Rafiki: Machine Learning as an Analytics Service System

Wei Wang[†], Jinyang Gao[†], Meihui Zhang[*], Sheng Wang[†]
Gang Chen[‡], Teck Khim Ng[†], Beng Chin Ooi[†], Jie Shao[◇], Moaz Reyad[†]

[†]National University of Singapore, [*]Beijing Institute of Technology
[‡]Zhejiang University, [◇] University of Electronic Science and Technology of China
[†]{wangwei, jinyang.gao, wangsh, ngtk, ooibc, moaz}@comp.nus.edu.sg
[*]meihui_zhang@bit.edu.cn, [‡] cg@zju.edu.cn, [◇]shaojie@uestc.edu.cn

ABSTRACT

Big data analytics is gaining massive momentum in the last few years. Applying machine learning models to big data has become an implicit requirement or an expectation for most analysis tasks, especially on high-stakes applications. Typical applications include sentiment analysis against reviews for analyzing on-line products, image classification in food logging applications for monitoring user's daily intake, and stock movement prediction. Extending traditional database systems to support the above analysis is intriguing but challenging. First, it is almost impossible to implement all machine learning models in the database engines. Second, expert knowledge is required to optimize the training and inference procedures in terms of efficiency and effectiveness, which imposes heavy burden on the system users. In this paper, we develop and present a system, called Rafiki, to provide the training and inference service of machine learning models. Rafiki provides distributed hyper-parameter tuning for the training service, and online ensemble modeling for the inference service which trades off between latency and accuracy. Experimental results confirm the efficiency, effectiveness, scalability and usability of Rafiki.

PVLDB Reference Format:
Wei Wang, Jinyang Gao, Meihui Zhang, Sheng Wang, Gang Chen, Teck Khim Ng, Beng Chin Ooi, Jie Shao, Moaz Reyad. Rafiki: Machine Learning as an Analytics Service System. *PVLDB*, 12(2): 128-140, 2018.
DOI: https://doi.org/10.14778/3282495.3282499

1. INTRODUCTION

Data analysis plays an important role in extracting valuable insights from a huge amount of data. Database systems have been traditionally used for storing and analyzing structured data, spatial-temporal data, graph data, etc. Other data, such as multimedia data (e.g., images and free text), and domain specific data (e.g, medical data and sensor data), are being generated at fast speed and constitute a significant portion of the Big Data [31]. It is beneficial to analyze these data for database applications [32]. For instance, inferring the quality of a product from the review column in the sales database would help to explain the sales numbers; Analyzing food images from the food logging application can extract the food preference of people from different ages. However, the above analysis requires machine learning models, especially deep learning [15] models, for sentiment analysis [3] to classify the review as positive or negative, and for image classification [13] to recognize the food type from images. Figure 1 shows a pipeline of data analysis, where database systems have been widely used for the first 3 stages and machine learning is good at the 4th stage.

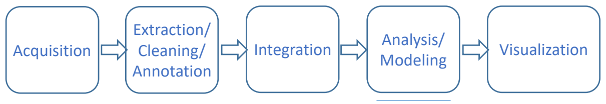


Figure 1: Data analytic pipeline.

One approach to integrating the machine learning techniques into database applications is to preprocess the data off-line and add the prediction results into a new column, e.g., a column for the food type. However, such off-line preprocessing has two-folds of restriction. First, it requires the developers to have expert knowledge and experience of training machine learning models. Second, the queries cannot involve attributes of the object, e.g., the ingredients of food, if they are not predicted in the preprocessing step. In addition, predicting the labels of all rows wastes a lot of time if the queries only read the food type column of few rows, e.g., due to a filtering on other columns. Another approach is to carry out the prediction on-line as user-defined functions (UDFs) in the SQL query. However, it is challenging to implement all machine learning models in UDFs by database users [23], for machine learning models vary a lot in terms of theory and implementation. It is also difficult to optimize the prediction accuracy in the database engine.

