download the model, with one click as well, by providing the model's ID. Participants reported that sharing models is very useful. Specifically, 8 participants *strongly agree*, and 5 participants *agree* that ModelKB is helpful for sharing models.

Execution time overhead: We evaluated the execution time overhead in training three architectures of different sizes from small to large: LeNet, ResNet50, and a ResNet150 models with and without using ModelKB. We trained each model for 50, 100, and 150 epochs, using MNIST dataset for training the LeNet and CIFAR-10 for ResNet50 and ResNet150. We conducted the experiments on a Linux Ubuntu 16.04 machine, 16 GB RAM, and NVIDIA GTX 1080 GPU. Overall, the execution time difference averaged 2-6 seconds when using ModelKB to manage the experiments. However, we argue that this time difference is neglectable compared to the expensive training time itself, which lasted more than 7 hours in the case of training a ResNet150 from scratch. The survey also illustrated that all of the participants agree that they were satisfied with the ModelKB performance (i.e., execution time).

Overall, all participants reported that using ModelKB eliminated the need for manually keeping track of the hyperparameters and metadata. They also agreed that using

ModelKB required minimal code changes. Overall, 9 out of 13 participants *strongly agree* that ModelKB saves a lot of time and effort in managing the lifecycle tasks and the other 4 textitagree with this statement. Additionally, 11 out of 13 participants *strongly agree* that ModelKB helps them focus on the actual modeling tasks rather than their management, which is usually done manually. The survey also illustrated that more than 70% of the participants did not use a management tool before due to several factors, including the code changes required by such tools. Additionally, out of the 30% participants who used other management tools, 75% of them mentioned that ModelKB is easier to use than some tools they used previously. The other 25% mentioned that ModelKB was ”somehow easier to use than other tools”. In all cases, we argue that compared to the tools that participants used before, such as TensorBoard, ModelKB is the only tool the covers all phases of the lifecycle.

3.7.4 Threats to Validity

A primary risk to our evaluation is small number of study participants. However, we argue that our study participants represents a wide rande of deep learnign developers from academic researchers to machine learning engineers to software developers. Moreover, the study subjects, i.e., models, included a wide range of deep learning models from multi-layer perceptrons (MLP), to autoencoders, U-Nets, ResNet150, and GAN models. Thus, those study subjects covered a wide range of neural network architectures, learning algorithms, optimization approaches, different tasks, and large datasets with different data types, including images and textual data.

Another possible threat to our validity is the usage of a predefined cloud serving system for deploying the models, which might not be scalable for a production-level system. In practice, systems like ModelKB will not provide ”free” serving and deployment for models on the cloud. Users will have first to set up and connect ModelKB to their own deployment servers. Then, ModelKB will be responsible for automatically collecting all required configurations, including dependencies, model parameters, the automatically generated prediction function, and then creation of the docker image to be deployed on the user’s selected service.

CHAPTER 4

FACILITATING PROGRAM COMPREHENSION

4.1 Introduction

4.1.1 Terminology

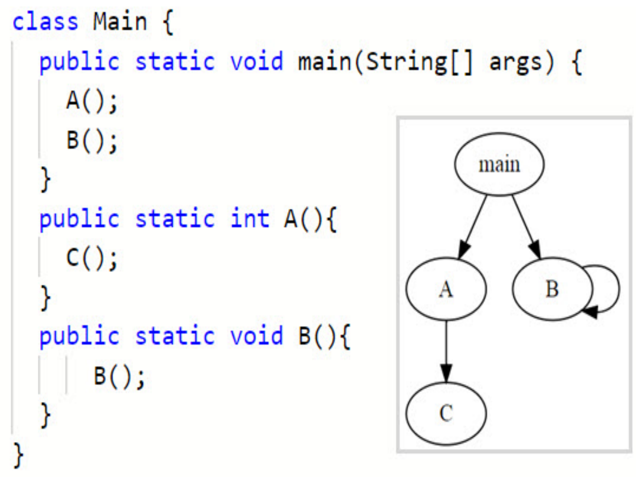
A **call graph** [1] is a visual representation of the software system's function calls, i.e. invocations. It is represented as a directed graph, where each node represents a function and each edge represents a function call. A call graph can be dynamic [5], constructed at runtime, or static [30], constructed at compile time. A dynamic call graph represents a single execution path of the system. A static call graph represents all possible execution paths of the system. Our research focuses on static call graphs and hereafter we use the term ”call graph” to refer to ”static call graph” for simplicity. Fig. 16 illustrates an example of a call graph for a Java code snippet.

An **execution path** is an ordered sequence of function calls from an entry point to an exit point. An execution path does not visit the same function or function call twice.

An **entry point** is the head of the execution path. It is not called by other functions in the system. It is represented as a node without incoming edges.

An **exit point** is the tail of the execution path. It does not call any other function in the system. It is represented as a node without outgoing edges in the call graph.

A ”**function**” refers to all types of procedures in the software system, including functions, static functions, and class methods, unless otherwise distinguished.



```
class Main {
  public static void main(String[] args) {
    A();
    B();
  }
  public static int A(){
    C();
  }
  public static void B(){
      B();
  }
}
```

Figure 16: An example of a Java code snippet and its corresponding call graph

4.1.2 Motivation and Challenges

Program comprehension is an imperative task that is required by software developers and researches in order to reuse, maintain, and improve software systems [13, 23]. As a software system increases in size and complexity, understanding its behavior and implementation becomes one of the most expensive and time-consuming tasks in software development and evolution. In order to facilitate the task of understanding the software and its implementation, developers often read the system's documentation, i.e. a high-level description of the system's behavior, and construct a call graph of the system, i.e. a low-level representation of the system's function calls. Call graphs can support software comprehension and operational analysis tasks and, thus, can enhance software-related activities, including maintenance and evolution.

With the increasing size and complexity of software systems, the generated call graphs are typically very large, including thousands of functions and millions of execution paths. Such large call graphs can have a contrary effect on the software comprehension task: they could be overwhelming and more difficult to understand. In addition, developers often need to manually map their understanding of the overall functionality of the system to its low-level implementation captured by the call graph. Manually mapping the high-level functionality to its low-level implementation is expensive, time consuming, and error prone. Consequently, *there exists a cognitive gap between the high-level functionality of a software system and its low-level implementation*. This creates a compelling demand and motivates the need for automated methods and tools that can enhance the software comprehension tasks and assist developers in understanding the system's structure from the high-level abstraction to the low-level implementation.

In order to assist developers in understanding the low-level structure of a software system, researchers have proposed several methods and tools that focus on optimizing call graphs. For example, several research studies aim at reducing the massive number of execution paths in the call graph in order to facilitate the overall comprehension task [11, 16, 35, 37, 41]. Most of these studies use encoding, filtering, and sampling techniques to reduce the number of execution paths. However, all of them address system comprehension at a single level of granularity, i.e. they depict the execution paths at the source-code level (function to function). A major issue with some of these approaches is that the reduction or filtering processes can implicitly filter-out significant paths of the call graph.

To overcome the aforementioned issue, i.e. representing the system at a single level of granularity, researchers have created methods that can represent the execution paths in multiple abstraction levels [12, 16, 24, 37]. This can help developers understand the system and investigate its functionality at several levels of abstraction. Some of these techniques cluster the call graphs based on the dynamic execution of the program, the package structures, or consider only few execution paths of the software system. Only few of these approaches can label the clustered execution paths. Clustering the paths without a proper label for each cluster still forces the developer to zoom into each cluster and investigate its function calls in order to understand the cluster's functionality.

Therefore, we conclude that the current call-graph-based program comprehension approaches and tools still face the following two challenges:

- **Call graphs are very large and can be difficult to understand.** A call graph may include thousands of functions and millions of execution paths and it could be very difficult to understand and act upon. Therefore, there is a need for a hierarchical clustering approach that can represent the call graph in multi-level hierarchical abstractions in order to facilitate the comprehension task.

- **Clustering the execution paths without proper labeling is not sufficient** While some approaches can cluster the execution paths hierarchically to reduce the size of the graph and represent the system at multiple abstraction levels, they rarely provide helpful information about their functionality. Therefore, it is imperative to identify the functionality of each cluster in the call graph, which can help understanding the system at multiple levels of granularity with a proper label for each cluster.

4.1.3 Goals and Contributions

To this end, we introduce an automated approach and tool support to bridge the cognitive gap between the high-level functionality of a software system and its low-level implementation to enhance the software comprehension tasks. In particular, our approach aims at automating the tasks of (1) constructing and visualizing the static call graph of a software system (2) clustering the execution paths in the call graph into hierarchical abstractions, and (3) labeling each cluster using a proper functionality name. As a result, developers will be able to zoom in and out between the hierarchical clusters to investigate specific functionality of the software system under investigation. We implemented our approach in a stand-alone tool named, *code2graph*. Code2graph is an extension of a previous work [25], which focused on constructing and visualizing static call graphs for software systems written in Python. However, the previous work did not address the two aforementioned challenges. Some results from this chapter are also published in a subsequent work [24].

To evaluate our approach, we used *code2graph* in four case studies ranging from small to large size open-source Python systems, including *code2graph*, *Detectron*, *Flask*, and *Keras*. In addition, we conducted a quantitative evaluation on each of the case studies to evaluate the efficiency and scalability of our tool. To demonstrate that our approach and tool can assist in software comprehension, we used *code2graph* to construct a static call graph of the tool itself, see Fig. 17. The results from the *code2graph* user study made a concrete evaluation since we were directly involved with the implementation of the tool and can rationale about the results. The main contributions in this Chapter are:

- An automated approach to construct static call graphs to help developers understand the software system and its inner interactions. .

- A hierarchical clustering approach to build multi-level clusters of the execution paths to visualize the call graph at different levels of granularity.

- A labeling mechanism to label each cluster with a proper functionality name to help developers with searching for and identifying functionality when needed.

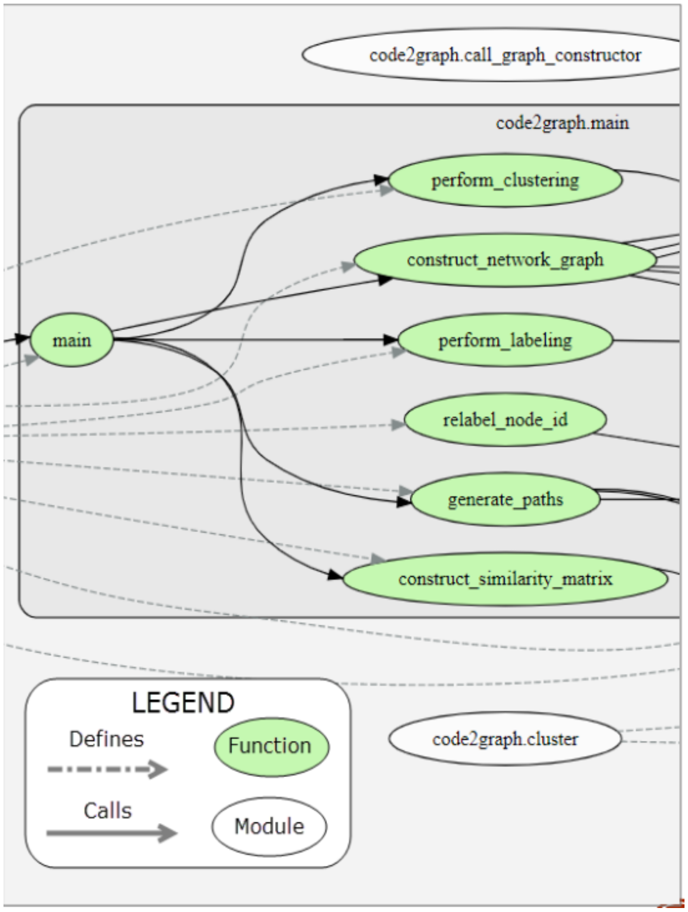
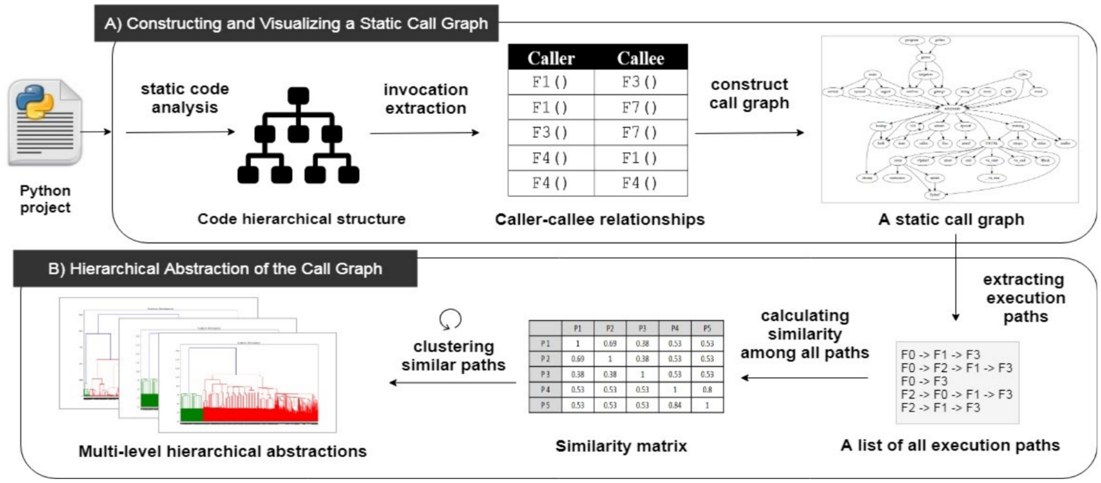


Figure 17: A snippet of the call graph of our tool–Code2Graph.

4.2 Approach

In this section, we present our approach that can automatically (1) construct and visualize the static call graph for a system written in Python, (2) hierarchically cluster the execution paths of the call graph, and (3) label each cluster with a proper functionality name. The goal is to facilitate system comprehension by bridging the cognitive gap between the high-level system functionality and the low-level call graph. Fig. 18 illustrates an overview of our approach, which consists of the following phases.


