Empirical Software Engineering (2021) 26: 17
https://doi.org/10.1007/s10664-020-09894-9

Automated end-to-end management of the modeling lifecycle in deep learning

Gharib Gharibi[1] · Vijay Walunj[1] · Raju Nekadi[1] · Raj Marri[1] · Yugyung Lee[1]

Accepted: 27 November 2020 / Published online: 19 February 2021

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 and training 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 deployment. In this paper, 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. Moreover, our system 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.

Keywords Data management · Deep learning · Software automation

1 Introduction

Deep learning (LeCun et al. 2015; Goodfellow et al. 2016), a subfield of Machine Learning, has improved the state-of-the-art results in an ever-growing number of domains such as computer vision (He et al. 2016; Szegedy et al. 2017), speech recognition (Hannun et al. 2014), and reinforcement learning (Silver et al. 2016). A *Deep learning model*, also known

Communicated by: Tim Menzies, Chakkrit Tantithamthavorn and Burak Turhan

This article belongs to the Topical Collection: *Software Engineering in the Age of Artificial Intelligence*

✉ Gharib Gharibi
ggk89@mail.umkc.edu

Extended author information available on the last page of the article.

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 (Hall 1999), deep learning is an end-to-end learning approach that can automatically learn the features from the input data (Goodfellow et al. 2016). 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 (Castelvecchi 2016; Vartak and Madden 2018b; Garcia et al. 2018).

Due to limited knowledge on the meaning of well-trained DNN parameters (i.e., weights and biases), training a deep learning model 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, 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 (Ghezzi et al. 2002), the heuristic and iterative nature of deep learning presents a new set of challenges–particularly **managing a large number of experiments and artifacts throughout their lifecycle** (Sculley et al. 2015; Miao et al. 2017b; Kumar et al. 2017; Schelter et al. 2018a; Vartak and Madden 2018b). We define *deep learning lifecycle management* as the task of tracking, 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.

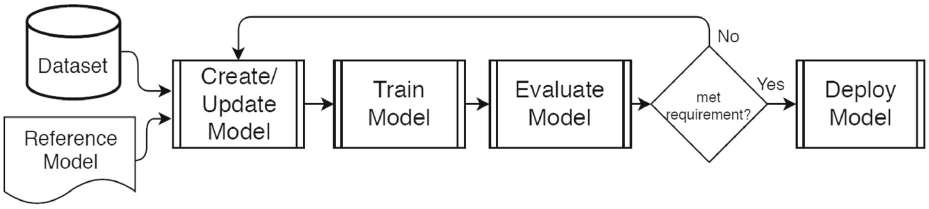


Fig. 1 Deep Learning Development Lifecycle. Adopted from Miao et al. (2017b)

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 (Bergstra et al. 2010), TensorFlow by Google (Abadi et al. 2015), CNTK by Microsoft (Seide and Agarwal 2016), PyTorch by Facebook (Paszke et al. 2017), and Keras (Chollet and et al 2015). 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 tasks. 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.

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 (Vartak et al. 2016), ModelHub (Miao et al. 2017a), ProvDB (Miao and Deshpande 2018; Schelter et al. 2017), Ground (Hellerstein et al. 2017; Kumar et al. 2016), and our own previous work (Gharibi et al. 2019a, b). 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 tools. For example, FBLearner Flow for PyTorch (Facebook 2019), TensorBoard for TensorFlow (Google 2019), Digits by Nvidia (Nvidia 2019), and Michelangelo by Uber (Uber 2019). However, these systems restrict the user to specific frameworks, processing pipelines, and deployment options. Other management systems include tools that recently emerged from industry startups that realized the critical need for management systems, such as CometML (https://www.comet.ml/), Weights and Biases (https://www.wandb.com), Neptune (https://neptune.ml/), and MLFlow (Zaharia et al. 2018). 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. Our approach's novelty crystallizes in unifying the lifecycle management 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, track their evolution, facilitate reproducibility, accelerate deployment, and enhance collaboration, i.e., model sharing and publishing. This allows data scientists to focus on the modeling process without worrying about managing their experiments or being 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 simple inference tasks; and (4) reproduce models when needed through model sharing and publishing. ModelKB is a stand-alone Python library. While not a direct contribution of this paper, 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.

Automatic extraction and tracking of the metadata are 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 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.

In order to assess the usefulness of ModelKB in managing the overall modeling lifecycle, we conducted a user study followed by a survey to collect users' 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 simple 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.

The rest of this paper is organized as follows: We present a brief background of deep learning and its modeling challenges in Section 2. We introduce our approach in Section 3. The implementation of ModelKB and its evaluation are presented in Sections 4 and 5, respectively. We present the current related work in Section 6. We conclude our paper in Section 7 and briefly discuss our future work in Section 8.

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–i.e., 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 nonlinear 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

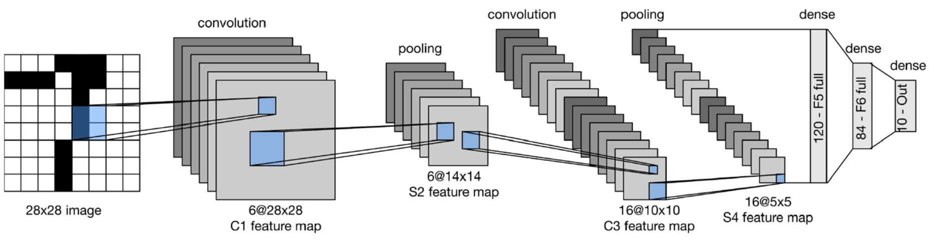


Fig. 2 Example of a convolutional DNN, LeNet-5, LeCun et al. (1998). The input is an image of hand-written digit (0-9), the output is a probability over 10 possible outcomes (classes), which predicts the digit written on the image. Diagram credit: Zhang et al. (2019)

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 data without human intervention (*feature engineering* Hall 1999), 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 (Montavon et al. 2018; Tantithamthavorn et al. 2018; Chen et al. 2016).