A better solution is to call the corresponding cloud machine learning service, e.g., APIs, in the UDFs for each prediction (or analysis) task. Cloud service is economical, elastic and easy to use. With the resurgence of AI, cloud providers like Amazon (AWS), Google and Microsoft (Azure) have built machine learning services on their cloud platforms. There are two types of cloud services. The first one is to provide an API for each specific task, e.g., image classification and sentiment analysis. Such APIs are available on Amazon AWS[1] and Google cloud platform[2]. The disadvantage is that the accuracy could be low since the models are trained by Amazon and Google with general data, e.g., food from all countries,

[*]Meihui Zhang is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 12, No. 2
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3282495.3282499

[1]https://aws.amazon.com/machine-learning/
[2]https://cloud.google.com/products/machine-learning/

which is likely to be different from the user's data, e.g., food from Singapore. The second type of service overcomes this issue by providing the training service, where users can upload their own datasets to conduct training. This service is available on Amazon AWS and Microsoft Azure. However, only a limited number of machine learning models are supported [33]. For example, only simple logistic regression or linear regression models[3] are supported by Amazon. Deep learning models such as convolutional neural networks (ConvNets) and recurrent neural networks (RNN) are not included.

As a machine learning service, it not only needs to cover a wide range of models, including deep learning models, but also provide an easy-to-use, efficient and effective service for users without much machine learning knowledge and experience, e.g., database users. Considering that different models may result in different performance in terms of efficiency (e.g., latency) and effectiveness (e.g., accuracy), the cloud service has to select the proper models for a given task. Moreover, most machine learning models and the training algorithms come with a set of hyper-parameters or knobs, e.g., number of layers and size of each layer in a neural network. Some hyper-parameters are vital for the convergence of the training algorithm and the final model performance, e.g., the learning rate of the stochastic gradient descent (SGD) algorithm. Manual hyper-parameter tuning requires rich experience and is tedious. Random search and Bayesian optimization are two popular automatic tuning approaches. However, both are costly as they have to test many hyper-parameter assignments and each trial (assignment) takes hundreds of epochs[4]. A distributed tuning platform is desirable. Besides training, inference also matters as it directly affects the user experience. Machine learning products often apply ensemble modeling to improve the prediction performance. However, ensemble modeling incurs a larger latency (i.e., response time) compared with using a single prediction model. Therefore, there is a trade-off between accuracy and latency.

There have been studies on these challenges for providing machine learning as a service. mlbench [33] compares the cloud service of Amazon and Azure over a set of binary classification tasks. Ease.ml [16] builds a training platform with model selection aiming at optimizing the resource utilization for multiple tenants. Google Vizier [8] is a distributed hyper-parameter tuning platform that provides tuning service for other systems. Clipper [4] focuses on the inference by proposing a general framework and specific optimization for efficiency and accuracy. However, none of them resolve all the challenges.

In this paper, we present a system, called Rafiki, to provide both the training and inference services for machine learning models. With Rafiki, (database) users are exempted from constructing the machine learning models, tuning the hyper-parameters, optimizing the prediction accuracy and speed. Instead, they simply upload their datasets and configure the service to conduct training and then deploy the model for inference. As a cloud service system [9, 5], Rafiki manages the hardware resources, failure recovery, etc. It comes with a set of built-in machine learning models for popular tasks such as image and text processing. In particular, we make the following contributions to make Rafiki easy-to-use, efficient and effective.

- We propose a unified system architecture for both the training and the inference services. We observe that the two services share some common components such as the parameter server for model parameter storage, and the distributed computing environment for distributed hyper-parameter tuning and parallel inference. By sharing the same underlying storage, communication protocols and computation resource, we implicitly avoid some technical debts [25]. Moreover, by combining the two services together, Rafiki enables instant model deployment after training.
- For the training service, we first propose a general framework for distributed hyper-parameter tuning, which is extensible for popular hyper-parameter tuning algorithms including random search and Bayesian optimization. In addition, we propose a collaborative tuning scheme specifically for deep learning models, which uses the model parameters from the current top performing training trials to initialize new trials.
- For the inference service, we propose a scheduling algorithm based on reinforcement learning to optimize the overall accuracy and reduce latency. Both algorithms are adaptive to the changes of the request arriving rate.
- We conduct micro-benchmark experiments to evaluate the performance of our proposed algorithms.

In the reminder of this paper, we review related works in Section 2 and then give an overview of Rafiki in Section 3. Section 4 describes the training service and the distributed hyper-parameter tuning algorithm. The inference service and optimization techniques are introduced in Section 5. Section 6 describes the system implementation. We explain the experimental evaluation in Section 7. A case study is introduced in Section 8. Section 10 concludes the paper.

2. RELATED WORK

Rafiki is a SaaS that provides training and inference services of machine learning models. In this section, we review related works on SaaS, hyper-parameter tuning for model training, inference optimization, and reinforcement learning which is adopted by Rafiki for inference optimization.