A) Constructing and Visualizing a Static Call Graph
Python project
static code analysis
Code hierarchical structure
invocation extraction
Caller-callee relationships
construct call graph
A static call graph
B) Hierarchical Abstraction of the Call Graph
extracting execution paths
A list of all execution paths
calculating similarity among all paths
Similarity matrix
clustering similar paths
Multi-level hierarchical abstractions

Caller	Callee
F1()	F3()
F1()	F7()
F3()	F7()
F4()	F1()
F4()	F4()

```
F0 -> F1 -> F3
F0 -> F2 -> F1 -> F3
F0 -> F3
F2 -> F0 -> F1 -> F3
F2 -> F1 -> F3
```

	P1	P2	P3	P4	P5
P1	1	0.69	0.38	0.53	0.53
P2	0.69	1	0.38	0.53	0.53
P3	0.38	0.38	1	0.53	0.53
P4	0.53	0.53	0.53	1	0.8
P5	0.53	0.53	0.53	0.84	1

Figure 18: Code2Graph approach overview.

4.2.1 Constructing and Visualizing the Call Graph

This phase consists of three main steps. First, we statically analyze the codebase to extract the functions’ relationships. Second, we use the functions relationships to construct the call graph. Third, we visualize the call graph in several formats. We further explain these steps as follows.

4.2.1.1 Static Code Analysis

The first step towards constructing the call graph is to extract the code structure and the functions’ relationships, i.e. calls. We extended and used an open-source tool Pyan [39] to extract the code structure and the caller-callee relationships. Pyan uses the AST module to process trees of the Python abstract syntax grammar. In particular, Pyan can traverse the codebase and build a dictionary of the hierarchical namespaces and their constructs, including the following:

- Package, which contains a number of modules and an additional ”_init_.py” file to distinguish it from other packages that might include non-Python files.

- Module, a file that contains Python code and it is treated as an object with an arbitrary number of attributes. Python modules have the ”.py” extension.

- Class, a user-defined data type that contains information about member functions and variables.

- Method, a procedure that is a member function of a class (i.e., instance method).

- $@Staticmethod$, a function decorator that is defined inside a class without any implicit arguments (i.e., a function that does not pass neither $self$ nor cls as implicit first arguments). $self$ is an implicit argument that represents the invoking object instance and it is required to be the first implicit argument of each method defined inside a class. cls is an implicit argument that represents the class.
- $@Classmethod$, a function decorator that allows the function to receive a class (i.e., cls) as an implicit first argument instead of self.

It is important at this step to maintain the functions fully qualified names, which represent the hierarchical namespaces. This is critical for the following steps since modules, classes and functions may have identical names in different namespaces and packages. The caller-callee matrix is important for two tasks. First, it is used to construct the actual call graph using a graph data structure, which allows us to manipulate and traverse the call graph using graph theory. Second, the caller-callee matrix is used to visualize the call graph. Note that the call graph visualization is based on the caller-callee matrix only and does not require any special data structure.

This step produces two main outputs. First, it provides statistics on the codebase structure, i.e. the number of packages, modules, classes, functions, and methods, which can be utilized in further program comprehension features. Second, it extracts the caller-callee relationships and records them in a 2D matrix. The first column holds the caller functions, i.e. functions initiating the call, while the second column holds the called functions, i.e. callee. Thus, each row of the matrix represents a single function call from the caller to the callee.

4.2.1.2 Constructing the Call Graph

a static call graph [66] is defined as a directed graph $\vec{G}$ = (V, E) where V is the set of v functions, and E is the set of edges where each edge, $\vec{e} \in E$, represents an function call (u_{caller}, v_{callee}). The nodes represent the functions of the system. E is the edge set whose elements are the directed edges, also called connections or arrows, of the graph. In a call graph, edges represent function calls (i.e., invocations). The node u (i.e., the head) is named the caller function and the node v (i.e., the tail) is named the callee. For a vertex u the number of incoming arrows is called the indegree and denoted $deg^{-}(u)$ and the number of the outgoing arrows is called the outdegree and is denoted $deg^{+}(u)$. A vertex with $deg^{-}(u) = 0$ is called a source (we call it an entry point), as it is the origin of each of its outgoing arrows. Similarly, a vertex with $deg^{+}(u) = 0$ is called a sink (we call it an exit point), since it is the end of each of its incoming arrows. A vertex with no incoming or outgoing arrows is called an isolated node. Isolated nodes are not included in any execution paths of the system, but we visualize them in the call graph. Isolated functions need further investigation by developers since they represent unused functionality.

In order to construct a call graph and be able to traverse it correctly, we used NetworkX [33]. NetwrokX is a Python graph data structure that facilitates the creation and manipulation of graphs. In particular, the caller-callee matrix was used to iteratively add each pair from the matrix, (u, v) as an edge to the graph. The caller function, u, represents the source node while the called function, v, represents the sink node and the function call represents the direction of the edge. Once the call graph is generated, it can be utilized in several analysis applications. Here, we use it to extract all possible execution paths of the system, which are then clustered into several levels of abstractions in order to facilitate program comprehension. Nevertheless, the call graph can be visualized to depict the systemâs low-level functionality in terms of function calls.

4.2.1.3 Visualizing the Call Graph

Once the caller-callee relationships are completely constructed, they are translated into a graph description language, DOT [46], written in a file with the dot extension. A dot file uses the DOT language to describe a graph, but does not provide facilities for rendering it. We use Graphviz [20], an open-source graph visualization software, to render the dot files and visualize the call graph. Graphviz can process a text description of any graph and render it into useful diagrams in several formats including jpg, svg, and pdf. We prefer the svg format (i.e., Scalable Vector Graphics) since it can preserve the high-quality of the graph when zooming in and out, a features that is critical for visualizing large call graphs. Fig. 17 illustrates a snippet of the code2gragh call graph.

4.2.2 Clustering the Execution Paths of the Call Graph

A static call graph can facilitate program comprehension. However, it depicts the system at a low-level of abstraction. Therefore, we aim to cluster the execution paths of the call graph into multiple abstraction levels to further facilitate system comprehension. This task consists of the following steps.

4.2.2.1 Extracting the Execution Paths

The first step in this phase is to identify all possible execution paths of the software system. An execution path is a list of edges connecting a list of vertices $v_1, v_2, v_3, etc.$ with the restrictions that all edges have the same direction. In addition, none of the edges or vertices can be repeated. An execution path may represent a complete run of the program or a particular functionality. A modified version of the depth-first search (DFS) algorithm built in NetworkX was used in this step to extract all possible simple execution paths. A simple execution path is the path that does not include any cycles. The algorithm is implemented in code2graph as a function called $get\_smiple\_paths(G, s, t)$, where G is the call graph, s is a source node, i.e. the entry point, and t is the target mode, i.e. the exit point. In particular, we first extract all entry and exit points of the system. Then, we use the algorithm to find all simple execution paths from each entry point to all other exit points.

4.2.2.2 Clustering the Execution Paths

Many of the execution paths perform similar functionalities and share a big number of functions with other paths. Therefore, clustering can play a key role in creating hierarchical abstractions from the low-level function calls to the high-level system architecture. Clustering similar execution paths can bridge the gap between the source code and the overall system functionality and, thus, help developers understand the system functionality at several hierarchical levels of abstraction.

To cluster the execution paths into hierarchical abstractions, we used the agglomerative hierarchical clustering technique (AHC). The algorithm requires the execution paths along with their similarity scores to build clusters at different hierarchical levels. It requires two fundamental parameters: a distance metric and a linkage type, which is used in agglomerating the nodes. The distance metric measures the similarity between each path and every other path in the system. We used *Jaccard Distance* to measure the similarity between the execution paths based on Equation 4.1. In the equation, P_i and P_j denote the i^{th} and j^{th} paths and F_i and F_j denote the set of functions called in the paths P_i and P_j, respectively. The result is a similarity matrix of the execution paths.

$$JD(P_i, P_j) = 1 - \frac{F_i \cap F_j}{F_i \cup F_j} \tag{4.1}$$

Once the similarity matrix is generated, the clustering process starts by considering each execution path as a single cluster on its own. Then,the clusters are sequentially combined into larger clusters until all clusters are grouped in one high-level cluster, i.e., one cluster representing the entire system. In an iterative bottom-up approach, at each step, the two clusters separated by the shortest distance are combined. The shortest distance in our approach is measured using the single-linkage clustering approach. The approach measures the distance based on the two closest elements in two adjacent clusters. Mathematically, the single-linkage distance approach is calculated using Equation 4.2. In the equation, $D(X, Y)$is the distance between X and Y, where X and Y are any two sets of clusters, in our case the sets represent the execution paths or clusters of the execution paths, and $d(x, y)$ denotes the distance between the two elements x and y, which are members of the sets X and Y, respective

$$D(X,Y) = min_{x \in X, y \in Y}(x,y) \tag{4.2}$$

The hierarchical levels are clustered from the low-level function calls up to a single cluster representing the entire system. As a result, developers can explore the system functionality starting with the highest level of abstraction, i.e. the entire system represented as a single cluster, and zoom into multiple levels of clusters until reaching the source code. Developers can also trace a function call from its implementation to any higher level of abstraction. However, while the clustered execution paths can enhance program comprehension, developers still need to investigate the functions of each cluster in order to understand the clusterâs overall functionality. Therefore, we were motivated by the fact that function names are usually expressive and contain clues to their functionality [16, 58] to label the generated clusters.

4.2.3 Labeling the Clusters

Labeling the generated clusters can facilitate the software system comprehension task and allows developers to understand the overall functionality of each cluster without having to inspect the functions that compose each cluster. In order to label the clusters with semantically relevant names to their underlying functions, we use the term frequency-inverse document frequency technique (TF-IDF) [75]. The $tf - idf$ weight is computed using two terms. The first computes the normalized Term Frequency (tf), the number of times a word appears in a document divided by the total number of words in that document. The second term is the Inverse Document Frequency (idf), the logarithm of the number of the documents in the corpus divided by the number of documents where the specific term appears. In our case, we treat each execution path as a document and each function name as a term. The equation is defined in Equation 4.3, where $|D|$ is the total number of documents and $|\{d \in D : t \in d\}|$ is the number of documents where t appears. In an iterative way, we calculate and present the top five terms as the label set for each cluster.

$$idf = log \frac{|D|}{|\{d \in D : t \in d\}|} \tag{4.3}$$

4.2.4 Characterizing Software Structure

Our approach analyzes the call graph generated in the previous step to quantify and visualize the software structure. First, we analyze meaningful graph metrics, including the number of nodes and edges, the graph average degree, the clustering coefficient, the graph diameter, and its modularity with a goal of quantifying software structure. The overarching goal is to demonstrate that the graph metrics can be used to quantify software quality based on its call graph. In the following, first define the used graph-based metrics.

- *Average degree*: The average degree of a graph is the largest vertex degree. A vertex degree is the number of edges incident to that vertex, with the loops counted twice. The average degree of a node, i.e., function, indicates its popularity, and thus, it can quantify the graph coarseness. Graphs with a higher average degree tend to be tightly connected. The average degree is defined as $\overline{k} = \frac{2|E|}{|V|}$ where $|E|$ denotes the number of edges and $|V|$ denotes the number of nodes.

- *Clustering coefficient*: It is a property of the nodes that measures to what degree the neighbors of a node v_i are also connected. A high value of the clustering coefficient means that the nodes in the neighborhood are tightly connected, and a value of 0 means that there are hardly any connections in the neighborhood. High values are discouraged in software engineering practices [27]. The neighborhood, N_i, of a vertex, v_i, is defined as its immediately connected neighbors and is given as $N_i = \{v_j : e_{ij} \in E \vee e_{ji} \in E\}$. The local clustering coefficient C_i for a vertex v_i, where k_i is the number of vertices in the neighborhood, is given as:

$$C_i = \frac{|\{e_{jk} : v_j, v_k \in N_i, e_{jk} \in E\}|}{k_i(k_i - 1)} \tag{4.4}$$

- *Graph diameter*: It is the longest shortest path in between any two vertices in the graph. Graphs with higher diameters are undesirable since they reflect deeper run-time stacks, which obstruct program comprehension, debugging, and maintenance.

- *Graph Modularity*: It is used to detect the community structures of a graph. A graph with a high modularity ratio between the nodes of a subgraph but sparse connections to the nodes in different subgraphs is desired. Inspired by the work in [7], a call graph modularity is defined as the ratio between its cohesion and its coupling values. It is well-known that the design of software systems is expected to have high cohesion and low coupling, which makes the system easier to test, debug, and maintain. The modularity of a subgraph A is defined as:

$$Modularity(A) = \frac{Cohesion(A)}{Coupling(A)}. \tag{4.5}$$

In a future work, we also cover software evolution buy comparing the software releases based on their portraits using the Portrait Divergence metric [4]. The graph metrics and portrait divergence are used to quantify and visually illustrate software evolution and execution-path code changes, which can ultimately facilitate several software engineering tasks, especially aiding integration tests.

4.3 Implementation

The implementation tasks were slightly straightforward. First, we extended and integrated an existing tool, *Pyan*, that extracts a list of the caller-callee functions given a source code. Second, we used the caller-callee relationships to construct a graph using a generic graph data structure. Third, once the graph object is generated, we can either visualize it to the user by translating the graph into a DOT language or use it as an input to the metrics calculator or clustering phase, which can calculate and visualize the aforementioned graph metrics. Fourth, we implemented the topic modeling algorithm mentioned above using Python.

The tool is fully implemented in Python and operated locally using a web-based interface. We used HTML and CSS for the visuals and designs, along with Javascript for client-side requests to data retrieval from the server. For the backend server development, we utilized Node.js, a well-known open-source cross-platform Javascript run-time environment that significantly simplifies the process of server-side programming for web applications. Therefore, to make the application even more dynamic and flexible, we employed Express.js, another Javascript library that functions on top of Node.js and is meant for convenient, scalable, and efficient routing of the whole system, without which it may add layers of complexity upon determining correct and accurate routing components from the standard Node.js environment.

4.4 Evaluation

In order to evaluate our approach, we developed a stand-alone Python tool, named code2grpah, and used it to conduct four case studies. We first present the four case studies to illustrate that our approach and tool are capable of constructing a static call graph and clustering it into labeled hierarchical clusters. Second, we present a quantitative evaluation of the case studies to assess the cost and efficiency of our tool using real-world applications. In both evaluations, we present a subset of the graphical results only, due to the limited space in the paper, and present the complete results at the toolâs website `https://info.umkc.edu/UDIC_Research/index.php/wgen/`.

The four case studies are open-source projects, namely Detectron, Flask, Keras, and code2graph. The first three case studies were selected based on their ranking on

GitHub (i.e., the most popular projects). Detectron, is an open-source software system from Facebook that implements state-of-the-art object detection algorithms. It is written in Python and has a relatively small-size codebase. Flask, is an open-source web micro framework written in Python. It provides tools, libraries, and technologies to facilitate web development. Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow. It was developed with a focus on enabling fast experimentation and it is the largest case study in our evaluation. In addition, we ought to analyze code2graph using code2graph in order to evaluate its ability to describe the software system functionality using the hierarchical call graph, since we were directly involved in its implementation and we can efficiently validate its results.