Figure 2 illustrates a sample of a classical convolutional neural network (CNN), called LeNet, which was developed by Yann LeCun to recognize hand-written digit images (Lecun et al. 1990, 1998). CNNs are mostly used in computer vision applications (Lawrence et al. 1997; Krizhevsky et al. 2012; Karpathy et al. 2014; Yu et al. 2018). 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, Fig. 2 illustrates the architecture of LeNet and its configuration. The number of layers, their configuration, the learning rate, 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 (Russakovsky et al. 2015) each had a distinct set of hyperparameters and a unique architecture, including ResNet (He et al. 2016), AlexNet (Krizhevsky et al. 2012), VGG (Simonyan and Zisserman 2014), and LeNet (LeCun et al. 1990).

2.2 Deep Learning Modeling Lifecycle

While both software and machine learning aim to introduce efficient and reliable solutions, their overall development lifecycles have differences that require diverse management approaches and toolsets. This subsection focuses on illustrating the development difference between machine learning and traditional software engineering. The goal is to shed light on the main machine-learning *development differences* that cause its management challenges versus traditional software development.

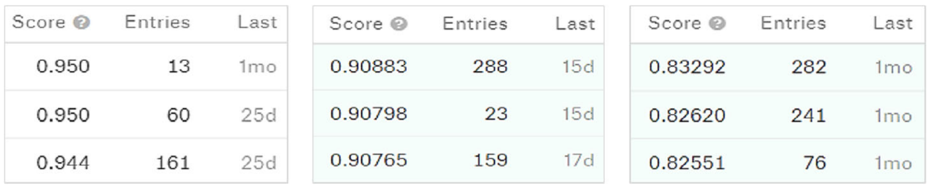
Unlike traditional software development that aims at meeting a set of well-defined functional and non-functional requirements, the process of training an efficient deep learning model aims at optimizing a specific metric (e.g., minimizing a loss function). Table 1 illustrates some of the differences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

Table 1 A brief comparison between conventional software development and Machine Learning development

	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

learning development, which motivate the need for novel software tools that facilitate the management of the machine learning lifecycle. In the Table, we focus on the differences only, not the commonalities. For example, both tasks' goal is to create useful applications that help the end-user, but here we point to the differences that introduce management challenges. Note that "Goal" in the Table refers to the development objective, not the overall process of creating a software or a model. Additionally, "Quality" refers to the leading quality differences in both cases. For example, executing the same implementation (code) on a given input will always produce the same results in a traditional software tool. This is not true in machine learning, where the quality of the final output, during the training phase, depends on several other factors than the code implementation, such as random idealization of the DNN. Thus, we consider that the quality of traditional software development depends mainly on the quality of its implementation, while it depends on additional factors in the case of machine learning development. Additionally, due to limited knowledge of the meaning of well-trained models, the model development 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, Fig. 3 illustrates a real-world scenario of a 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 competition deadline. In contrast, a typical software engineering hackathon includes a single submission per team.

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



Score	Entries	Last
0.950	13	1mo
0.950	60	25d
0.944	161	25d

Score	Entries	Last
0.90883	288	15d
0.90798	23	15d
0.90765	159	17d

Score	Entries	Last
0.83292	282	1mo
0.82620	241	1mo
0.82551	76	1mo

Fig. 3 An example illustrating the large number of models (*Entries*) submitted by top three winners of three randomly selected Kaggle competitions

- *Data Preparation:* Collecting and preparing the datasets needed to develop the model.
- *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 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.

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 a significant role in informing the next set of experiments and identifying best-performing models. However, without a proper management system, experiments and their metadata are lost, and valuable time and resources are wasted–sometimes even running the same experiments. Subsequently, the lack of an adequate management system for deep learning has revealed 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. It is currently challenging to track each experiment, model evaluation, analyze abnormal behaviors, identify data provenance, and reason for the produced results. The large number of involved metadata elements make it challenging to track experiments manually, and it is 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 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, to run predictions using the deployed model (inference), ingested data need to go through the same preprocessing steps 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 pre-trained PyTorch models (PyTorch 2019), or raw files via traditional file repositories such as Model Zoo on GitHub (ModelZoo 2019). While some repositories have recently emerged, such as ModelHubAI (2019), these repositories are manually curated by the owners or require significant manual efforts to share the trained models.

3 ModelKB: Approach

Figure 4 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

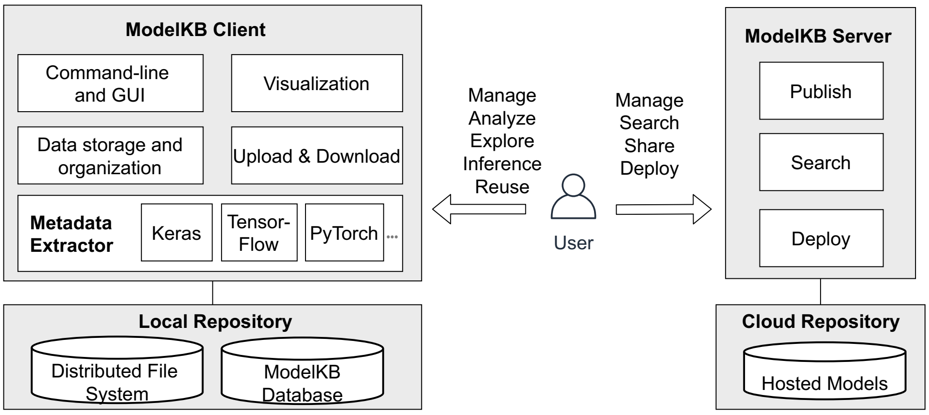


Fig. 4 ModelKB System Architecture

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) (Documentation 2019) for tracking data province, and *Jinja* (Jinja 2019) 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.

As shown in Fig. 4, 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 2019) 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 prediction functions that expose the trained models for 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.

3.1 Automatic Metadata Extraction

The availability of metadata constitutes a concrete stepping-stone that enables the rest of our system's tasks. 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* refers to variables that govern the learning algorithm and are set by the user, such as optimization algorithm, learning rate, loss function, and the number of epochs. *Parameters* refers to the variables learned by the algorithm–specifically weights and biases. Furthermore, we use the term *Context* metadata to refer to additional data about

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

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.