2.1 Cloud Service

Cloud computing has changed the way of IT operation in many companies by providing infrastructure as a service (IaaS), platform as a service (PaaS) and software as a service (SaaS). With IaaS, users and companies can use remote physical or virtual machines instead of establishing their own data center. Based on user requirements, IaaS can scale up the compute resources quickly. PaaS, e.g., Microsoft Azure, provides development toolkit and running environment, which can be adjusted automatically based on business demand. SaaS, including database as a service[9, 5], installs software on cloud computing platforms and provides application services, which simplifies software maintenance. The 'pay as you go' pricing model is convenient and economic for (small) companies and research labs.

Recently, machine learning (especially deep learning) has gain a lot of interest due to its outstanding performance in analytics and predictive tasks. However, it is not easy to build a machine learning application [7] since there are multiple non-trivial steps involved. In this paper, we focus on two primary steps, namely training and inference. Cloud providers, like Amazon AWS, Microsoft Azure and Google Cloud, have already included some services for the two steps. However, their training services have limited support

[3]https://docs.aws.amazon.com/machine-learning/latest/dg/learning-algorithm.html

[4]Training the model by scanning the dataset once is called one epoch.

for deep learning models, and their inference services cannot be customized for customers' data (see Section 1). There are also research papers towards efficient resource allocation for training [16] and efficient inference [4] on the cloud. Rafiki differs from the existing cloud services on both the service types and the optimization techniques. Firstly, Rafiki allows users to train machine learning (including deep learning) models on their own data, and then deploy them for inference. Secondly, Rafiki has special optimization for the training (Section 4) and inference (Section 5) service.

2.2 Hyper-parameter Tuning

To train a machine learning model for an application, we need to decide many hyper-parameters related to the model structure, optimization algorithm and data preprocessing operations. All hyper-parameter tuning algorithms work via empirical trials. Random search [2] randomly selects the value of each hyper-parameter and then tries it. It has shown to be more efficient than grid search that enumerates all possible hyper-parameter combinations. Bayesian optimization [27] assumes the optimization function (hyper-parameters as the input and the inference performance as the output) follows Gaussian process. It samples the next point in the hyper-parameter space based on existing trials (according to an acquisition function). Google Vizier [8] provides the hyper-parameter tuning service on the cloud. Rafiki provides distributed hyper-parameter tuning service, which is compatible with all the above mentioned hyper-parameter tuning algorithms. In addition, it supports collaborative tuning that shares model parameters across trials. Our experiments confirm the effectiveness of the collaborative tuning scheme.

2.3 Inference Optimization

To improve the inference accuracy, ensemble modeling that trains and deploys multiple models is widely applied. For real-time inference, latency is another critical metric of the inference service. NoScope [12] proposes specific inference optimization for video querying. Clipper [4] studies multiple optimization techniques to improve the throughput (via batch size) and reduce latency (via caching). It also proposes to do model selection for ensemble modeling using multi-armed bandit algorithms. Compared with Clipper, Rafiki provides both training and inference services, whereas Clipper focuses only on the inference service. In addition, Clipper optimizes the throughput, latency and accuracy separately, whereas Rafiki models them together to find the optimal model selection and batch size. TFX [20] provides the training and inference service through a set of libraries including Tensorflow transform[5] for data preprocessing, TensorFlow serving [1] for inference, etc..

2.4 Reinforcement Learning

In reinforcement learning (RL)[17], each time an agent takes one action, it enters the next state. The environment returns a reward based on the action and the states. The learning objective is to maximize the aggregated rewards. RL algorithms decide the action to take based on the current state and experience gained from previous trials (exploration). Policy gradient based RL algorithms maximize the expected reward by taking actions following a policy function $\pi_\theta(a_t|s_t)$ over n steps (Equation 1), where ς represents a trajectory of n (action a_t, state s_t, reward R_t) tuples, and γ is a decaying factor. The expectation is taken over all possible trajectories. π_θ is usually implemented using a multi-layer perceptron model (θ represents the parameters) that takes the state vector as input and generates the action (a scalar value for continuous action or a Softmax output for discrete actions). Equation 1 is optimized using stochastic gradient ascent methods. Equation 3 is used as a 'surrogate' function for the objective as its gradient is easy to compute via automatic differentiation.