The goals of our evaluation are to validate the feasibility, functionality, and scalability of our approach. The first step in the evaluation was to use our tool to construct and visualize the static call graphs for each of the case studies. In particular, we used code2graph to analyze the case studies and extract invaluable metadata about the codebase, which can be helpful in understanding the overall structure of the system. We removed the test cases from each case study since they do not contribute to the overall functionality of the system. While the main task in this step was to construct and visualize the static call graphs, we first illustrate that code2graph is capable of extracting useful statistics on the code structure.

The static code analysis of the case studies showed that code2graph consists of 13 modules, 15 classes, and a total of 157 functions, out of them 56 functions are entry points and 42 functions are exit points. The Detectron case study consists of 48 modules, 11 classes, and 383 functions. Detectron has 82 entry points and 116 exit points. In contrast, we observed that while Flask was relatively larger in size than Detectron, it had fewer number of modules, 18, and a larger number of classes, 51. It was also obvious that the number of non-class functions in Flask, i.e. 98, is lower than Detectron, i.e. 313. However, the number of methods in Flask, 260, is much larger than those in Detectron, i.e. 70. This might be derived from the programming conventions that different developer teams follow. In addition, Flask has a bigger number of classes and it is, therefore, expected to have a bigger number of methods. The Keras case study was the largest in size among the three other case studies. In particular, Keras includes 61 Python modules, 162 classes and 1,176 functions. The total number of entry points in Keras is 278 compared to 296 exit points. It was obvious that the number of caller-callee relationships was proportional to the number of functions in the system. Code2graph had 198 relationships, Detectron and Flak had close numbers of relationships, i.e. 453 and 494 respectively. Keras, the largest in size, had 4,292 caller-callee relationships. The results of the four case studies are summarized in Table 5.

Table 5: Structure Analysis Results of the Case Studies

Approach	Code2Graph	Detectron	Flask	Keras
# Modules	13	48	18	61
# Classes	15	11	51	162
# Functions	51	313	98	435
# Methods	106	70	260	741
# Entry points	56	82	89	278
# Exit points	42	116	88	298
# Caller-Callee relations	198	453	494	4,292

Once the caller-callee relationships were extracted, we constructed and visualized the call graph for each of the case studies. The call graphs are all listed at the toolâs website `https://info.umkc.edu/UDIC_Research/index.php/wgen/`.

Once the call graph of each case study was generated, we focused on clustering the execution paths into multiple hierarchical clusters. In order to cluster the execution paths, we first had to extract all possible execution paths of the system. The quantitative results of this process are presented in the next subsection. The execution paths are stored in a *csv* file, where each path is given a unique *id* and treated as a set of sequential functions. The number of execution paths ranged from 207 paths in the code2graph case study to 3.7 million paths in Keras.

After extracting all possible execution paths for each case study, we clustered each of the call graphs using the AHC algorithm into multiple levels of abstractions. The number of hierarchical levels differed based on the size and functionality of each case study. The clustering results can be produced in different formats including *dendrograms* for visualization and *csv* for further investigation. For example, Fig. 19 illustrates the dendrogram of the clustered call graph from the code2graph case study. Notice that we have assigned each cluster a unique id, which will be replaced by a proper set of labels in the next step. In addition to the dendrograms, code2graph can produce the clusters data in *csv* format for further investigation by the developer. The rest of the dendrograms for the other case studies are listed at the toolâs website

It is important to mention here that clustering the call graph provided multiple abstraction levels that facilitated understanding the software system at several levels of

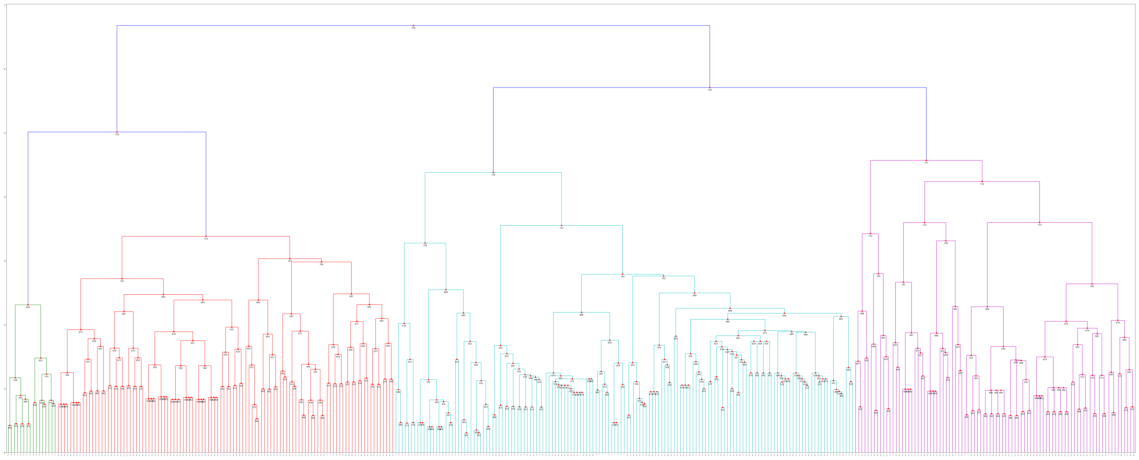


Figure 19: A dendrogram of the clustered call graph from the code2graph case study/

granularity. However, it was evident to us, as developers trying to understand the system, that we still needed to investigate the functions composing each of the clusters in order to understand the overall functionality of the cluster. This requirement was indeed the main motivation to provide proper labels for each of the clusters. Thus, after clustering the entire call graph into hierarchical clusters, we focused on using the technique to provide proper functionality labels for each cluster. We applied the labeling technique in a top-bottom approach for all case studies, in which we first labeled the cluster representing the entire system, then we labeled its two sub-clusters, and so forth. The label of each cluster consisted of the top five major function names in that cluster.

We notice that the top five function names returned by the algorithm can be used to act as a proper label for each cluster. Table 6 presents the labels of two clusters selected randomly form the code2graph case study. For example, the cluster with the id 293 is a very high-level cluster, i.e. two levels from the top. Without a proper label for this cluster, developers need to investigate the low-level functions at the source code in order to figure out the clusterâs overall functionality. In this example, the cluster is made up from almost half of the execution paths in this case study, about 157 paths, with some of them containing up to 8 functions. In contrast, using our system, the top five major terms in this cluster are considered the label for the cluster, including $write\_edge$, $write\_node$, $indx\_node$, $start\_subgraphxx$, and def_d. It is arguable that these terms can provide an overview of the overall functionality of this cluster. We have manually traced the cluster to the execution paths level. We found out that this cluster mainly included execution paths that are used to construct the call graph. Thus, we conclude that the major terms of this cluster are expressive and provide a clue about the overall functionality of the cluster.

Table 6: A sample of the top five labels per Cluster.

Cluster id	Label	tf-idf score
293	write_edge	0.1627
	start_subgraphxx	0.1627
	write_node	0.0697
	indx_node	0.0697
	def_d	0.0697
10	dfs_visitrecurs	0.3333
	def_rmv_backedge	0.3333
	construct_new_graphxx	0.3333
	write	0.3333
	write_count	0.3333

Nevertheless, we have also noticed that one of the top five terms, def_d, is neither expressive nor meaningful. This is derived from the fact that some developers do not follow common naming conventions. Similarly, investigating the top five terms in the cluster with the id 10 demonstrates that it is possible to some extent to derive the overall functionality of the cluster without any prior knowledge about it. The cluster seems responsible to process some kind of a graph using the DFS algorithm, remove the cycles form this graph, and write some data, often, about the graph. With our prior knowledge about the system, we know that this cluster encompasses the functionality responsible for traversing the call graph in a DFS approach to detect the cycles in the graph and write those edges to a specific file. Thus, we conclude that the top five terms used to label each of the clusters can provide a clue about the clusterâs functionality.

We conclude that our approach and tool are feasible to cluster the execution paths in a given call graph into hierarchical levels of abstraction and label each cluster in the hierarchy with meaningful names. In addition, we conclude that clustering the execution paths can indeed facilitate the program comprehension task. The clusters allow developers to investigate the system in different levels of granularity with a proper label for each cluster.

4.4.1 Quantitative Evaluation

In order to evaluate the computational efficiency and scalability of code2graph, we conducted the following quantitative evaluation. In particular, we computed the numbers and time cost for each of the phases done in the previous evaluation, i.e., constructing the static call graph, extracting all execution paths, clustering, and labeling. We evaluated each of the case studies in the same approach. Our case studies range from small size, code2graph case study, to large size, Keras case study. The results are summarized in Table 7.

Table 7: Summary of the Quantitative Analysis Res

Case study	Total number	Execution Paths		Clustering	Labeling
	of functions	total number	time(s)	time (s)	time(s)
code2graph	157	207	0.06	1.38	5.36
Detectron	383	1,407	1.2	10.87	32.89
Flask	358	2,495	1.5	25.63	61.72
Keras	1,176	3.74 million	4,562.04	7,319.18	8,740.81

These experiments were done using a 8-core CPU and 8 GB of RAM. Examining the results in Table 7, we observe that labeling the clusters is the most time consuming task followed by the clustering and finally the paths extraction task. We noticed that for the small to medium size systems, it takes less than 2 seconds to construct a call graph and extract all of its executions paths, which is considered efficient for systems with similar sizes. In particular, constructing the call graph and extracting all of its execution paths required only 0.06 seconds in code2graph, compared to 1.2, 1.5, and 4,562.04 seconds in Detectron, Flask, and Keras, respectively, which is not surprising giving the differences in sizes of the four case studies. Therefore, the time of clustering the execution paths in Keras is much larger than the time of clustering the execution paths in the other three case studies. It is obvious that the clustering task required more time than the paths extraction task. While the times of extracting the execution paths in both Detectron and Flask are very close to each other, the clustering time in Flask is almost as double as the clustering time in Detectron. This is mainly due to the fact that Flask has double the number of execution paths as Detectron.

For the same reason, we noticed that the labeling time in Flask, 61.72 seconds, is almost as double the labeling time in Detectron, 32.89 seconds. Nevertheless, in the first three case studies the system is efficient enough to run an end-to-end static analysis and produce labeled clusters of the call graph in roughly two minutes. However, in the case of large systems, such as Keras, the call graph can be visualized, clustered and labeled in roughly five hours. This process is usually done one time only to create a detailed multilevel abstraction of the system and its functionality. Often, developers need to inspect a particular part of the system and therefore do not need to label all of the clusters. Therefore, we provide features in our tool to allow developers to run specific tasks, such as labeling particular clusters only to save the long running time of labeling the entire system.

Overall, we conclude that our tool is efficient and scalable for large systems. In the case of large systems, the tool can be used off-hours to build the hierarchical clusters of the system, which can then be navigated to understand the software system functionality across multiple levels of granularity.

4.4.2 Threats to Validity

We identify three main validity threats that we aim to address in future work. First, a user study is missing, and it is made our future work. In particular, we aim to release the tool for software developers that were not involved in the research and had no prior knowledge about the tool. The developers will use our tool to understand the behavior and functionality of particular system under investigation. Then, they need to answer a survey about our tool. The goal is to evaluate to what level our tool can help facilitate the program comprehension tasks.

Second, another possible threat to validity is that the case studies evaluated in our system are open-source applications that follow good naming conventions and they had descriptive function names. Therefore, systems with less descriptive functions names are more likely to produce less descriptive and useful labels for the clusters, and vice-versa. However, this is not tightly related to the approach used in our tool, rather the practices followed by the developers themselves. Nevertheless, we plan to explore more advanced machine-learning topic-modeling approaches to label the clusters.

4.5 Related Work

The recent surge in graph data across a variety of scientific domains has led to new tools and methods for understanding and evaluating big graphs. While call graphs have been used to facilitate the tasks of software comprehension, they are still limited to capturing the structure of a a software system at a single level of granularity. To address this limitation, several researchers proposed different visualization tools to help developers better understand the software structure [2, 9, 44, 105]. For example, Wettel et al. [105] and Brito et al. [9] used a city metaphor, where classes or interfaces are represented as buildings and packages are described as districts. The height, width, and color of a building reflect different software matrices such as the taller the building, the higher the number of methods. Similar work by Khalooas et al. [44] used a room to represent one class, and rooms are grouped together based on the codeâs project file structure. Although these tools have been proved to aid developers to comprehend the structure of a system, the dependency among the methods and the classes showing the flow of events in the software system is missing.

Researchers have proposed many reductions and compression techniques to reduce the size of large call graphs. Hamou-Lhaji et al. [35] used filtering techniques to reduce the number of execution paths, which resulted in smaller call graphs and higher abstraction levels of the system. The filtering approach filters out the utility components from the execution paths. Similarly, Cornelissen et al. [15, 14] focused on reducing the graph size by reducing the stack depth of the execution path. While both approaches aim at reducing the call graph size, they may omit essential execution paths from the call graph. In contrast, our approach considers every execution path in the system and focuses on clustering them into several hierarchies to reduce the overall graph size.

Software clustering is one of the commonly used techniques for program comprehension. Several syntactic based clustering techniques have been proposed [62, 40]. These clustering researches are based on static structural dependencies (e.g., inheritance, method calls, references). Other clustering approaches rely on semantic clustering [92, 48, 82], which group the source code in terms of identifier names and comments. Unlike previous software clustering techniques, our approach uses execution paths to guide the clustering of source code entities that perform similar functionality. Execution paths can also reflect the overall system workflow [41].

Walunj et al [103] provided a tool, named GraphEvo, to facilitate understanding software evolution using color-coded call graphs. The tool mainly focused on finding the differences among subsequent call graphs of different versions of the system and colored the added and removed nodes in Green and Red, respectively, to illustrate the code changes using call graphs. However, the overall approach still faces the two main challenges that we addressed in their paper. Moreover, the tool does not provide neither different views of the system nor cluster the complicated call graphs.