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 a given time. The execution of Callbacks can be initiated at 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 1 illustrates a snippet of our callbacks implementation for Keras.

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 2 illustrates how to use ModelKB to manage the modeling experiments in the case of Keras, which only requires importing the ModelKB library and then passing our pre-defined Callbacks. From Listing 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

```
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('batch_size')
        # 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 1 A snippest of the Keras callback

```
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, batch_size=32, epochs=15,
     callbacks=[KerasCallbacks(my_exp)])
# rest of the code
```

Listing 2 A code snippet illustrating how to use ModelKB in Keras

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.

Another contribution of our work is the hierarchical organization of deep learning projects. Each project consists 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 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. Then, as long as the project name is not changed, all experiments created under that project will be grouped together. This holds 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 the 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 the 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 predefined high-level functions. 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 using the defined model, `pred = model(training_data)`, 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, to facilitate the usage of ModelKB within PyTorch, we first implemented a high-level function wrapper, *fit*, and then developed the necessary Callbacks to enable metadata extraction.

While not an explicit 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. On 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. We did not reinvent the wheel in this work, 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 Keras' implementation to facilitate reusing/extending our already-developed Callbacks for PyTorch.

Listing 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.

An essential feature of our approach is the ability to unify the seed value among experiments–if needed. It is challenging to automatically obtain the seed value from the underlying system during the training phase. If known and managed correctly, the seed value can 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 the proper tools to track the seed values across different experiments.

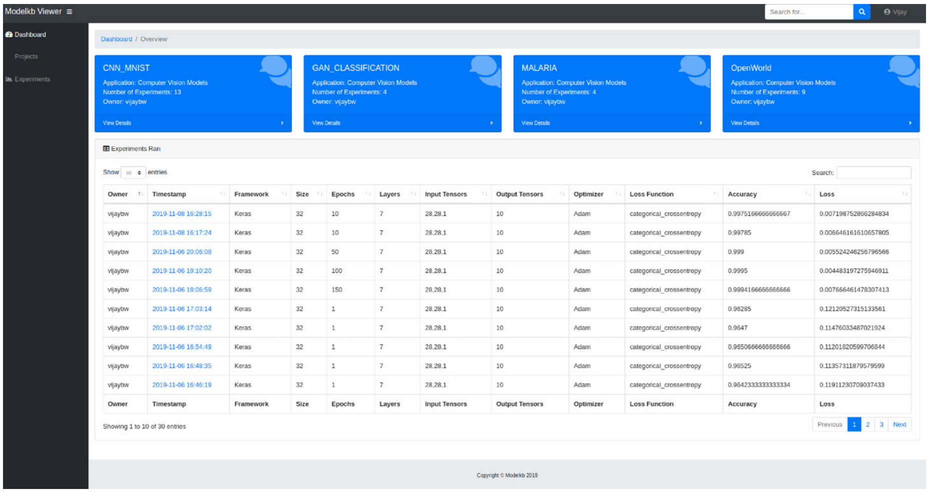
At the end of the training cycle, we organize the extracted metadata into dictionaries and store them in a local database along with the model file (i.e., model architecture and parameters) and implementation files (source code). The model file is then parsed to visualize its architecture and per-layer configuration. Then, the deployment functionality is invoked to generate a prediction function for the inference task 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.

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

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

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

Listing 3 A code snippet illustrating how to use ModelKB in PyTorch using our customized *fit* function



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

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

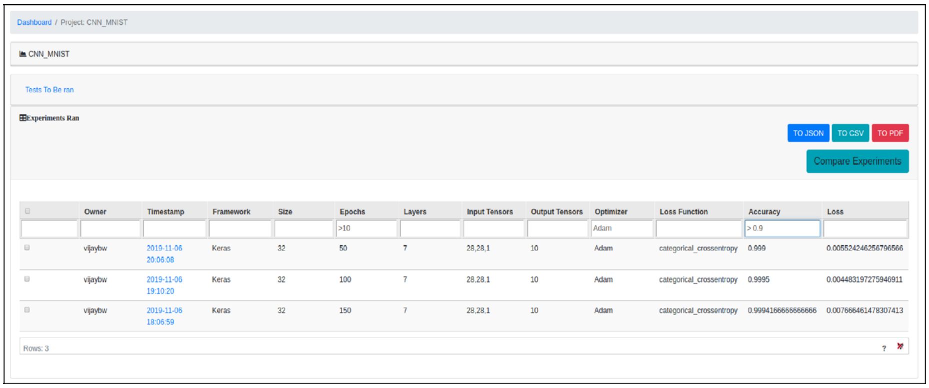
Fig. 5 The Dashboard of ModelKB (landing page)

3.2 Data Visualization