[5]https://www.tensorflow.org/tfx/

$$J(\theta) = E_{\varsigma\sim\pi_\theta(\varsigma)}[\sum_{t=0}^{n}\gamma^t R_t], \quad (1)$$

$$\nabla_\theta J(\theta) = E_{\varsigma\sim\pi_\theta(\varsigma)}[\sum_{t=0}^{n}\nabla_\theta \log\pi_\theta(a_t|s_t)\sum_{t=0}^{T}\gamma^t R_t] \quad (2)$$

$$\hat{J}(\theta) = E_{\varsigma\sim\pi_\theta(\varsigma)}[\sum_{t=0}^{T}\log\pi_\theta(a_t|s_t)\sum_{t=0}^{n}\gamma^t R_t] \quad (3)$$

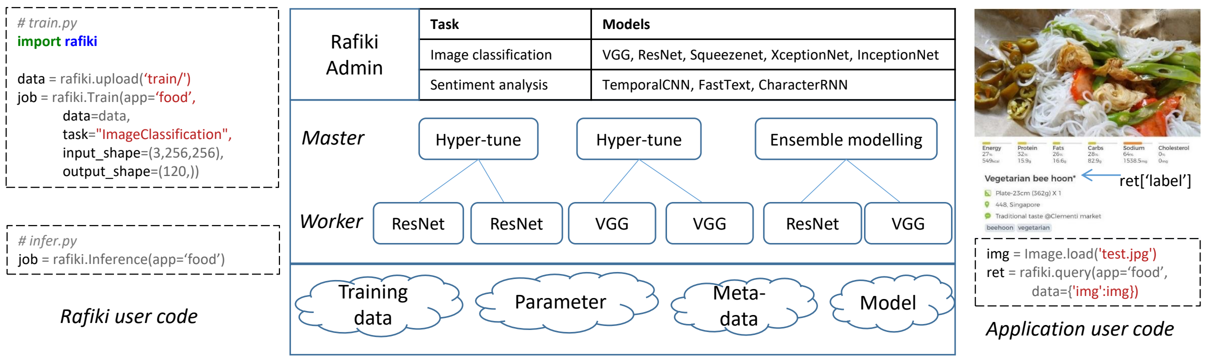
Many approaches have been proposed to improve the policy gradient by reducing the variance of the rewards, including actor-critic model [19] which subtracts the reward R_t by a baseline $V(s_t)$. $V(s_t)$ is an estimation of the reward, which is also approximated using a neural network like the policy function. Then R_t in the above equations becomes $R_t - V(s_t)$. Yuxi [17] has done a survey of deep reinforcement learning, including the actor-critic model used in this paper.

3. SYSTEM OVERVIEW

To develop an application, Rafiki users configure the training or inference jobs via either RESTFul APIs or Python SDK. In Figure 2 (left column), we show one image classification example using the Python SDK. The training code firstly loads the training data from a local folder into Rafiki's distributed data storage (HDFS) and returns the data path. Next, it creates the training job by passing the unique application name, the data path, and the input/output shapes (i.e., a tuple or a list of tuples). The input-output shapes are used for model customization. For example, ConvNets usually accept input images of a fixed shape, and adapt the final output layer to have the same number of neurons as the size of the output shape. The job ID is returned for monitoring the execution status and metrics. Once the job is submitted, Rafiki selects the models (Section 4.1), launches the master and workers (Section 6), downloads the training data onto each worker's local disk and then starts hyper-parameter tuning (Section 4.2). During training, the best version of parameters are stored in the parameter server. After training, we can deploy the models instantly as shown in *infer.py*. The application name is used to retrieve the models and parameters from the training stage. Once the model is deployed, application users can submit their requests for prediction as shown by the code in *query.py*. We can see that there are 3 types of Rafiki users:

- Model contributors (e.g., Rafiki developers), who are machine learning experts and are responsible for adding new models into Rafiki.
- Application developers without much machine learning background. They provide the training data following the task requirements designed by model contributors, and call Rafiki's APIs (left column of Figure 2) to train models and then deploy the models for inference.
- Application users (right column of Figure 2) who send requests to Rafiki's inference services.

In Rafiki, we focus on distributed hyper-parameter tuning and inference optimization as shown in the middle of Figure 2. In the figure, Rafiki tunes two models, i.e., ResNet [10] and VGG [26] separately using 2 workers for each model. For the inference job, two models (running on different workers) are combined via ensemble modeling for better prediction accuracy.



```
# train.py
import rafiki

data = rafiki.upload('train/')
job = rafiki.Train(app='food',
        data=data,
        task="ImageClassification",
        input_shape=(3,256,256),
        output_shape=(120,))
```

```
# infer.py
job = rafiki.Inference(app='food')
```

Task	Models
Image classification	VGG, ResNet, Squeezenet, XceptionNet, InceptionNet
Sentiment analysis	TemporalCNN, FastText, CharacterRNN

```
img = Image.load('test.jpg')
ret = rafiki.query(app='food',
        data={'img':img})
```

Figure 2: Overview of Rafiki.