Another research direction is to compress and reduce the size of the execution paths. For example, Chan et al. [11] reduced the size of the execution paths using sampling techniques and provided a tool to sample, visualize, and animate the dynamic traces of the system. Riess et al. [77] on the other hand, focused on reducing the size of the execution paths by encoding the repeated ones. The authors used a comparative approach to find similar paths in the system and encode them together, which resulted in reducing the overall size of the graph. While these approaches help reduce the overall size of the execution paths and ultimately the graph, they still provide a single level of granularity only. In contrast, we focus on clustering the execution paths over multiple levels of abstractions. Not only does this approach reduce the size of the graph at each hierarchical level, but it can also help developers to better understand the software system with multiple hierarchical levels of granularity.

Another work that is very similar to our approach is the Sage tool [22]. Sage is a dynamic analysis approach that focuses on building hierarchical abstractions from function calls and can label each hierarchical abstraction with a proper functionality name. Our work focuses on the static analysis of the software system. The dynamic analysis can represent a single execution of the software system based on the input, while the static analysis considers all possible execution paths of the system. Moreover, we focus on building hierarchical abstractions from different abstraction levels of the system. Sage visualizes the identified clusters in vertically stacked layers, where users may lose the position of the visible layer due to the lack of a global overview.

In addition, other studies have focused on software collaborative graphs. For example, [67] examined the software collaborative graphs and found them scale-free, small-world networks similar to those found in other sociological and biological systems. Unlike these tools, our work focuses on *static* code analysis to generate and use *static* call graphs.

CHAPTER 5

MEDL.AI: AN APPLICATION OF MODELKB

5.1 Introduction

Deep learning [37] (DL) has improved the state-of-the-art results in an ever-growing number of medical domains, including image segmentation [104], cancer detection [106], classification [68, 73], and protein structure prediction [19]. Deep learning, a subfield of Artificial Intelligence (AI), is defined as a set of simple but non-linear mathematical functions, known as deep neural networks, that are trained to map raw input data (e.g., patient's symptoms) to the desired output (e.g., diagnosis). Often, the training process in DL is based on observing large amounts of data (e.g., prior diagnosis results) in a process known as supervised learning.

The process of training a DL model is the most expensive and time-consuming in the DL development lifecycle. However, once a DL model is trained, it holds the promise to present high-accuracy prediction results and support existing Clinical Decision Support Systems (CDSS). A key aspect of training a DL model is that it can learn specific patterns from the training data *automatically* with minimal expert intervention. Recent breakthroughs illustrate that DL is capable of outperforming current knowledge-based systems. For example, Ding et al. [17] developed a DL model to predict Alzheimer's disease an average of 75.8 months before the final diagnosis with 82% specificity and 100% sensitivity.

While there has been a surge in developing state-of-the-art DL models specifically in the medical domain, the areas of managing the development lifecycle have mainly been ignored. In particular, it is still challenging to share and reuse trained models efficiently [84, 49, 100]. There exist limited platforms in the medical domain to facilitate exploring, sharing, reusing, testing, and discussing existing DL models. Often, trained models are shared via traditional file repositories or authors' websites. Most of the state-of-the-art models are only published in papers. Even when DL models are shared, it is still challenging to reproduce previous experiments due to missing source code or datasets. Subsequently, it is challenging to explore and reuse existing models, especially for medical applications. This causes limited interpretability and generalizability of the current models and can lead to duplicated efforts on similar medical applications. [94].

In this case study, we present a web-based platform, named *Medl.AI* (Medical applications using DL), where *medl.ai* is the actual website URL. In the rest of this subsection, we use the name *Medl* to refer to our platform for simplicity. *Medl* enables physicians to easily explore, share, and reuse DL models specific for medical applications. It also presents three novel features, including advanced data-driven search, on-the-fly predictions, and discussion boards specific for prediction results. It is important to mention here that *Medl* is not by itself a framework for building or training DL models. The main contributions of *Medl* are the underlying software methodologies that facilitate and accelerate the exchange, reuse, deployment, and testing of DL models tailored for medical applications. The models are trained by physicians and data scientists who voluntarily share them to support open, networked collaboration and research.

We summarize the unique features of *Medl* as follows:

- **Data-driven search:** allows physicians to select an application (e.g., breast cancer detection) and then upload a data instance (e.g., breast measurement). The search will automatically run the input data on all related hosted models and list them ranked by prediction results.

- **On-the-fly prediction:** allows physicians to run low-latency, online predictions without the need to download the model or have prior technical knowledge of DL. Physicians will easily use a specific model by entering their data and then receiving the prediction results. Once the physician decides to use a model, they can submit batches for inference or clone the model for local reuse.

- **Discussion boards:** allow physicians to share and discuss prediction results of edge cases. We envision that the discussion boards will foster physicians' communication, collaboration, and their impression towards the DL applicability and performance. The discussion results will be utilized to improve the models' performance as well during the retraining phase.

5.2 Approach

Medl is composed of five main components: Search, Predict, Discussion Boards, Share, and Metadata Extractor in addition to an intuitive user interface and a database system, as illustrated in Figure 20. Physicians need to register and go through a simple authentication process before they can use *Medl*. After logging in, a physician can explore and search for the hosted models. Once a model is selected, the physician can efficiently run several online predictions and compare multiple results. When a result seems inaccurate, the physician can share and discuss such edge-case predictions with the physicians' community using the discussion boards. After a conclusion is made on the prediction, the physician can mark the prediction as correct or incorrect to improve the model's performance.

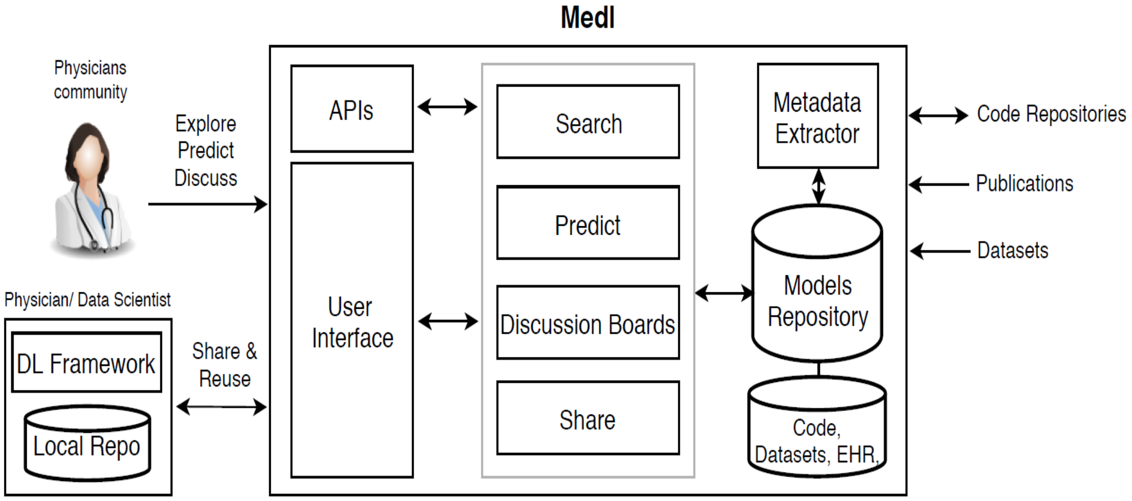
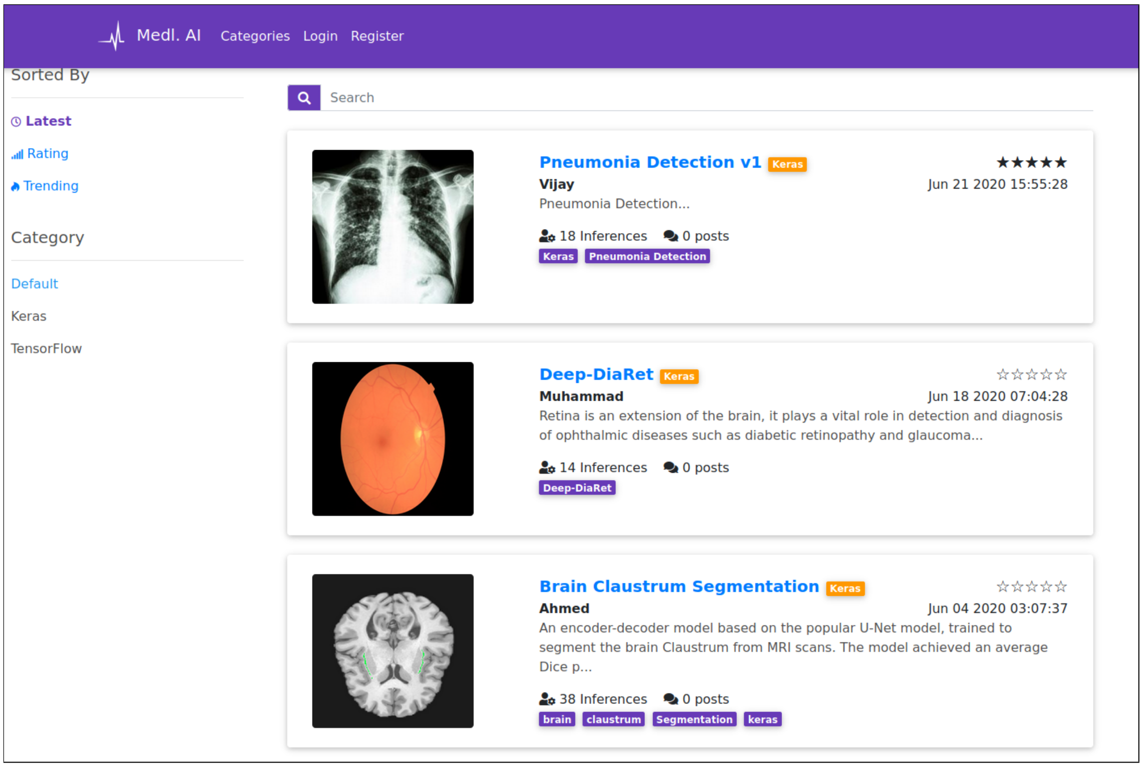


Figure 20: *Medl* System Architecture



Medl. AI Categories Login Register

Sorted By

- Latest
- Rating
- Trending

Category

- Default
- Keras
- TensorFlow

Search

Pneumonia Detection v1 Keras ★★★★★

Vijay Jun 21 2020 15:55:28

Pneumonia Detection...

18 Inferences 0 posts

Keras Pneumonia Detection

Deep-DiaRet Keras ☆☆☆☆☆

Muhammad Jun 18 2020 07:04:28

Retina is an extension of the brain, it plays a vital role in detection and diagnosis of ophthalmic diseases such as diabetic retinopathy and glaucoma...

14 Inferences 0 posts

Deep-DiaRet

Brain Claustrum Segmentation Keras ☆☆☆☆☆

Ahmed Jun 04 2020 03:07:37

An encoder-decoder model based on the popular U-Net model, trained to segment the brain Claustrum from MRI scans. The model achieved an average Dice p...

38 Inferences 0 posts

brain claustrum Segmentation keras

Figure 21: The main view of *Medl* illustrating featured DL models.

The share feature allows physicians and data scientists to exchange models, i.e., publish a model to *Medl* and download existing models for local reuse. The metadata extractor can automatically extract relevant information about the model and its data provenance, which play critical roles in facilitating the exchange, deployment, and reuse of models.

5.2.1 Interactive Web Interface

Medl provides an intuitive, user-friendly web interface accessible to a wide range of physicians despite their technical background. It includes three primary views: the main view, the dashboard view, and the model view. The main view is the landing page, where the recent and most popular models are featured (see Figure 21). This view also allows new users to register, log-in, and navigate to other views. The dashboard view is customized for each user, and it lists a set of user-related activities (e.g., recent discussions). The model view is where individual models are viewed, and we discuss this view in the next subsection.

5.2.2 Model Sharing

Efficient sharing of DL models has a profound effect on the research, reproducibility, accessibility, and applicability of models. [5, 32, 91]. Unlike sharing datasets and source code, there is a scarcity of services that facilitate sharing DL models, including the model's configuration, weights and biases, hyperparameters, and provenance information. The weights and biases are the model parameters whose values can be learned from the data. The hyperparameters are external to the model and are assigned by a DL engineer to determine how the network is structured, e.g., number of layers and the learning rate. Without an easy-to-use method to publish and download DL models, it is almost impossible to reproduce or build upon published results. [32].

In order to share DL models, *Medl* provides two methods: a client-side application named ModelKB [24, 26] and a cloud service named Share. ModelKB is a light-weight tool that aims at automating the management of DL models locally during the training phase, and therefore it can be used to publish trained models from the user's local device to the cloud in one click. ModelKB is not a direct contribution of this case study. The cloud service allows physicians to share their trained models using *Medl*'s web-interface