ModelKB provides a local web-based visualization that allows users to explore, analyze, compare, and run 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 requires little to no external dependencies. Moreover, Flask is popular and has an outstanding support community, making it easy to extend and debug (Grinberg 2018).

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 Fig. 5) 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 particular project will lead to the Project view, which we explain as follows.

The Project view, illustrated in Fig. 6, 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 created between two specific dates. For example, in Fig. 6, 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 dataset). Clicking on any of the experiments will lead to the experiment view. Note that users can download those tables in



	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

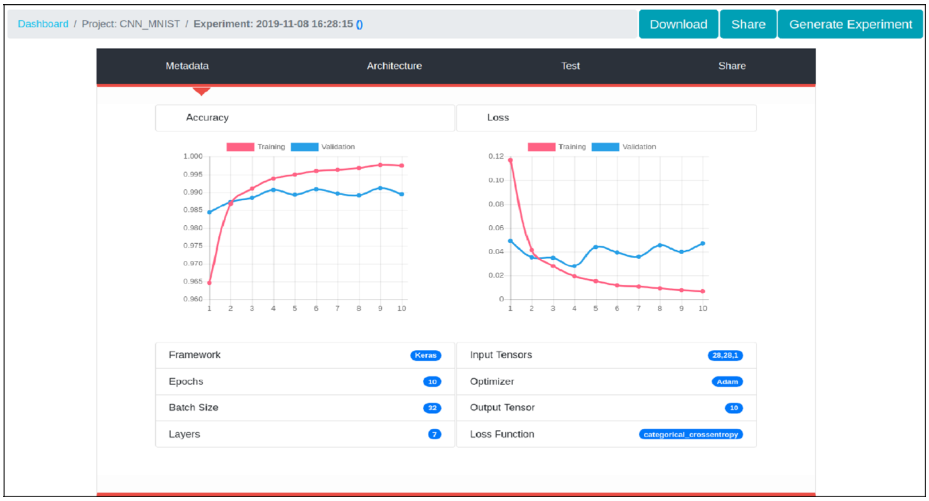
Rows: 3

Fig. 6 The Project view lists the experiments and their metadata related to a single project

several different formats, including *json*, *csv*, *pdf*, and we are currently working on supporting LaTeX, in order to facilitate exporting the metadata from the GUI in different formats, rather than using copy-and-paste.

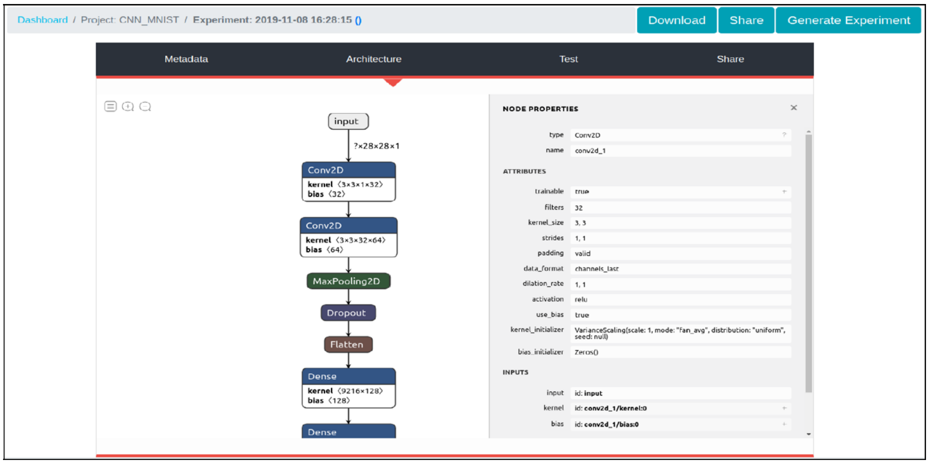
The Experiment view focuses on visualizing one experiment at a time, and it includes four tabs: *Metadata*, *Architecture*, *Inference*, and *Share*. The Metadata tab lists all the metadata that the user selects to view, including plots for the accuracy and loss scores over time for both training and validation datasets–if available. Figure 7 illustrates the Metadata tab in the Experiment view.

The Architecture tab visualizes the model's architecture 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* (Roeder 2019), which we integrated into ModelKB. Figure 8 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



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

Fig. 7 The Experiment-Metadata view visualizes the selected metadata, including for loss and accuracy



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()

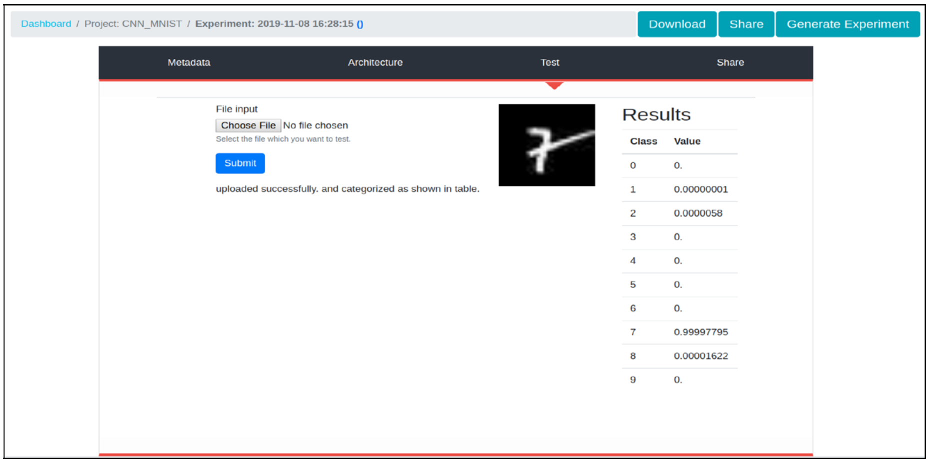
INPUTS

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

Fig. 8 The Architecture view visualizes the neural network architecture and per-layer configurations

corresponding layer. This feature is beneficial for debugging and reusing trained neural networks, where the user needs to reach some configuration parameters.

The Inference tab allows the user to run simple inferences on the given model. For example, the user can upload an image and predict its label directly from the GUI without deploying the model manually. 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, Fig. 9 illustrates an 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)


Dashboard / Project: CNN_MNIST / Experiment: 2019-11-08 16:28:15 ()
Download
Share
Generate Experiment
Metadata
Architecture
Test
Share
File input
Choose File
No file chosen
Select the file which you want to test.
Submit

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.

Fig. 9 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

of this inference function was generated entirely by ModelKB. Moreover, the inference task can assist users in model selection 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.

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 new cloud service and integrating it with an existing system requires significant amounts of time and effort. For example, our experiments illustrated that given a well-trained model, a graduate student with no prior knowledge of 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