Rafiki is compatible with any machine learning framework, including TensorFlow [1], Scikit-learn, Kears, etc., although we have tested it on top of Apache Singa [21] due to its efficiency and scalable architecture. More details of Rafiki implementation shall be described in Section 6.

4. TRAINING SERVICE

The training service consists of two steps, namely model selection and hyper-parameter tuning. We adopt a simple model selection strategy in Rafiki. For hyper-parameter tuning, we first propose a general programming model for distributed tuning, and then propose a collaborative tuning scheme.

4.1 Model Selection

Every built-in model in Rafiki is registered with a task category, e.g., image classification or sentiment analysis via the *register* API shown below.

```
class Model:
  def train(app, dpath, inshape, outshape)
  def predict(x)
  def load(app)
  def save(app)

def register(model, task)
```

New models must implement the four functions for training, prediction, model saving and restoring respectively. The argument *app* is a unique identifier of the application, which is used to identify the data and parameters in the distributed storage (see Section 6). The models should also follow the requirement of the corresponding task, e.g, the inference code of image classification should expect input data to be a tensor with 4 dimensions for the batch, channel, height and weight respectively.

For each training job, Rafiki selects the corresponding built-in models based on the task type. The table in Figure 2 lists some built-in tasks and the models. With the proliferation of machine learning, we are able to find open source implementations for almost every popular model.

By default, Rafiki uses the models suggested by the model contributors for each task. The principle is to use models with similar performance on the same benchmark dataset but with different architectures. This is to create a competitive and diverse model set [14] for better ensemble accuracy. Other model selection strategies are also available, e.g., selecting all models, selecting top-K fast or accurate models. We observe that the models for the same task perform consistently across datasets. For example, ResNet[10] is better than AlexNet [13] and SqueezeNet [11] for a bunch of datasets including ImageNet [6], CIFAR-10[6], etc.. Therefore, the performance on one benchmark dataset is a reliable reference for model selection for new datasets.

Table 1: Hyper-parameter groups.

Group	Hyper-parameter	Example Domain
1. Data preprocessing	Image rotation	[0,30)
	Image cropping	[0,32]
	Whitening	{PCA, ZCA}
2. Model architecture	Number of layers	Z^+
	N_cluster	Z^+
	Kernel	{Linear, RBF, Poly}
3. Training algorithm	Learning rate	R^+
	Weight decay	R^+
	Momentum	R^+

4.2 Distributed Hyper-parameter Tuning

Hyper-parameter tuning usually has to run many different sets of hyper-parameters. Distributed tuning by running these instances in parallel is a natural solution for reducing the tuning time. There are three popular approaches for hyper-parameter tuning, namely grid search, random search [2] and Bayesian optimization [27]. To support these algorithms, we need a general and extensible programming model.

4.2.1 Programming Model

We cluster the hyper-parameters involved in training a machine learning model into three groups as shown in Table 1. Data preprocessing adopts different approaches to normalize the data, augment the dataset and extract features from the raw data, i.e., feature engineering. Rafiki expects the training code to pre-define the possible pre-processing operations, and lets the hyper-parameter tuning algorithm to decide which pre-processing operation to use and the configuration of each operation. For example, image cropping is one pre-processing operation, which is included in the code but whether it is applied or not is decided by the tuning algorithm. Most machine learning models have some tuning knobs about the architecture, e.g., the number of trees in a random forest, number of layers of ConvNets and the kernel function of SVM. The optimization algorithms, especially the gradient based optimization algorithms

[6]https://www.cs.toronto.edu/ kriz/cifar.html

like SGD, have another set of hyper-parameters, including the initial learning rate, the decaying rate and decaying method (linear or exponential), etc.

From Table 1 we can see that the hyper-parameters could come from a range, or a list of numbers or categories. All possible assignments of the hyper-parameters construct the **hyper-parameter space**, denoted as $\mathcal{H}$. Following the convention [8], we call one point in the space as a **trial**, denoted as h. Rafiki provides a *HyperSpace* class with the functions shown in Figure 3 for model contributors to specify the hyper-parameter space.

```
class HyperSpace():
  def add_range_knob(name, dtype, min, max,
          depends=None, pre_hook=None, post_hook=None)

  def add_categorical_knob(name, dtype, list,
          depends=None, pre_hook=None, post_hook=None)
```

Figure 3: HyperSpace APIs.