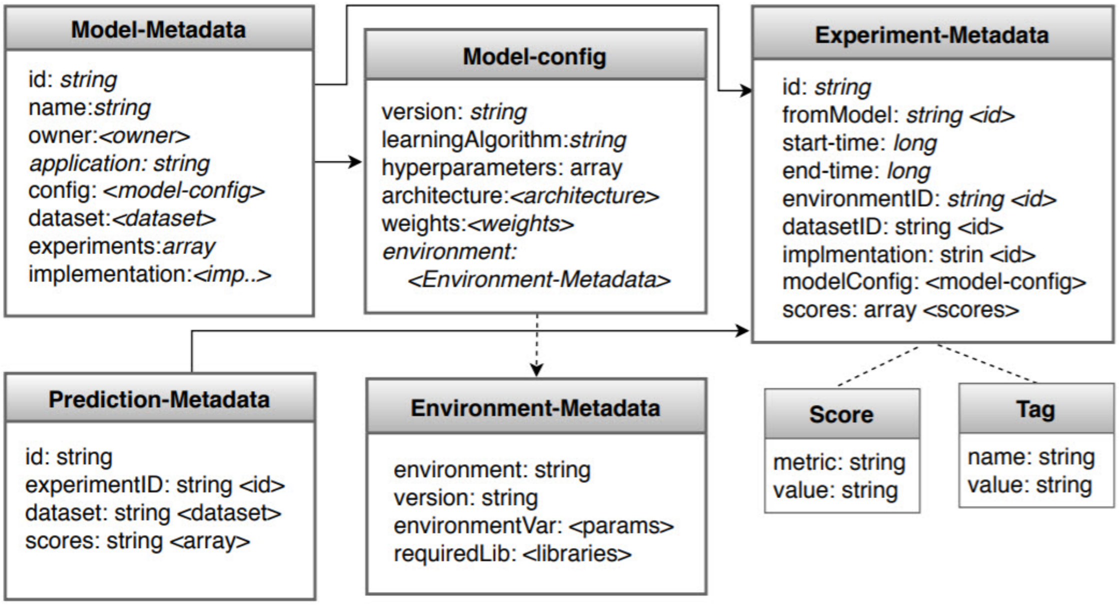
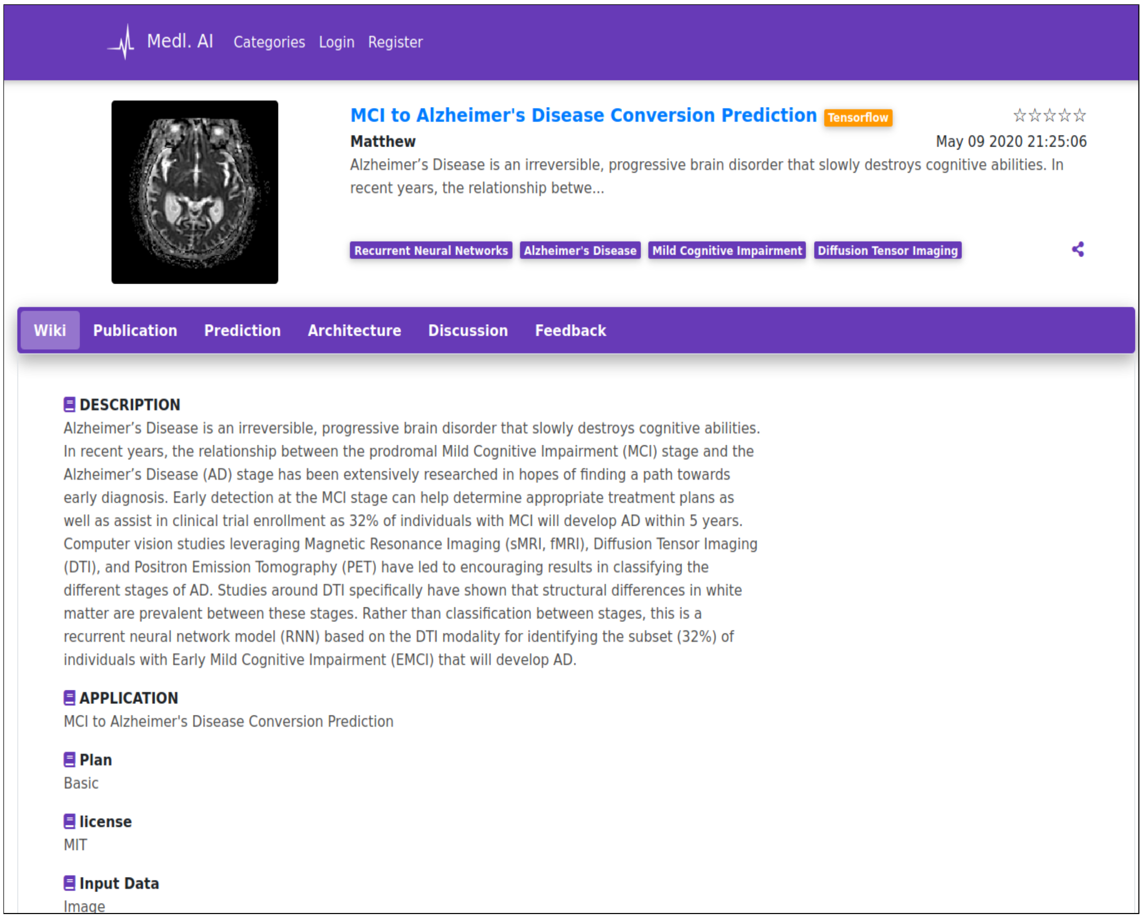


Figure 22: A Simplified version of the data schema used to describe the hosted models.

directly. This requires the user to upload a set of related files. Then, the Metadata Extractor will extract the model's metadata, provide interactive visualizations for the shared models, and populate the model view page with the model's metadata, such as the expected input type. An overview of the data schema used in storing the model's metadata is illustrated in Figure 22. The current implementation applies to models built using TensorFlow, Keras, and PyTorch. The above mentioned features are built with extensibility at their core, so *Medl* can be easily extended to support new DL frameworks, such as and MXNet with minimum user intervention. Similarly, it is a straightforward process to download and reuse the models hosted on *Medl*. If the user uses ModelKB locally, she or he can input the model's ID directly into ModelKB and download it. In contrast, the user can download the model from the web interface as a zip file, including its artifacts.

The models shared on *Medl* are show-cased individually in the model view, which



MCI to Alzheimer's Disease Conversion Prediction

Matthew May 09 2020 21:25:06

Alzheimer's Disease is an irreversible, progressive brain disorder that slowly destroys cognitive abilities. In recent years, the relationship betwe...

DESCRIPTION

Alzheimer's Disease is an irreversible, progressive brain disorder that slowly destroys cognitive abilities. In recent years, the relationship between the prodromal Mild Cognitive Impairment (MCI) stage and the Alzheimer's Disease (AD) stage has been extensively researched in hopes of finding a path towards early diagnosis. Early detection at the MCI stage can help determine appropriate treatment plans as well as assist in clinical trial enrollment as 32% of individuals with MCI will develop AD within 5 years. Computer vision studies leveraging Magnetic Resonance Imaging (sMRI, fMRI), Diffusion Tensor Imaging (DTI), and Positron Emission Tomography (PET) have led to encouraging results in classifying the different stages of AD. Studies around DTI specifically have shown that structural differences in white matter are prevalent between these stages. Rather than classification between stages, this is a recurrent neural network model (RNN) based on the DTI modality for identifying the subset (32%) of individuals with Early Mild Cognitive Impairment (EMCI) that will develop AD.

APPLICATION

MCI to Alzheimer's Disease Conversion Prediction

Plan

Basic

license

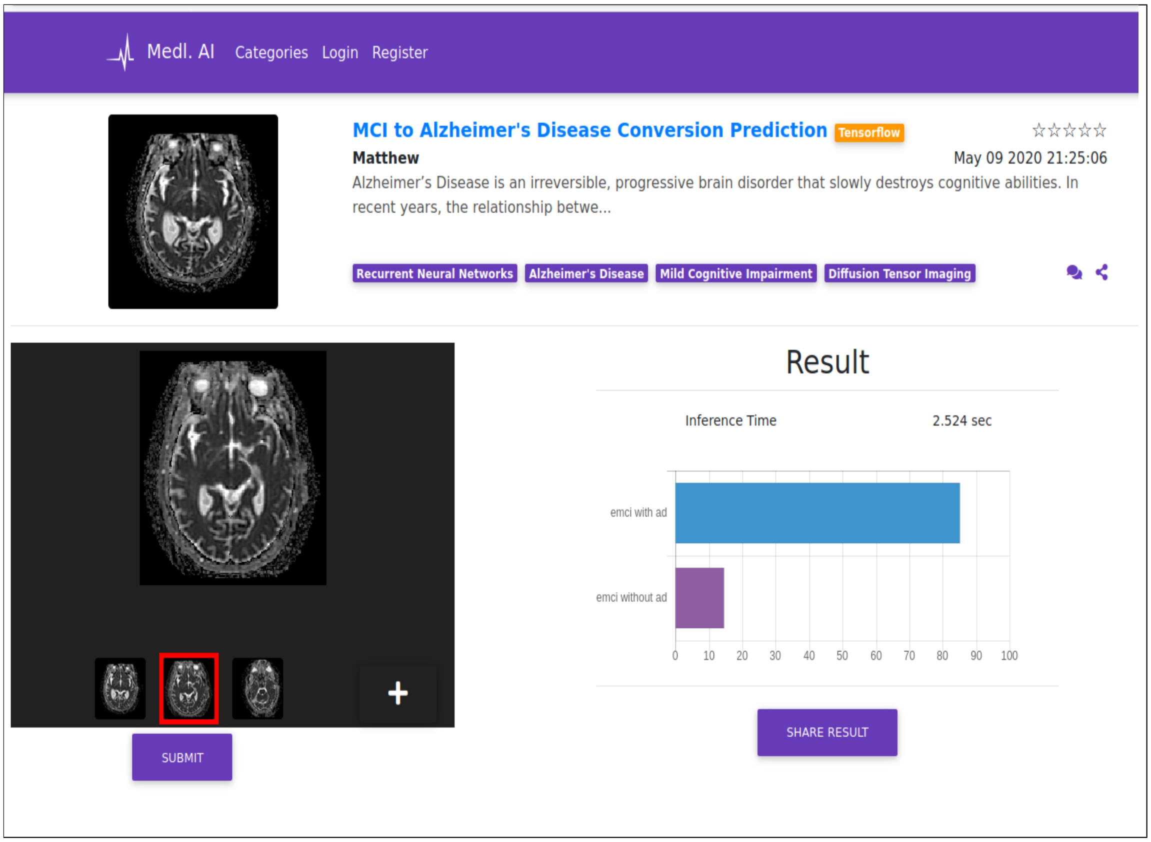
MIT

Input Data

Image

Figure 23: *Medl*'s Wiki View

includes the tabs: Wiki, Publications, Architecture, Prediction, Discussion, and Feedback. The Wiki page describes the model, its application, metadata, underlying algorithm, links to the training dataset, the contributor, how to use the model, and its license. Figure 23 illustrates the Wiki page for one of the currently hosted models, RNN-Based Alzheimer's Disease Prediction .[102]. The Wiki page can also list a wide range of relevant information, such as a link to the used dataset, one of the hosted models used NIH Chest



MCI to Alzheimer's Disease Conversion Prediction Tensorflow

Matthew May 09 2020 21:25:06

Alzheimer's Disease is an irreversible, progressive brain disorder that slowly destroys cognitive abilities. In recent years, the relationship betwe...

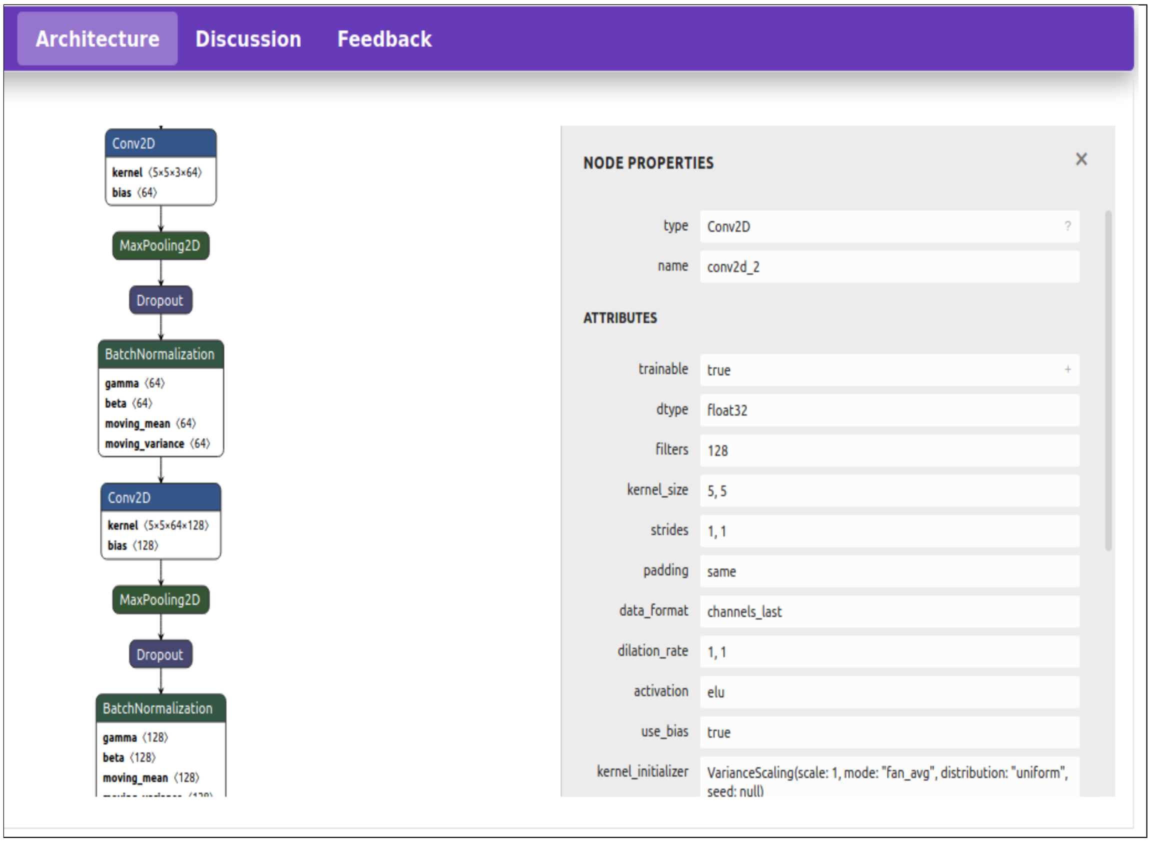
Inference Time 2.524 sec

Figure 24: The Prediction view allows the user to run single predictions using hosted models

X-Rays dataset `https://www.kaggle.com/nih-chest-xrays/data`, for example. *Medl* can automatically generate a reference format to cite the model. The wiki page also includes information about the model's license, selected by the original author, who first uploaded the model to our platform. Thus, you might notice that the code and trained models of some hosted models are not downloadable based on the original owners' settings.

The Publication tab includes the title, abstract, and a link to the paper that published the model–if it exists. The Prediction tab allows the user to directly run predictions on the model from the browser (see Figure 24). This feature will enable physicians to test the models on the fly without the need to download them locally. The Architecture tab uses a third-party open-source tool called Netorn [78] that can automatically load the underlying model and visualize its architecture and per-layer configurations. It provides an intuitive tool to quickly understand the models and its architecture whiteout the need to inspect it manually. Figure 25 illustrates part of the architecture and its configuration for one of the models hosted at *Medl*. The Discussion tab includes a list of all predictions made using the model–only those selected to be shared by the physician. Each prediction is linked to a discussion board, where physicians can discuss, study, and research the result. Unlike the Discussions table that focuses on reviewing inference results, the Feedback tab is specific for rating the model. Users can rate the model and write reviews about it. We also include other metadata in each model that help review the model and its performance, such as the number of inferences that run using a model. This could help ranking models in the future.