```
#import statements
import sys
{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 4 A code snippet illustrating a simple template for generating inference functions for computer vision models

main features of ModelKB is to facilitate automatic deployment of trained models locally and semi-automatic deployment on the cloud using "one-click" directly from the GUI.

First, ModelKB uses a templating engine to generate inference functions automatically and 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 capable of parsing the collected metadata and subsequently generating 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. Listing 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 5 illustrates a fully functional inference function that was automatically generated by ModelKB. Our deployment templates support tabular, textual, and image data types.

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 Fig. 10). Notice that we 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 service.

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 set up the necessary containers and protocols. However, users are responsible for double-checking the generated inference functions' correctness, 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:

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

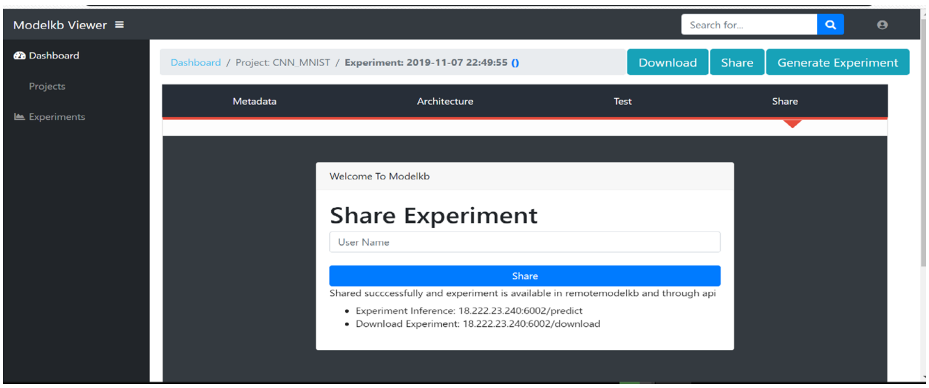
- *Remote user with ModelKB:* If the receiving user already has ModelKB installed on their system, they can use the download feature to clone a remote model to their system locally by providing the model's global ID. This will download and unpack a Docker container and create a new database entry to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

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 imported into a proper IDE for further reusability. In contrast, the user can download a shared model manually, as explained above. Figure 10 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 sharing 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, its metadata, artifacts into a reusable docker image that can be shared through ModelKB Server, 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 and 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



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

Fig. 10 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

(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

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.

tool, such as ModelKB. In Table 3, we list the NeurIPS reproducibility checklist and show what information is collected by ModelKB automatically. This validates the usefulness and correctness of ModelKB in enhancing model reproducibility.

4 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 integrating 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 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`.

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

def fit(self, x, y, validation_data=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 6 A snippet of the fit function implementation for PyTorch

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 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 (Roeder 2019), 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.

The wide range of software methods and tools used to enable this project illustrates the importance of interdisciplinary research. In particular, this project demonstrates that some innovative solutions from the machine learning domain heavily rely on technical solutions, methods, and tools from the software engineering community.

5 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 H&R 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 from Sprint. Additionally, the user study included seven Ph.D. researchers from the Computer Science Department at UMKC, who are 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, Alzheimer's disease prediction (Velazquez et al. 2019), building autoencoders for textual data (Goudarzvand et al. 2020), and image segmentation for brain MRI scans (Albishri et al. 2019). 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).

5.1 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 can extract and track 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/).

5.2 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 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:

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.

- Step 1: Clone ModelKB from our private repository in GitHub and set it up locally.
- Step 2: Use ModelKB to track experiments and then visualize 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 the model's architecture.
- Step 5: Use ModelKB to run inference tasks using the 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 the model remotely using ModelKB's GUI and use the automatically generated inference API to run inferences from a different application.
- Step 7: Delete the deep learning project from ModelKB and download it from the remote Server. Alternatively, download the 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 (Fig. 11).

5.3 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

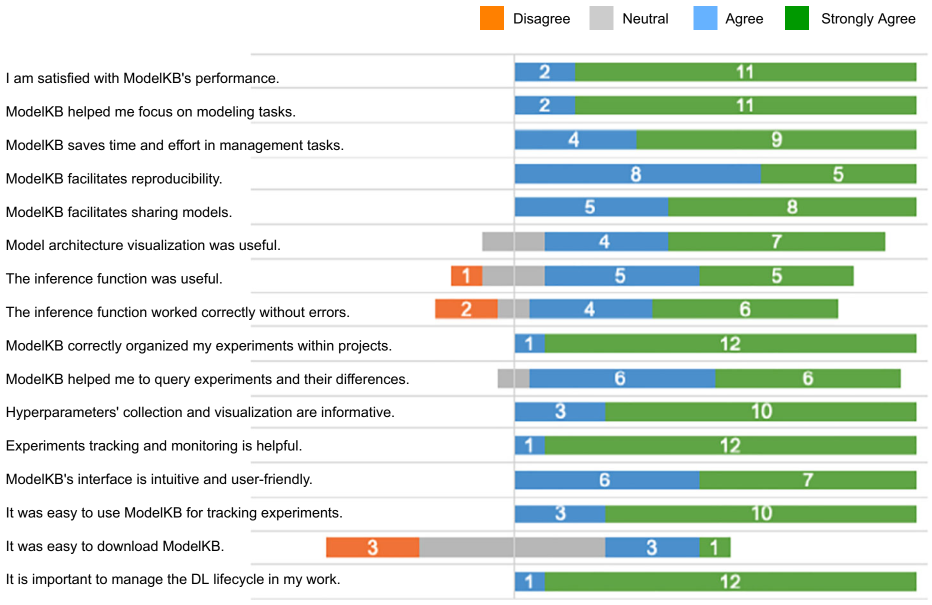


Fig. 11 Likert-scale answers to the survey. Total number of participates: 13

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 (Vartak 2018a).

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 to install 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. 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 are not downloaded automatically by our system, 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 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 to 6 seconds more 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 for hours. 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 participants *agree* 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.

5.4 Threats to Validity

A primary risk to our evaluation is the small number of study participants. However, we argue that our study participants represent a wide range of deep learning 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 multilayer 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 using 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 deployment servers. Then, ModelKB will be responsible for automatically collecting all required configurations, including dependencies, model parameters, the automatically generated prediction function, and then creating the docker image to be deployed on the user's selected service.