The first function defines the domain of a hyper-parameter as a range $[min, max)$; *dtype* represents the data type which could be float, integer or string. *depends* is a list of other hyper-parameters whose values directly affect the generation of this hyper-parameter. For example, if the initial learning rate is very large, then we would prefer a larger decay rate to decease the learning rate quickly. Therefore, the decay rate has to be generated after generating the learning rate. To enforce such relation, we can add *learning rate* into the *depends* list and add a *post_hook* function to adjust the value of *learning rate decay*. The second function defines a categorical hyper-parameter, where *list* represents the candidates. *depends, pre_hook, post_hook* are analogous to those of the range knob.

The whole hyper-parameter tuning process for one model over a dataset is called a study. The performance of the trial h is denoted as p_h. A larger p_h indicates a better performance (e.g., accuracy). For distributed hyper-parameter tuning, we have a master and multiple workers for each study communicating via RPC (remote procedure call). The master generates trials for workers, and the workers evaluate the trials. At one time, each worker trains the model with a given trial. The workflow of a study at the master side is explained in Algorithm 1. The master iterates over an event loop to collect the performance of each trial, i.e., $< h, p_h, t >$, and generates the next trial by *TrialAdvisor* that implements the hyper-parameter search algorithm, e.g., random search or Bayesian optimization. It stops when there is no more trials to test or the user configured stop criteria is satisfied (e.g., total number of trials). Finally, the best parameters are put into the parameter server for the inference service. The worker side keeps requesting trials from the master, conducting the training and reporting the results to the master.

4.2.2 *Collaborative Tuning*

In this section, we extend the distributed hyper-parameter tuning framework by proposing a collaborative tuning scheme.

We observe that some hyper-parameters should be changed during training to get better performance. We use the hyper-parameters of the training algorithm, i.e., SGD, as an example. SGD is widely used for training machine learning models. Typically the training loss stays in a plateau after a while, and then drops suddenly if we decrease the learning rate of SGD, e.g., from 0.1 to 0.01 and from 0.01 to 0.001[10]. If we fix the model architecture and tune the hyper-parameters from group 1 and 3 in Table 1, then the model parameters trained using one hyper-parameter trial should be reused

Algorithm 1 Study(HyperTune conf, TrialAdvisor adv)

```
 1: num = 0
 2: while conf.stop(num) do
 3:     msg = ReceiveMsg()
 4:     if msg.type == kRequest then
 5:         trial = adv.next(msg.worker)
 6:         if trial is nil then
 7:             break
 8:         else
 9:             send(msg.worker, trial)
10:         end if
11:     else if msg.type == kReport then
12:         adv.collect(msg.worker, msg.p, msg.trial)
13:     else if msg.type == kFinish then
14:         num += 1
15:         if adv.is_best(msg.worker) then
16:             send(msg.worker, kPut)
17:         end if
18:     end if
19: end while
20: return adv.best_trial()
```

to initialize the same model with another trial. By selecting the parameters from the top performing trials to continue with the hyper-parameter tuning, the old trials are just like pre-training [15]. In fact, a good model initialization results in faster convergence and better performance[29]. Users typically fine-tune popular ConvNet architectures over their own datasets. Consequently, this collaborative tuning scheme is likely to converge to a good state quickly.

It is not straightforward to apply collaborative tuning for model architecture related hyper-parameters. This is because when the architecture changes, the parameter shapes also change. For example, the parameters for a convolution layer with filter size 3x3 cannot be used to initialize another convolution layer whose filter size is 5x5. However, if some sub-networks are shared across different architectures [22], the parameters of the sub-networks can be tuned collaboratively. During training, parameters of all sub-networks are stored in the parameter server. For every trial, a specific architecture is created and its parameters (a subset of all parameter) are updated.

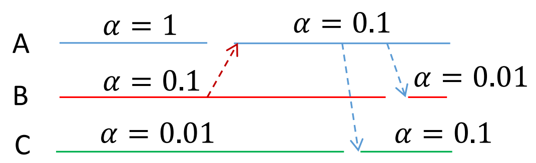


Figure 4: A collaborative tuning example.