Medl currently hosts eight models for medical applications, only two of them were uploaded by the authors as test models, the rest include: Malaria detection [72], liver segmentation from CT scans, [1], Alzheimer disease prediction, [102], liver tumor segmentation from CT scans, [1], brain Clustrum segmentation, [3], diabetic retinopathy detection, and Phenomia detection (a model produced in a Kaggle competition. `https://www.kaggle.com/c/rsna-pneumonia-detection-challenge`).



type	Conv2D
name	conv2d_2
trainable	true
dtype	float32
filters	128
kernel_size	5, 5
strides	1, 1
padding	same
data_format	channels_last
dilation_rate	1, 1
activation	elu
use_bias	true
kernel_initializer	VarianceScaling(scale: 1, mode: "fan_avg", distribution: "uniform", seed: null)

Figure 25: The Architecture view illustrates the model's architecture and per-layer configurations.

5.2.3 Data-Driven Model Search

Practitioners often spend significant amounts of time searching for pre-trained models for inference or reuse in transfer learning. However, due to the lack of centralized repositories, searching for a particular model is tedious and time-consuming. Not only is it challenging to search over multiple websites for a specific model, but often there exist several models for a similar task, each with a distinct set of characteristics (e.g., built using different frameworks or require different input data types and shapes).

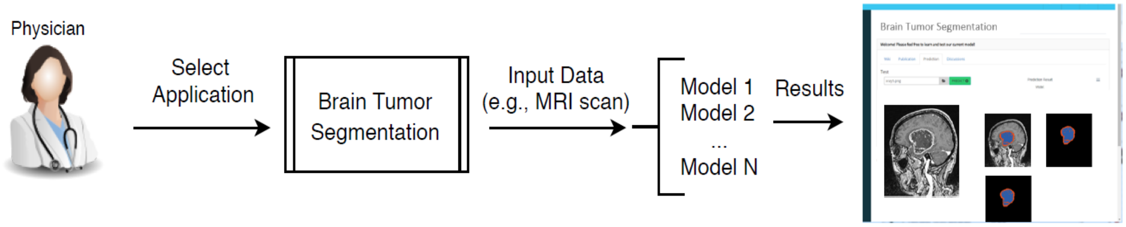


Figure 26: Data-Driven Search Workflow

To facilitate the search task, we developed an innovative data-driven search mechanism. Specifically, the physician needs first to select the task (e.g., brain tumor segmentation) and then upload the data to be examined. *Medl* will automatically run, in parallel, the input data on the available DL models of the selected task, and then return the results sorted based on the prediction accuracy. Figure 26 illustrates the workflow of the search task. Unfortunately, this feature cannot be efficiently utilized without a sufficient number of DL models serving the same or similar applications. As more models with related applications are shared on the platform, physicians will use the data-driven search feature. The current implementation provides keyword-based search mechanisms with various filter types.

5.2.4 On-the-Fly Predictions

One of the essential features of *Medl* is that it allows physicians to perform online predictions (i.e., model inference) without the need to download the DL models or have any prior knowledge on how to deploy them. Once a specific model is selected, the physician can explore its Wiki page to learn how to use it, i.e., the input data type expected by the model, and then navigate to the Prediction tab to run the model. An underlying DL engine, running at the *Medl*'s back-end, is responsible for (1) automatically preparing the data to be consumed by the model, (2) applying model inference using the input data, and (3) visualizing the results to the user in the Prediction view (see Figure 24).

Often, physicians use DL models as a decision support system to back up their results or arrive at a final diagnosis in shorter amounts of time. For example, clinicians may take up to two days to diagnose and classify Diabetic Retinopathy using retinal images. In contrast, using *Medl*, a physician can upload an image and run it using the Diabetic Retinopathy Detection Model to recognize the Diabetic Retinopathy stage using the state-of-the-art model. Notice that the accuracy depends on the used model itself, which our team had not developed rather than hosted on our platform by other physicians. Thus, prediction results may vary and not be fully trusted by physicians. To overcome this issue and provide means for networked science, *Medl* provides the physicians with two options: Share Prediction and Discuss Prediction.

The Share Prediction option allows physicians to quickly, with one click, share the prediction result and artifacts (i.e., the input data, model name, URL, results) with other physicians via the email for further study. The Discuss Prediction option creates a new entry in the *Medl*'s discussion board and shares it with other physicians to discuss such results. We further explain the discussion boards in the next subsection. The current implementation of this feature (running online predictions) allows physicians to make a single inference. However, we have also developed an API equipped with a user-friendly interface that allows the inference of a large number of instances.

5.2.5 Discussion Boards

Medl provides a service that allows the physicians' community to discuss state-of-the-art DL models and their prediction results. For example, edge-case results cannot be verified by the system and require the feedback of expert physicians (e.g., a specific result might yield a positive cancer diagnosis with 50% accuracy). With *Medl*'s discussion boards, such edge cases can be easily shared with thousands of expert physicians, who can share their opinions on the diagnosis.

Each prediction can be shared on the board once the result is visualized. This tab for each model lists all of the predictions that were carried out using the corresponding model (only those that physicians select to share). Each one of the entries (i.e., prediction result and its metadata) corresponds to a discussion board, where that specific entry is discussed. Physicians can create a new reply to the entry or reply to other responses. They can also upload attachments to the discussion board to provide explanatory materials. An essential feature of the discussion board is to up-vote other responses, highlighting more correct opinions. Moreover, the discussion boards provide a great feedback loop to improve the models' performance. Once an edge case has been resolved, the physician can mark the prediction case as correct or incorrect. In both cases, the prediction result, along with other corrected predictions, will be used to periodically and automatically update the existing models. The entries and their discussion boards compose a dataset of edge cases, which can be utilized for further study and research, not to mention the educational benefits. Discussion boards can also lead to exciting research ideas, applications, and networked collaborations.

5.2.6 Automated Vulnerability Testing

A novel feature of *Medl* is its ability to automatically assess the vulnerability of the models being shared through the platform. Specifically, we have implemented an automated engine that can perform two types of attacks on the models uploaded to *Medl* and then provide detailed reports for the authors on the vulnerabilities and how to defend against them. Currently, *Medl* can preform a membership inference attack [88, 81], and an adversarial attack [18, 28]. Since in both cases we will have access to the model, we perform white-box attacks, which–theoretically–increases the accuracy of our attack criteria.

During membership inference attacks, the user is required to run several data objects via our system without letting the system know which data objects were used in training the model to maintain the privacy of the user's data. Our system will classify those inputs as members or non-members of the training dataset. It is left to the author of the model to decide whether our system made correct predictions or not.

In the case of adversarial attacks, our system aims to force the author's model to make a wrong prediction by adding a non-visible noise to the input. Figure 27 illustrates a famous adversarial attack using a method known as Fast Gradient Sign Method (FGSM) [28], which requires access to the model and its gradients. We have implemented the FGSM and integrated it into our system to automatically test the uploaded models before sharing them publicly.

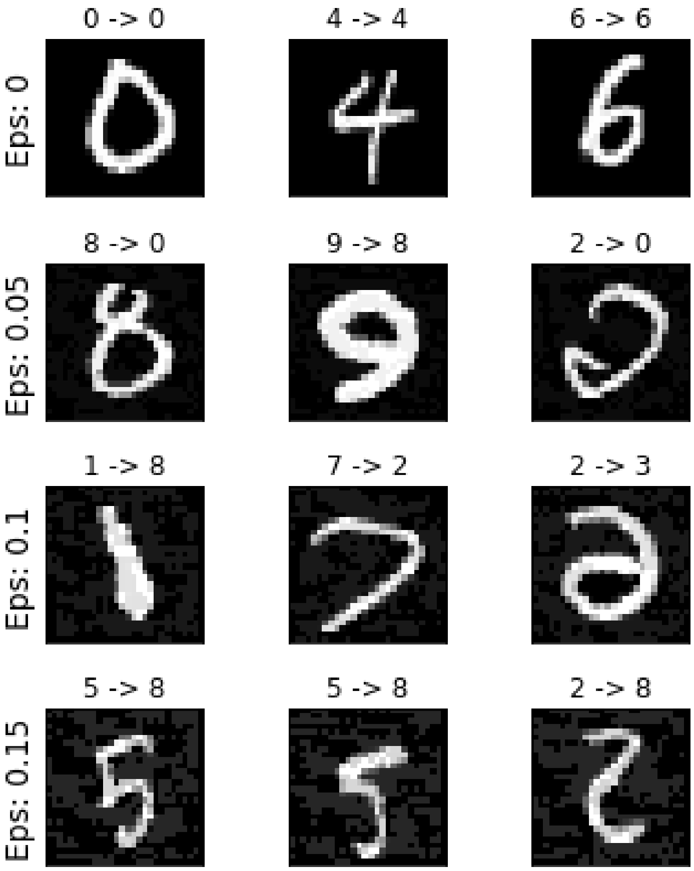


Figure 27: Illustrating the accuracy of a classification model with different values of epsilon in an FGSM attack (Eps = 0 means no changes were made to the input). While the added noise is barely visible to a human eye, the model fails to classify the noisy images. (ground truth ->prediction result)

5.3 Evaluation: Illustrative Applications

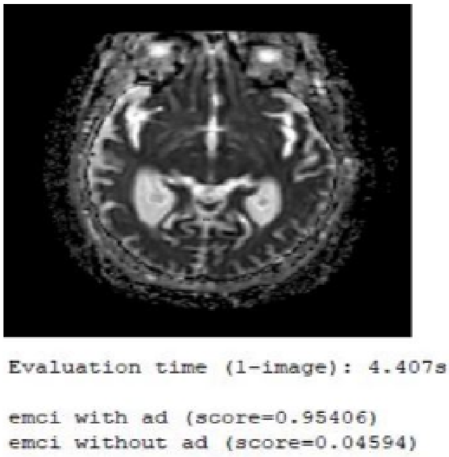
In order to assess the applicability, feasibility, and performance of *Medl*, we conducted (1) a user study and (2) a quantitative analysis for the execution time. The user study included seventeen (17) participants: four Medical Doctors participated as end-users, six industry Machine Learning Engineers and Research Scientists who participated as end-users (both medical doctors and industry users did not upload any models), and seven Ph.D. students working on DL projects, who used our system to share their trained models. None of the users were involved in the development of *Medl*. Three of the participants used Keras, and four of them used TensorFlow. The users used *Medl* to share their trained models, run the models online, search for other models, download models, and used the discussion boards.

In the following, we first illustrate the applications that used *Medl* as an open-collaborative platform to share their models and results. Then, we discuss the survey results. Finally, we present and discuss the platform execution times.

5.3.1 Case Studies

5.3.1.1 Liver and Tumor Segmentation

The first case study used *Medl* to share and expose a trained model for liver and tumor segmentation. The model is trained to segment the human liver and liver tumor from input images extracted from CT scans of the abdominal cavity. The authors of the model used the Liver Tumor Segmentation (LiTS) challenge dataset for training and validation purposes. They achieved a Dice score of 0.894 for liver segmentation and



```
Evaluation time (1-image): 4.407s

emci with ad (score=0.95406)
emci without ad (score=0.04594)
```

Figure 28: Example input/output for AD prediction for an EMCI subject using *Medl.AI*

0.595 for tumor detection. This application is an example of image classification and segmentation tasks.

The authors of this model, a team of Computer Science and Medical students and faculty, used *Medl* to facilitate their collaborative research that leads to the production of the corresponding model. Before *Medl*, the model developers used to manually share–via email–their model results with the faculty supervisor from the medical school. Manually sharing and exchanging information is tedious, error-prone, and time-consuming. Using *Medl*, the authors were able to share their model and expose the platform features with their medical faculty supervisor, who was able to independently run inferences and examine different use cases without requiring any technical background in deep learning.

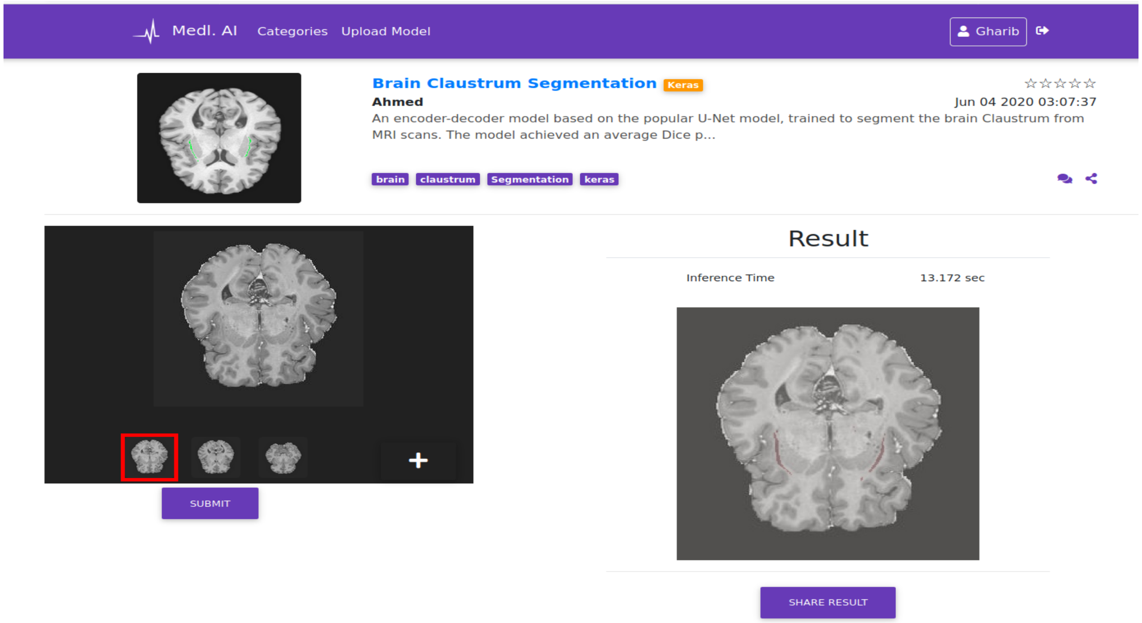
5.3.1.2 MCI to Alzheimer's Disease Conversion Prediction

Computer vision studies leveraging Magnetic Resonance Imaging (sMRI, fMRI), Diffusion Tensor Imaging (DTI), and Positron Emission Tomography (PET) have led to encouraging results in classifying the different stages of Alzhemier's Disease (AD). Studies around DTI correctly have shown that structural differences in white matter are prevalent between these stages. Rather than classification between stages, this is a recurrent neural network model (RNN) based on the DTI modality for identifying the subset (32%) of individuals with Early Mild Cognitive Impairment (EMCI) that will develop AD.

In this use case, the authors developed a new model for Alzheimer's Disease prediction from Prodromal Stage using Diffusion Tensor Imaging, which achieved outstanding performance with accuracy higher than 96% and demonstrated that water molecule diffusion differences could be used to distinguish between patients at-risk for AD. The authors of this work used *Medl* to expose their model for medical doctors as a decision support system. Figure 28 illustrates an input to the model and its results printed our using our proposed system.