6 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 (Microsoft 2017), Digits (Nvidia 2019), and Deep Cognition

(DeepCognition 2019). 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. (Vartak 2018a) 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 (Abadi et al. 2016), Caffee (Jia et al. 2014), and PyTorch (Paszke et al. 2017). 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 (Facebook 2019), TensorBoard for TensorFlow (Google 2019), Digits by Nvidia (2019), Michelangelo by Uber (2019), Amazon's SageMaker (SageMaker 2018) a fully-loaded machine learning training and deployment system. Additionally, other frame-work independent tools include CometML (https://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 12 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 (Zaharia et al. 2018), 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 (Miao et al. 2017a) was one of the first systems that aimed at addressing model management issues in machine learning. It 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 (Miao et al. 2017b) 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. (2018b) 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 (VanRijn et al. 2013; Hines et al. 2004) and Google

SeebBank (Seedbank 2019). 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.

```
model = create_model()
model.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)
log_dir = "/logs"+datetime.datetime.now().strftime("..")
tensorboard_callback = tf.keras.callback.TensorBoard(
    log_dir,
    histogram_freq=1
)

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

a

```
my_exp = modelkb.Experiment('MNIST-LeNet','user1')

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

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

b

Fig. 12 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

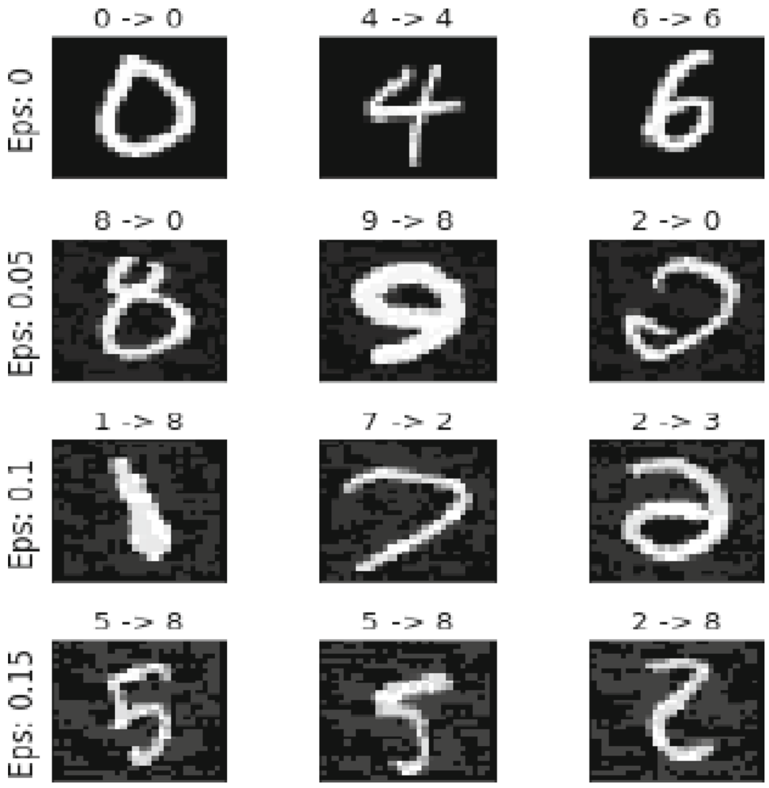


Fig. 13 Illustrating the accuracy of a classification model with different values of epsilon in an FGSM attack (Eps = 0 means no changes were made to the input). While the added noise is barely visible to a human eye, the model fails to classify the noisy images. (ground truth ->prediction result)

7 Conclusions

In this paper, we presented and discussed our approach and software system for automating the modeling lifecycle in deep learning. Specifically, 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. Our user study evaluation validated the feasibility and efficiency of ModelKB in automatically monitoring the modeling experiments, deploy selected models, and share and publish models. Moreover, the case study validated that ModelKB can significantly facilitate model reproducibility. A demo of ModelKB can be accessed at https://info.umkc.edu/UDIC_Research/index.php/modelkb/

8 Future Work

Different phases of the deep learning lifecycle are vulnerable to adversary attacks, including the training and inference phases. Deep learning models that are deployed in vital systems make great incentives for malicious adversaries. For example, adversaries can poison public datasets with wrongly-labeled data, which can ease future attacks on models trained using such datasets (Goodfellow et al. 2018). Another popular set of attacks includes adversarial attacks that aim at introducing small perturbations in the input data to fool a deployed model, causing it to produce wrong predictions (Goodfellow et al. 2014). Therefore, we aim to focus, in our future work, on automating the assessment of model vulnerability during the training phase, with minimal user intervention. We strongly believe that our customized Callbacks developed in this work can facilitate the development of automated vulnerability tests without additional training cycles.

Our initials trials illustrate that Callbacks can be beneficial to integrate training cycles with vulnerability tests to assess the privacy and security of deep learning models under training. For example, we developed a Callback to assess the model vulnerability towards the Fast Gradient Sign Method (FGSM) (Goodfellow et al. 2014), which requires access to the model and its parameters. We evaluated the vulnerability of the model while training a LeNet architecture using the MNIST-Digits dataset. Figure 13 illustrates the accuracy of the model with several values of epsilon (i.e., the amount of perturbation) added to the input images and their classification results given the shape: ground truth ->prediction result. We notice that the accuracy drastically drops from about 98% without attacking the model to less than 10% when attacking the model with an epsilon value of 0.15.

Acknowledgements We would like to thank Sirisha Rella and Duy Ho for their help in some implementation parts in early versions of ModelKB. We would like to thank the Ph.D. students and the industry participants who helped in conducting the user study and evaluate the software system. We also thank the anonymous reviewers for their time and effort in reviewing this work. The first author thanks Yasmin Hussein for her help and support throughout this work. The coauthor, Yugyung Lee, would like to acknowledge the partial support of the NSF Grant No. 1747751

References

Abadi M, Agarwal A, Barham P, Brevdo E, Chen Z, Citro C, Corrado GS, Davis A, Dean J, Devin M, Ghemawat S, Goodfellow I, Harp A, Irving G, Isard M, Jia Y, Jozefowicz R, Kaiser L, Kudlur M, Levenberg J, Mané D, Monga R, Moore S, Murray D, Olah C, Schuster M, Shlens J, Steiner B, Sutskever I, Talwar K, Tucker P, Vanhoucke V, Vasudevan V, Viégas F, Vinyals O, Warden P, Wattenberg M, Wicke M, Yu Y, Zheng X (2015) TensorFlow: Large-scale machine learning on heterogeneous systems. https://www.tensorflow.org/, Software available from tensorflow.org