The whole process of collaborative tuning is illustrated using an example in Figure 4, where 3 workers are running to tune the hyper-parameters (e.g., the learning rate) of the same model. After a while, the performance of worker A stops increasing. The training stops automatically according to early stopping criteria, e.g the training loss is not decreasing for 5 consecutive epochs. Early stopping is widely used for training machine learning models[7], which is cheaper than training for a fixed number of epochs. The new trial on worker A uses the parameters from the current best worker, which is B. The work done by B serves as the pre-training for the

[7]`https://keras.io/callbacks/#earlystopping`

new trial on A. Similarly, when C stops, the master instructs it to start a new trial using the parameters from A, whose model has the best performance at that time.

The control flow for this collaborative tuning scheme is described in Algorithm 2. It is similar to Algorithm 1 except that the master instructs the worker to save the model parameters (Line 9) if the model's performance is significantly larger than the current best performance (Line 8). The performance difference, i.e., *conf.delta* is set according to the user's expectation about the performance of the model. For example, for MNIST image classification, an improvement of 0.1% is very large as the current best performance is above 99%.

We notice that bad parameter initialization degrades the performance in our experiments. This is very serious for random search. Because the checkpoint from one trial with poor accuracy would affect the next trials as the model parameters are initialized into a poor state. To resolve this problem, a α-greedy strategy is introduced in our implementation, which initializes the model parameters either by random initialization or using the parameters from the parameter server. A threshold α represents the probability of choosing random initialization and (1-α) represents the probability of using pre-trained parameters. α decreases gradually to decrease the chance of random initialization, i.e., increasing the chance of CoStudy. This α-greedy strategy is widely used in reinforcement learning to balance the exploration and exploitation.

Algorithm 2 CoStudy(HyperTune conf, TrialAdvisor adv)

```
1:  num = 0, best_p = 0
2:  while conf.stop(num) do
3:      msg = ReceiveMsg()
4:      if msg.type == kRequest then
5:          ...                                  ▷ // same as Algorithm 1
6:      else if msg.type == kReport then
7:          adv.collect(msg.worker, msg.p, msg.trial)
8:          if msg.p - best_p > conf.delta then
9:              send(msg.worker, kPut)
10:             best_p = msg.p
11:         else if adv.early_stopping(msg.worker, conf) then
12:             send(msg.worker, kStop)
13:         end if
14:     else if msg.type == kFinish then
15:         num += 1
16:     end if
17: end while
18: return adv.best_trial()
```

5. INFERENCE SERVICE

Inference service provides real-time request serving by deploying the trained model. Other services, like database services, simply optimize the throughput with the constraint on latency, which is set manually as a service level objective (SLO), denoted as τ, e.g., $\tau = 0.1$ seconds. For machine learning inference services, accuracy becomes an important optimization objective. The accuracy refers to a wide range of performance measurements, e.g., negative error, precision, recall, F1, area under curve, etc. A larger value (accuracy) indicates better performance. If we set latency as a hard constraint as shown in Equation 4, overdue requests would get 'time out' responses. Typically, a delayed response is better than an error of 'time out' for the application users. Hence, we process the requests in the queue sequentially following FIFO (first-in-first-out). The objective is to maximize the accuracy and minimize the exceeding time according to τ. However, typically, there is a trade-off between accuracy and latency. For example, ensemble modeling with more models increases both the accuracy and the latency. We shall optimize the model selection for ensemble modeling in Section 5.2. Before that, we discuss a simpler case with a single inference model. Table 2 summarizes the notations used in this section.

Table 2: Notations.

Name	Definition
S	request list
M	model list
τ	latency requirement
$b \in B$	one batch size from a candidate list
q_k	the $k-$th oldest requests in the queue
$q_{:k}$	is the oldest k requests
$q_{k:}$	is the latest $\lvert Q\rvert - k$ requests
$c(b)$	inference time for batch size b
$c(m, b)$	inference time for model m and batch size b
$w(s)$	waiting time for a request s in the queue
$l(s)$	latency (waiting + inference time) of a request
β	balancing factor between accuracy and latency
$\mathbf{v}$	binary vector for model selection
$R()$	reward function over a set of requests

$$\max Accuracy(S) \quad (4)$$
$$\text{subject to } \forall s \in S, l(s) < \tau$$

5.1 Single Inference Model

When there is only one single model deployed for an application, the accuracy of the inference service is fixed from the system's perspective. Therefore, the optimization objective is reduced to minimizing the exceeding time, which is formalized in Equation 5.

$$\min \frac{\sum_{s\in S} max(0, l(s) - \tau)}{|S|} \quad (5)$$

The latency $l(s)$ of a request includes the waiting time in the queue $w(s)$, and the inference time which depends on the model complexity, hardware efficiency (i.e., FLOPS) and the batch size. The batch size decides the number of requests to be processed together. Modern processing units, like CPU and GPU, exploit data parallelism techniques (e.g., SIMD) to improve the throughput and reduce the computation cost. Hence, a large batch size is necessary to saturate the parallelism capacity. Once the model is deployed, the model complexity and hardware efficiency are fixed. Therefore, Rafiki tunes the batch size to optimize the latency.