5.3.1.3 Brain Clustrum Segmentation

The developer of this use case developed an encoder-decoder model based on the popular U-Net model to segment the brain Clustrum from MRI scans. The model achieved an average Dice per case score of 0.72 for Clustrum segmentation, with K=5 for cross-validation. Figure 29 illustrates how this use case is used to segment Clustrum in MRI scans using *Medl*. This project was a research collaboration between the UMKC's Computer Science department and the School of Medicine. Using *Medl* drastically improved the management, collaboration, the process in which the medical experts were able to run and test the produced model.



Brain Claustrum Segmentation Keras

Ahmed

Jun 04 2020 03:07:37

An encoder-decoder model based on the popular U-Net model, trained to segment the brain Claustrum from MRI scans. The model achieved an average Dice p...

Result

Inference Time 13.172 sec

Figure 29: Brain Clustrum segmentation using *Medl.AI*

5.3.1.4 Deep-DiaRet

The retina is an extension of the brain; it plays a vital role in detecting ophthalmic diseases such as diabetic retinopathy and glaucoma. Timely identification is significant to reduce the rate of blindness. This use case covers a novel deep network model trained and validated over the fundus image dataset to detect and classify diabetic retinopathy in target subjects. This use case used *Medl* to expose the model while its research paper is under review. Using *Medl*, the reviewers of the articles will be able to validate and evaluate the model's claims easily. This is another dimension in which Medl can facilitate and accelerate the papers and their results in the machine learning domain.

5.3.2 Evaluation Objectives

Our user study's overarching goal is to assess the usefulness and applicability of our software system in facilitating and accelerating open-collaboration at the intersections of the medical and deep learning domains. Specifically, our evaluation objective is to investigate the benefits, issues, and difficulties of using *Medl* in deep learning projects. We are specifically interested in whether or not our platform facilitates and accelerates model sharing, exploring, reusing, and maintenance (i.e., fine-tuning trained models on new samples).

5.3.3 Evaluation Methodology

To conduct our evaluation, we developed a set of tasks that applies all features of *Medl* to be carried out by the participants and then asked them to answer a Likert-Scale survey to quantify their feedback. The set of questions used in our survey, developed in Google Forms. Participants respond to each item in the survey by a score from 1 to 5 (1: strongly disagree, 2: disagree, 3: neutral, 4: agree, 5: strongly agree). The tasks included the following steps:

- Step 1: Use *Medl* to explore new deep learning models in the medical domain.
- Step 2: Use *Medl* to run inference using an interesting model and observe its results.
- Step 3: Discuss the results of the model with an expert, either by sharing it directly or adding it to the Discussion boards.
- Step 4: Rate the model using the Feedback feature.

- Step 5: Use *Medl* to host your own deep learning model.

- Step 6: Share your hosted model and test its automatically generated inference function.

- Step 7: Alternatively, download your hosted model as a zipped file and run the inference function locally using the downloaded set of files.

5.3.4 Results

Overall, participants reported that *Medl* provides an imperative, unmet demand in the DL domain. Expressly, all participants Agree (23.5%) and Strongly Agree (76.5%) that it is crucial to facilitate sharing and re-usability of deep learning models in the medical domain. It is was evident that using *Medl* for users who uploaded their models was rated as 41.2% Very Easy and 35.3% as Easy, while only 11.8% answered that using *Medl* was A Little Difficult. On the other hand, 42.4% of participants who used *Medl* to explore existing models and run online inferences Agree that it was easy to use *Medl* and that its interface is user intuitive.

It is essential to mention that users suggested augmenting the Wiki View of the models with more data. However, more than 88% of the users Agree and Strongly Agree that the Wiki and Architecture Visualizations features are useful. Similarly, more than 84% of the participants Agree and Strongly Agree that the online Inference feature is very useful. Specifically, when we also followed up with some of the users, they mentioned that this feature is among the most desirable and helpful features.

Some of the questions targeted at industry and Medical Doctors discussed the platform's future potential and its feasibility. In particular, 52.9% of the participants Strongly Agree that *Medl* has a future possibility to assist medical doctors in finding the next DL model that they can use as a decision support system. Additionally, more than 58% of the users Agree that, overall, *Medl* can help medical doctors interested in ML and DL to easily explore, share, reuse, and understand well-trained DL models.

It was not surprising to notice from the survey that 100% of the users answered Yes to the statement, "I will upload my future DL model to Medl.AI." Moreover, more than 60% of the users who tested other similar systems and that *Medl.ai* is more accessible to the user.

5.3.5 Limitations

The users reported that the current implementation of *Medl* allows running a single prediction at a time only. We argue here that our overarching goal in this case study is to assist physicians in exploring DL models and using them to run on-the-fly predictions as a decision support system and a collaborative science platform. Meanwhile, we have implemented an API with a user-friendly interface that allows running large-scale inference operations, e.g., making hundreds of predictions. However, this feature is not released yet since it requires computational resources (i.e., GPUs) that are not free.

To tackle this problem, we provide two possible solutions: First, to commercialize *Medl* and turn it into a profitable marketplace for medical DL models. In this case, the authors of the models can upload their models to our platform, where other physicians can explore the models and test them on-the-fly. Once they like a specific model, they can pay to use that model based on the number of predictions they make. Second, a not-for-profit organization can support the cause of *Medl* and provide computational resources that allow it to continue serving the medical field for free.

Another current limitation of the platform is that it supports DL models built-in Keras, TensorFlow, and PyTorch only. We argue that extending the platform is a straightforward implementation process that can be achieved by enhancing the Metadata Extractor, and it is made our future work.

5.3.6 Execution Time Analysis

The preliminary experiments of the execution-time were conducted on a Linux Ubuntu 16.04 machine, 16 GB RAM, Intel Core i5-4590 CPU @ 3.30GHz, and NVIDIA GTX 1080 GPU. The local server was implemented using Flask, and the tasks were implemented using Python 3.6. The user interface was developed in JavaScript. The inference process refers to running a single sample through the DL model. The invocation process refers to the entire process of loading the model in the back-end, pre-processing the input data sample, and then running the model. A primary CNN neural network with two hidden layers is used as a benchmark.

Overall, the inference process took an average of 2.3s to run one sample at a time. Meanwhile, the entire invocation process took an average of 8.7s to run the model end-to-end (i.e., from loading to prediction). To better reflect on the required execution time, we have added the execution time for each inference on the platform, and we also visualize

Table 8: Model Execution-Time Results

CNN Image Classification	1.3 MB	Keras	0.66	0.01
Breast Cancer Histology Classification [74]	29.9 MB	Keras	0.47	0.98
Cardiomegaly Classification [45]	57.7 MB	TensorFlow	8.4	1.4
Diabetic Retinopathy Classification [31, 51]	1.44 GB	TensorFlow	25.3	7

that for the user. Please note that our platform currently uses free CPUs at the back-end, and therefore it requires a few more seconds compared to local inference execution time.

5.4 Related Work

Research efforts have recently emerged to address some of the pointed challenges on model sharing and reusability. For example, OpenML [97] is an online platform for machine learning scientists built to facilitate sharing, collaborating, and organizing machine learning models, datasets, and experiments. Model Zoo [64] is an open, collaborative effort to publish machine learning models, where users need to publish their models via traditional file repositories first and then link them to Model Zoo. Kipoi [**kip**] is another community-based effort to facilitate sharing and reusing models for genomics, which currently includes more than 2,000 models of 21 different genomics applications.

A recent effort that focuses on providing a centralized web repository for DL medical applications is ModelHub. [60]. It allows exploring, reusing, and testing the models online, but it is limited to computer vision models. Overall, to the best of our knowledge, only a few existing DL repositories target medical applications; and none of them allows low-latency, on-the-fly testing, and discussion of the models and their prediction results. Moreover, none of the reviewed systems allows users to easily upload their models and

Table 9: A Comparison of Popular DL Model Repositories

	OpenML	**Model Zoo**	**ModelHub**	**Kipoi**	***Medl* (ours)**
Model Sharing	Open	Open	Curated	Curated	Open
Source Code	Native	GitHub	GitHub	GitHub	GitHub
Dataset	Native	External	External	External	External
Domain	General	General	Medical	Genomics	Medical
Search	Yes	None	None	Yes	Data-Driven
Online Inference	Yes	None	Yes	None	Yes
Model Export	Native	Git	Git/Docker	Git/Docker	Git
Metadata	Ad-hoc	Ad-hoc	Ad-hoc	Structured	Structured
Discussions	Ad-hoc	None	None	None	Yes

provide automated deployment at the back-end with minimal user intervention. Table 9 summarizes the related work, in addition to our platform *Medl*, along nine dimensions explained as follows:

- **Model Sharing:** Whether users can contribute trained models (Open) or (Curated).
- **Source Code:** Whether the model's source code is available on the same platform (Native) or not (i.e., the source code resides on other file sharing platforms, such as GitHub).
- **Dataset:** Whether the training dataset is available on the same platform (Native) or not (External: available on other platforms).
- **Domain:** The domain type of the repository (e.g., general, medical).
- **Search:** The type of the search mechanism: domain (e.g., imaging, EHR), task (e.g., classification, prediction, segmentation), metadata (e.g., user, time, accuracy).
- **Model Export:** Whether the repository supports exporting the model or not.

- **Online Inference:** Whether the repository allows online inference using the hosted models or not.

- **Metadata:** Whether or not the repository can extract and visualize the model's metadata (e.g., framework, number of layers) automatically. Ad-hoc refers to limited access to the model's metadata that requires manual intervention from the user. Structured refers to the ability to provide the model's metadata in a structured format automatically without user intervention.

- **Discussions:** Whether the platform allows users to discuss models and predictions or not.

CHAPTER 6

CONCLUSIONS

In this dissertation, we presented and discussed our approach and software system for automating the modeling lifecycle in deep learning. First, we introduced ModelKB, a system that can automatically manage deep learning experiments in their native frameworks across the different modeling phases: training, evaluation, deployment, and sharing. Our overarching goal is to reduce code changes required by data scientists to manage their experiments and accelerate the overall modeling lifecycle. Second, we developed a static analysis tool to facilitate program comprehension using call graphs. We aspire to provide a stepping-stone to facilitate understanding the implementation of experiments and their differences. Third, we develop and publish a real software system, Medl.AI, to automate and accelerate open collaborative research in deep learning medical applications.

Our future work will focus on extending ModelKB and integrate privacy-preserving mechanisms to mitigate data leakage and adversarial attacks. Our design will focus on encapsulating such algorithms form the end-user to enable him/her to completely focus on the modeling tasks.

Bibliography

[1] Ahmed Awad Albishri, Syed Jawad Hussain Shah, and Yugyung Lee. “CU-Net: Cascaded U-Net Model for Automated Liver and Lesion Segmentation and Summarization”. In: *2019 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. IEEE. 2019, pp. 1416–1423.

[2] Mohammad Alnabhan et al. “2D visualization for object-oriented software systems”. In: *2018 International Conference on Intelligent Systems and Computer Vision (ISCV)*. IEEE. 2018, pp. 1–6.

[3] Ahmed Awad Albishri et al. “Automated Human Claustrum Segmentation using Deep Learning Technologies”. In: *arXiv* (2019), arXiv–1911.

[4] James P Bagrow and Erik M Bollt. “An information-theoretic, all-scales approach to comparing networks”. In: *arXiv preprint arXiv:1804.03665* (2018).

[5] Monya Baker. “1,500 scientists lift the lid on reproducibility”. In: (2016).

[6] James Bergstra et al. “Theano: A CPU and GPU math compiler in Python”. In: *Proc. 9th Python in Science Conf.* Vol. 1. 2010, pp. 3–10.

[7] Pamela Bhattacharya et al. “Graph-based Analysis and Prediction for Software Evolution”. In: *Proceedings of the 34th International Conference on Software Engineering*. ICSE ’12. Zurich, Switzerland: IEEE Press, 2012, pp. 419–429. ISBN: 978-1-4673-1067-3.

[8] Matthew Botvinick et al. “Reinforcement learning, fast and slow”. In: *Trends in cognitive sciences* 23.5 (2019), pp. 408–422.

[9] Rodrigo Brito et al. “GoCity: code city for go”. In: *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2019, pp. 649–653.

[10] Davide Castelvecchi. “Can we open the black box of AI?” In: *Nature News* 538.7623 (2016), p. 20.

[11] Andrew Chan et al. “Scaling an object-oriented system execution visualizer through sampling”. In: *11th IEEE International Workshop on Program Comprehension, 2003*. IEEE. 2003, pp. 237–244.

[12] Xi Chen et al. “Infogan: Interpretable representation learning by information maximizing generative adversarial nets”. In: *Advances in neural information processing systems*. 2016, pp. 2172–2180.

[13] François Chollet et al. *Keras*. `https://keras.io`. 2015.

[14] Bas Cornelissen et al. “A systematic survey of program comprehension through dynamic analysis”. In: *IEEE Transactions on Software Engineering* 35.5 (2009), pp. 684–702.

[15] Bas Cornelissen et al. “Understanding execution traces using massive sequence and circular bundle views”. In: *15th IEEE International Conference on Program Comprehension (ICPC’07)*. IEEE. 2007, pp. 49–58.

[16] Andrea De Lucia et al. “Using IR methods for labeling source code artifacts: Is it worthwhile?” In: *2012 20th IEEE International Conference on Program Comprehension (ICPC)*. IEEE. 2012, pp. 193–202.

[17] Yiming Ding et al. “A deep learning model to predict a diagnosis of Alzheimer disease by using 18F-FDG PET of the brain”. In: *Radiology* 290.2 (2019), pp. 456–464.

[18] Yinpeng Dong et al. “Boosting adversarial attacks with momentum”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 9185–9193.

[19] Iddo Drori et al. “High quality prediction of protein q8 secondary structure by diverse neural network architectures”. In: *arXiv preprint arXiv:1811.07143* (2018).

[20] John Ellson et al. “Graphvizâopen source graph drawing tools”. In: *International Symposium on Graph Drawing*. Springer. 2001, pp. 483–484.

[21] Facebook. *Introducing FBLearner Flow: Facebookâs AI backbone*. 2019. URL: `https://conferences.oreilly.com/strata/big-data-conference-ny-2015/public/schedule/detail/42988`.