Abadi M, Barham P, Chen J, Chen Z, Davis A, Dean J, Devin M, Ghemawat S, Irving G, Isard M et al (2016) Tensorflow: a system for large-scale machine learning. In: OSDI, vol 16, pp 265–283

Albishri AA, Shah SJH, Schmiedler A, Kang SS, Lee Y (2019) Automated human claustrum segmentation using deep learning technologies. arXiv:1911.07515

Bergstra J, Breuleux O, Bastien F, Lamblin P, Pascanu R, Desjardins G, Turian J, Warde-Farley D, Bengio Y (2010) Theano: A cpu and gpu math compiler in python. In: Proc. 9th Python in Science Conf, vol 1, pp 3–10

Castelvecchi D (2016) Can we open the black box of ai? Nat 538(7623):20

Chen X, Duan Y, Houthooft R, Schulman J, Sutskever I, Abbeel P (2016) Infogan: Interpretable representation learning by information maximizing generative adversarial nets. In: Advances in neural information processing systems, pp 2172–2180

Chollet F et al (2015) Keras. https://keras.io

DeepCognition (2019) One stop for deep learning developers. https://deepcognition.ai/

Documentation P (2019) Abstract syntax trees. https://docs.python.org/3/library/ast.html

Facebook (2019) Introducing fblearner flow: Facebook's ai backbone. https://conferences.oreilly.com/strata/big-data-conference-ny-2015/public/schedule/detail/42988

Garcia R, Sreekanti V, Yadwadkar N, Crankshaw D, Gonzalez JE, Hellerstein JM (2018) Context: The missing piece in the machine learning lifecycle. In: KDD CMI Workshop, vol 114

Gharibi G, Walunj V, Alanazi R, Rella S, Lee Y (2019a) Automated management of deep learning experiments. In: Proceedings of the 3rd International Workshop on Data Management for End-to-End Machine Learning, pp 1–4

Gharibi G, Walunj V, Rella S, Lee Y (2019b) 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, pp 28–34

Ghezzi C, Jazayeri M, Mandrioli D (2002) Fundamentals of software engineering. Prentice Hall PTR

Goodfellow IJ, Shlens J, Szegedy C (2014) Explaining and harnessing adversarial examples. arXiv:1412.6572

Goodfellow I, Bengio Y, Courville A (2016) Deep learning. MIT Press. http://www.deeplearningbook.org

Goodfellow I, McDaniel P, Papernot N (2018) Making machine learning robust against adversarial inputs. Commun ACM 61(7)

Google (2019) Tensorboard: Visualizing learning. https://www.tensorflow.org/guide/summaries_and_tensorbard

Goudarzvand S, Gharibi G, Lee Y (2020) Scat: Second chance autoencoder for textual data. arXiv:2005.06632

Grinberg M (2018) Flask web development: developing web applications with python. O'Reilly Media, Inc.

Hall MA (1999) Correlation-based feature selection for machine learning

Hannun A, Case C, Casper J, Catanzaro B, Diamos G, Elsen E, Prenger R, Satheesh S, Sengupta S, Coates A et al (2014) Deep speech: Scaling up end-to-end speech recognition. arXiv:1412.5567

He K, Zhang X, Ren S, Sun J (2016) Deep residual learning for image recognition. In: Proceedings of the IEEE conference on computer vision and pattern recognition, pp 770–778

Hellerstein JM, Sreekanti V, Gonzalez JE, Dalton J, Dey A, Nag S, Ramachandran K, Arora S, Bhattacharyya A, Das S et al (2017) Ground: A data context service. In: CIDR

Hines ML, Morse T, Migliore M, Carnevale NT, Shepherd GM (2004) Modeldb: a database to support computational neuroscience. J Comput Neurosci 17(1):7–11

Jia Y, Shelhamer E, Donahue J, Karayev S, Long J, Girshick R, Guadarrama S, Darrell T (2014) Caffe: Convolutional architecture for fast feature embedding. In: Proceedings of the 22nd ACM international conference on Multimedia. ACM, pp 675–678

Jinja (2019) Python template language. https://jinja.palletsprojects.com/en/2.11.x/

Karpathy A, Toderici G, Shetty S, Leung T, Sukthankar R, Fei-Fei L (2014) Large-scale video classification with convolutional neural networks. In: Proceedings of the IEEE conference on Computer Vision and Pattern Recognition, pp 1725–1732

Krizhevsky A, Sutskever I, Hinton GE (2012) Imagenet classification with deep convolutional neural networks. In: Advances in neural information processing systems, pp 1097–1105

Kumar A, McCann R, Naughton J, Patel JM (2016) Model selection management systems: The next frontier of advanced analytics. ACM SIGMOD Record 44(4):17–22

Kumar A, Boehm M, Yang J (2017) Data management in machine learning: Challenges, techniques, and systems. In: Proceedings of the 2017 ACM International Conference on Management of Data. ACM, pp 1717–1722

Lawrence S, Giles CL, Tsoi AC, Back AD (1997) Face recognition: A convolutional neural-network approach. IEEE Trans Neural Netw 8(1):98–113

LeCun Y, Boser BE, Denker JS, Henderson D, Howard RE, Hubbard WE, Jackel LD (1990) Handwritten digit recognition with a back-propagation network. In: Advances in neural information processing systems, pp 396–404

LeCun Y, Bottou L, Bengio Y, Haffner P et al (1998) Gradient-based learning applied to document recognition. Proc IEEE 86(11):2278–2324

LeCun Y, Bengio Y, Hinton G (2015) Deep learning. Nature 521(7553):436

Miao H, Li A, Davis LS, Deshpande A (2017a) Modelhub: Deep learning lifecycle management. In: 2017 IEEE 33rd International Conference on Data Engineering (ICDE). IEEE, pp 1393–1394

Miao H, Li A, Davis LS, Deshpande A (2017b) Towards unified data and lifecycle management for deep learning. In: 2017 IEEE 33rd International Conference on Data Engineering (ICDE). IEEE, pp 571–582

Miao H, Deshpande A (2018) Provdb: Provenance-enabled lifecycle management of collaborative data analysis workflows. IEEE Data Eng Bull 41(4):26–38

Microsoft (2017) Machine learning studio. https://azure.microsoft.com/en-us/services/machine-learning-studio/

ModelHubAI (2019) 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. http://modelhub.ai/

ModelZoo (2019) A set of pretrained models models hosted on github. https://github.com/BVLC/caffe/wiki/Model-Zoo

Montavon G, Samek W, Müller K-R (2018) Methods for interpreting and understanding deep neural networks. Digital Signal Process 73:1–15

Nvidia (2019) Digits: A graphical web interface for nvcaffe and tensorflow. https://docs.nvidia.com/deeplearning/digits/digits-user-guide/index.html

Paszke A, Gross S, Chintala S, Chanan G, Yang E, DeVito Z, Lin Z, Desmaison A, Antiga L, Lerer A (2017) Automatic differentiation in PyTorch. In: NIPS Autodiff Workshop

PyTorch (2019) A set of pretrained pytorch models. https://pytorch.org/docs/stable/torchvision/models.html

Roeder L (2019) Netron: Visualizing deep learning models. https://github.com/lutzroeder/netron

Russakovsky O, Deng J, Su H, Krause J, Satheesh S, Ma S, Huang Z, Karpathy A, Khosla A, Bernstein M, Berg AC, Fei-Fei L (2015) ImageNet Large Scale Visual Recognition Challenge. Int J Comput Vis (IJCV) 115(3):211–252. https://doi.org/10.1007/s11263-015-0816-y

SageMaker (2018) Sagemaker. https://aws.amazon.com/sagemaker//

Schelter S, Böse J-H, Kirschnick J, Klein T, Seufert S (2017) Automatically tracking metadata and provenance of machine learning experiments. In: Machine Learning Systems Workshop at NIPS

Schelter S, Biessmann F, Januschowski T, Salinas D, Seufert S, Szarvas G, Vartak M, Madden S, Miao H, Deshpande A et al (2018a) On challenges in machine learning model management. IEEE Data Eng Bull 41(4):5–15

Schelter S, Böse J-H, Kirschnick J, Klein T, Seufert S (2018b) Declarative metadata management: A missing piece in end-to-end machine learning

Sculley D, Holt G, Golovin D, Davydov E, Phillips T, Ebner D, Chaudhary V, Young M, Crespo J-F, Dennison D (2015) Hidden technical debt in machine learning systems. In: Advances in neural information processing systems, pp 2503–2511

Seedbank G (2019) A set of models shared via google colab. https://research.google.com/seedbank/

Seide F, Agarwal A (2016) Cntk: Microsoft's open-source deep-learning toolkit. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM, pp 2135–2135

Silver D, Huang A, Maddison CJ, Guez A, Sifre L, Van DenDriessche G, Schrittwieser J, Antonoglou I, Panneershelvam V, Lanctot M et al (2016) Mastering the game of go with deep neural networks and tree search. Nature 529(7587):484

Simonyan K, Zisserman A (2014) Very deep convolutional networks for large-scale image recognition. arXiv:1409.1556

SQL (2019) A c-language library to run sql engine. https://www.sqlite.org/index.html

Szegedy C, Ioffe S, Vanhoucke V, Alemi AA (2017) Inception-v4, inception-resnet and the impact of residual connections on learning. In: AAAI, vol 4, pp 12

Tantithamthavorn C, Hassan AE, Matsumoto K (2018) The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. IEEE Trans Softw Eng

Uber (2019) Imeet michelangelo: Uber's machine learning platform. https://eng.uber.com/michelangelo/

VanRijn JN, Bischl B, Torgo L, Gao B, Umaashankar V, Fischer S, Winter P, Wiswedel B, Berthold MR, Vanschoren J (2013) Openml: A collaborative science platform. In: Joint European Conference on Machine Learning and Knowledge Discovery in Databases. Springer, pp 645–649

Vartak M, Subramanyam H, Lee W-E, Viswanathan S, Husnoo S, Madden S, Zaharia M (2016) M odel db: a system for machine learning model management. In: Proceedings of the Workshop on Human-In-the-Loop Data Analytics. ACM, pp 14

Vartak M (2018a) Infrastructure for model management and model diagnosis. Ph.D. Thesis, Massachusetts Institute of Technology

Vartak M, Madden S (2018b) Modeldb: Opportunities and challenges in managing machine learning models. IEEE Data Eng Bull 41(4):16–25

Velazquez M, Anantharaman R, Velazquez S, Lee Y (2019) Rnn-based alzheimer's disease prediction from prodromal stage using diffusion tensor imaging. In: 2019 IEEE International Conference on Bioinformatics and Biomedicine (BIBM). IEEE, pp 1665–1672

Yu X, Sohn K, Chandraker M (2018) Video security system using a siamese reconstruction convolutional neural network for pose-invariant face recognition. US Patent App. 15/803,318

Zaharia M, Chen A, Davidson A, Ghodsi A, Hong SA, Konwinski A, Murching S, Nykodym T, Ogilvie P, Parkhe M et al (2018) Accelerating the machine learning lifecycle with mlflow. Data Engineering:39

Zhang A, Lipton ZC, Li M, Smola AssJ (2019) Dive into deep learning. http://www.d2l.ai

Affiliations

Gharib Gharibi[1] · Vijay Walunj[1] · Raju Nekadi[1] · Raj Marri[1] · Yugyung Lee[1]

Vijay Walunj
vijay.walunj@mail.umkc.edu

Raju Nekadi
rn8mh@mail.umkc.edu

Raj Marri
rmwwc@mail.umkc.edu

Yugyung Lee
LeeYu@umkc.edu

1 School of Computing and Engineering, University of Missouri-Kansas City, 5000 Holmes St, Kansas City, MO 64110, USA