[22] Yang Feng et al. “Hierarchical abstraction of execution traces for program comprehension”. In: *Proceedings of the 26th Conference on Program Comprehension*. ACM. 2018, pp. 86–96.

[23] Rolando Garcia et al. “Context: The missing piece in the machine learning lifecycle”. In: *KDD CMI Workshop*. Vol. 114. 2018.

[24] Gharib Gharibi, Rakan Alanazi, and Yugyung Lee. “Automatic Hierarchical Clustering of Static Call Graphs for Program Comprehension”. In: *2018 IEEE International Conference on Big Data (Big Data)*. IEEE. 2018, pp. 4016–4025.

[25] Gharib Gharibi, Rashmi Tripathi, and Yugyung Lee. “Code2graph: automatic generation of static call graphs for Python source code”. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM. 2018, pp. 880–883.

[26] Gharib Gharibi et al. “ModelKB: towards automated management of the modeling lifecycle in deep learning”. In: *2019 IEEE/ACM 7th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE)*. IEEE. 2019, pp. 28–34.

[27] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of software engineering*. Prentice Hall PTR, 2002.

[28] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. “Explaining and harnessing adversarial examples”. In: *arXiv preprint arXiv:1412.6572* (2014).

[29] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016.

[30] Google. *TensorBoard: Visualizing Learning*. 2019. URL: https://www.tensorflow.org/guide/summaries%5C_and%5C_tensorbard.

[31] Varun Gulshan et al. "Development and validation of a deep learning algorithm for detection of diabetic retinopathy in retinal fundus photographs". In: *Jama* 316.22 (2016), pp. 2402–2410.

[32] Odd Erik Gundersen and Sigbjørn Kjensmo. "State of the art: Reproducibility in artificial intelligence". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 32. 1. 2018.

[33] Aric Hagberg, Pieter Swart, and Daniel S Chult. *Exploring network structure, dynamics, and function using NetworkX*. Tech. rep. Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[34] Mark Andrew Hall. "Correlation-based feature selection for machine learning". In: (1999).

[35] Abdelwahab Hamou-Lhadj et al. "Recovering behavioral design models from execution traces". In: *Ninth European Conference on Software Maintenance and Reengineering*. IEEE. 2005, pp. 112–121.

[36] Awni Hannun et al. "Deep speech: Scaling up end-to-end speech recognition". In: *arXiv preprint arXiv:1412.5567* (2014).

[37] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[38] Joseph M Hellerstein et al. "Ground: A Data Context Service." In: *CIDR*. 2017.

[39] Edmund Horner. *Constructing Call Graphs*. 2019. URL: https://github.com/davidfraser/pyan.

[40] Syed Islam et al. "Coherent clusters in source code". In: *Journal of Systems and Software* 88 (2014), pp. 1–24.

[41] Wuxia Jin et al. “Functionality-Oriented Microservice Extraction Based on Execution Trace Clustering”. In: *2018 IEEE International Conference on Web Services (ICWS)*. IEEE. 2018, pp. 211–218.

[42] Kaggle. *State of Machine Learning and Data Science 2020*. 2020. URL: https://www.kaggle.com/kaggle-survey-2020.

[43] Andrej Karpathy et al. “Large-scale video classification with convolutional neural networks”. In: *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*. 2014, pp. 1725–1732.

[44] Pooya Khaloo et al. “Code park: A new 3d code visualization tool”. In: *2017 IEEE Working Conference on Software Visualization (VISSOFT)*. IEEE. 2017, pp. 43–53.

[45] Klaggle. *Heart Classification using X-Rays*. 2019. URL: https://www.kaggle.com/kmader/cardiomegaly-pretrained-vgg16/notebook/.

[46] Eleftherios Koutsofios and Stephen C North. “Drawing graphs with dot”. In: (1996).

[47] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[48] Adrian Kuhn, Stéphane Ducasse, and Tudor Gírba. “Semantic clustering: Identifying topics in source code”. In: *Information and Software Technology* 49.3 (2007), pp. 230–243.

[49] Arun Kumar, Matthias Boehm, and Jun Yang. “Data management in machine learning: Challenges, techniques, and systems”. In: *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM. 2017, pp. 1717–1722.

[50] Arun Kumar et al. “Model selection management systems: The next frontier of advanced analytics”. In: *ACM SIGMOD Record* 44.4 (2016), pp. 17–22.

[51] Carson Lam et al. “Automated detection of diabetic retinopathy using deep learning”. In: *AMIA summits on translational science proceedings* 2018 (2018), p. 147.

[52] Steve Lawrence et al. “Face recognition: A convolutional neural-network approach”. In: *IEEE transactions on neural networks* 8.1 (1997), pp. 98–113.

[53] Yann LeCun et al. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[54] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. “Deep learning”. In: *nature* 521.7553 (2015), p. 436.

[55] Yann LeCun et al. “Handwritten digit recognition with a back-propagation network”. In: *Advances in neural information processing systems*. 1990, pp. 396–404.

[56] Guoxu Liu et al. “YOLO-Tomato: A Robust Algorithm for Tomato Detection Based on YOLOv3”. In: *Sensors* 20.7 (2020), p. 2145.

[57] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: `https://www.tensorflow.org/`.

[58] Soumaya Medini et al. “A fast algorithm to locate concepts in execution traces”. In: *International Symposium on Search Based Software Engineering*. Springer. 2011, pp. 252–266.

[59] Hui Miao and Amol Deshpande. “ProvDB: Provenance-enabled Lifecycle Management of Collaborative Data Analysis Workflows.” In: *IEEE Data Eng. Bull.* 41.4 (2018), pp. 26–38.

[60] Hui Miao et al. “ModelHub: Deep Learning Lifecycle Management”. In: *Data Engineering (ICDE), 2017 IEEE 33rd International Conference on*. IEEE. 2017, pp. 1393–1394.

[61] Hui Miao et al. “Towards unified data and lifecycle management for deep learning”. In: *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE. 2017, pp. 571–582.

[62] Brian S Mitchell and Spiros Mancoridis. “On the automatic modularization of software systems using the bunch tool”. In: *IEEE Transactions on Software Engineering* 32.3 (2006), pp. 193–208.

[63] ModelHubAI. *A collection of deep learning models managed by the Computational Imaging and Bioinformatics Lab at the Harvard Medical School, Brigham Women’s Hospital, and Dana-Farber Cancer Institute*. 2019. URL: `http://modelhub.ai/`.

[64] ModelZoo. *A set of pretrained models models hosted on GitHub*. 2019. URL: `https://github.com/BVLC/caffe/wiki/Model-Zoo`.

[65] Grégoire Montavon, Wojciech Samek, and Klaus-Robert Müller. “Methods for interpreting and understanding deep neural networks”. In: *Digital Signal Processing* 73 (2018), pp. 1–15.

[66] Gail C Murphy et al. “An empirical study of static call graph extractors”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 7.2 (1998), pp. 158–191.

[67] Christopher R Myers. “Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs”. In: *Physical Review E* 68.4 (2003), p. 046116.

[68] Kamyar Nazeri, Azad Aminpour, and Mehran Ebrahimi. “Two-stage convolutional neural network for breast cancer histology image classification”. In: *International Conference Image Analysis and Recognition*. Springer. 2018, pp. 717–726.

[69] Nvidia. *DIGITS: A graphical web interface for NVCaffe and TensorFlow*. 2019. URL: https://docs.nvidia.com/deeplearning/digits/digits-user-guide/index.html.

[70] Adam Paszke et al. "Automatic Differentiation in PyTorch". In: *NIPS Autodiff Workshop*. 2017.

[71] PyTorch. *A set of pretrained PyTorch models*. 2019. URL: https://pytorch.org/docs/stable/torchvision/models.html.

[72] Sivaramakrishnan Rajaraman et al. "Pre-trained convolutional neural networks as feature extractors toward improved malaria parasite detection in thin blood smear images". In: *PeerJ* 6 (2018), e4568.

[73] Alexander Rakhlin et al. "Deep convolutional neural networks for breast cancer histology image analysis". In: *International Conference Image Analysis and Recognition*. Springer. 2018, pp. 737–744.

[74] Alexander Rakhlin et al. "Deep convolutional neural networks for breast cancer histology image analysis". In: *International Conference Image Analysis and Recognition*. Springer. 2018, pp. 737–744.

[75] Juan Ramos et al. "Using tf-idf to determine word relevance in document queries". In: *Proceedings of the first instructional conference on machine learning*. Vol. 242. New Jersey, USA. 2003, pp. 133–142.

[76] Joseph Redmon and Ali Farhadi. "Yolov3: An incremental improvement". In: *arXiv preprint arXiv:1804.02767* (2018).

[77] Steven P Reiss and Manos Renieris. "Encoding program executions". In: *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*. IEEE. 2001, pp. 221–230.

[78] Lutz Roeder. *Netron: Visualizing Deep Learning Models*. 2019. URL: https://github.com/lutzroeder/netron.

[79] Pablo Romero-Fresco. *Subtitling through speech recognition: Respeaking*. Routledge, 2020.

[80] Olga Russakovsky et al. "ImageNet Large Scale Visual Recognition Challenge". In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252. DOI: 10.1007/s11263-015-0816-y.

[81] Ahmed Salem et al. "Ml-leaks: Model and data independent membership inference attacks and defenses on machine learning models". In: *arXiv preprint arXiv:1806.01246* (2018).

[82] Gustavo Santos, Marco Tulio Valente, and Nicolas Anquetil. "Remodularization analysis using semantic clustering". In: *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. IEEE. 2014, pp. 224–233.

[83] Tom Schaul, Sixin Zhang, and Yann LeCun. "No more pesky learning rates". In: *International Conference on Machine Learning*. 2013, pp. 343–351.

[84] Sebastian Schelter et al. "On Challenges in Machine Learning Model Management." In: *IEEE Data Eng. Bull.* 41.4 (2018), pp. 5–15.

[85] Sebastian Schelter et al. "Automatically tracking metadata and provenance of machine learning experiments". In: *Machine Learning Systems Workshop at NIPS*. 2017.

[86] David Sculley et al. "Hidden technical debt in machine learning systems". In: *Advances in neural information processing systems*. 2015, pp. 2503–2511.

[87] Frank Seide and Amit Agarwal. "CNTK: Microsoft's open-source deep-learning toolkit". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM. 2016, pp. 2135–2135.

[88] Reza Shokri et al. "Membership inference attacks against machine learning models". In: *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2017, pp. 3–18.

[89] David Silver et al. “Mastering the game of Go with deep neural networks and tree search”. In: *nature* 529.7587 (2016), p. 484.

[90] Karen Simonyan and Andrew Zisserman. “Very deep convolutional networks for large-scale image recognition”. In: *arXiv preprint arXiv:1409.1556* (2014).

[91] Victoria Stodden et al. “Enhancing reproducibility for computational methods”. In: *Science* 354.6317 (2016), pp. 1240–1241.

[92] Xiaobing Sun et al. “Clustering classes in packages for program comprehension”. In: *Scientific Programming* 2017 (2017).

[93] Christian Szegedy et al. “Inception-v4, inception-resnet and the impact of residual connections on learning.” In: *AAAI*. Vol. 4. 2017, p. 12.

[94] Nozomi Takeshima et al. “Which is more generalizable, powerful and interpretable in meta-analyses, mean difference or standardized mean difference?” In: *BMC medical research methodology* 14.1 (2014), p. 30.

[95] Chakkrit Tantithamthavorn, Ahmed E Hassan, and Kenichi Matsumoto. “The impact of class rebalancing techniques on the performance and interpretation of defect prediction models”. In: *IEEE Transactions on Software Engineering* (2018).

[96] Uber. *IMeet Michelangelo: Uberâs Machine Learning Platform*. 2019. URL: https://eng.uber.com/michelangelo/.

[97] Jan N Van Rijn et al. “OpenML: A collaborative science platform”. In: *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer. 2013, pp. 645–649.

[98] Manasi Vartak. “Infrastructure for model management and model diagnosis”. PhD thesis. Massachusetts Institute of Technology, 2018.

[99] Manasi Vartak and Samuel Madden. “MODELDB: Opportunities and Challenges in Managing Machine Learning Models.” In: *IEEE Data Eng. Bull.* 41.4 (2018), pp. 16–25.

[100] Manasi Vartak et al. “M odel DB: a system for machine learning model management”. In: *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*. ACM. 2016, p. 14.

[101] Rajesh Vasa, Jean-Guy Schneider, and Oscar Nierstrasz. “The inevitable stability of software change”. In: *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*. IEEE. 2007, pp. 4–13.

[102] Matthew Velazquez et al. “RNN-Based Alzheimer’s Disease Prediction from Prodromal Stage using Diffusion Tensor Imaging”. In: *2019 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. IEEE. 2019, pp. 1665–1672.

[103] Vijay Walunj et al. “GraphEvo: Characterizing and Understanding Software Evolution using Call Graphs”. In: *2019 IEEE International Conference on Big Data (Big Data)*. IEEE. 2019, pp. 4799–4807.

[104] Guotai Wang et al. “Automatic brain tumor segmentation using cascaded anisotropic convolutional neural networks”. In: *International MICCAI brainlesion workshop*. Springer. 2017, pp. 178–190.

[105] Richard Wettel. “Visual exploration of large-scale evolving software”. In: *2009 31st International Conference on Software Engineering-Companion Volume*. IEEE. 2009, pp. 391–394.

[106] Nan Wu et al. “Deep neural networks improve radiologistsâ performance in breast cancer screening”. In: *IEEE transactions on medical imaging* 39.4 (2019), pp. 1184–1194.

[107] Xiang Yu, Kihyuk Sohn, and Manmohan Chandraker. *Video security system using a siamese reconstruction convolutional neural network for pose-invariant face recognition*. US Patent App. 15/803,318. May 2018.

[108] Matei Zaharia et al. “Accelerating the Machine Learning Lifecycle with MLflow”. In: *Data Engineering* (2018), p. 39.

[109] Aston Zhang et al. *Dive into Deep Learning*. `http://www.d2l.ai`. 2019.

VITA

Gharib Gharibi is a Ph.D. candidate at UMKC and an Applied Scientist at Triple-Blind Inc. His interests include privacy-preserving Machine Learning (ML), ML systems, and AI applications in healthcare. He also teaches Deep Learning at Nvidia Deep Learning Institution. Gharib has built several software tools to aid software comprehension. He published several papers in world-ranked journals and conferences, published a book chapter on text generation in Arabic, and filed several patents in privacy-preserving ML in 2020. You can find more information at Gharib's personla website `https://sites.google.com/view/ggharibi`.