To construct a large batch, e.g., with b requests, we have to delay the processing until all b requests arrive, which may incur a large latency for the old requests if the request arriving rate is low. The optimal batch size is thus influenced by SLO τ, the queue status (e.g., the waiting time), and the request arriving rate which varies along time. Since the inference time of two similar batch sizes varies little, e.g., b=8 and b=9, a candidate batch size list should include values that have significant difference with each other w.r.t the inference time, e.g., $B = \{16, 32, 48, 64, ...\}$. The largest batch size is determined by the system (GPU) memory. $c(b)$, the latency of processing a batch of b requests $b \in B$, is determined by the hardware resource (e.g., GPU memory) and utilization (e.g., in multi-tenant environment), and the model's complexity. Figure 5

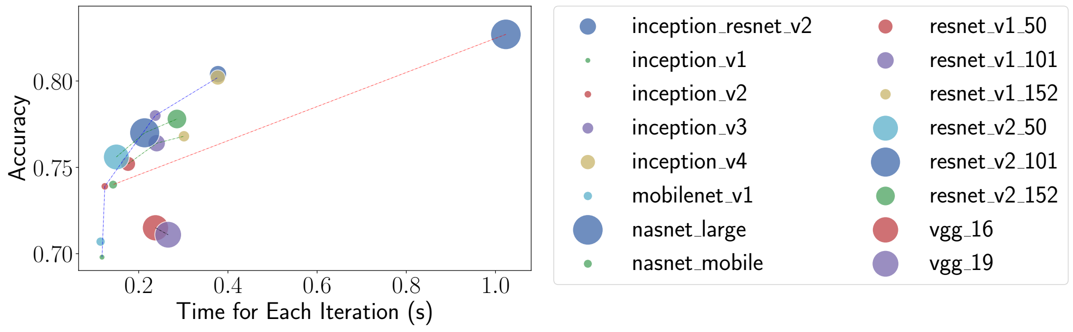


Figure 5: Accuracy, inference time and memory footprint of popular ConvNets.

shows the inference time, memory footprint and accuracy of popular ConvNets trained on ImageNet[8]. The accuracy is measured based on the top-1 prediction of images from the validation dataset of ImageNet. The inference time and memory footprint is averaged over 50 iterations, each with 50 images (i.e., batch size=50).

Algorithm 3 shows a greedy solution for this problem. It always applies a large batch size. If the queue length (i.e., number of requests in the queue) is larger than the largest batch size $b = \max(B)$, then the oldest b requests are processed in one batch. Otherwise, it waits until the oldest request (q_0) is about to overdue as checked by Line 8. b is the largest batch size in B that is smaller or equal to the queue length (Line 8). δ is a back-off constant, which is equivalent to reducing the batch size in Additive-Increase-Multiplicative-Decrease scheme (AIMD)[4], e.g., $\delta = 0.1\tau$.

Algorithm 3 Inference(Queue q, Model m)

```
1:  while True do
2:      $b = \max B$
3:      if $len(q) >= b$ then
4:          m.infer($q_{0:b}$)
5:          deque($q_{0:b}$)
6:      else
7:          $b = \max\{b \in B, b <= len(q)\}$
8:          if $c(b) + w(q_0) + \delta >= \tau$ then
9:              m.infer($q_{0:b}$)
10:             deque($q_{0:b}$)
11:         end if
12:     end if
13: end while
```

5.2 Multiple Inference Models

Ensemble modeling is an effective and popular approach to improve the inference accuracy. To give an example, we compare the performance of different ensemble of 4 ConvNets as shown in Figure 6. Majority voting is applied to aggregate the predictions. The accuracy is evaluated over the validation dataset of ImageNet. Generally, with more models, the accuracy is better. The exception is that the ensemble of resnet_v2_101 and inception_v3, which is not better than the single best model, i.e., inception_resnet_v2. In fact, the prediction of the ensemble modeling is the same as inception_v3 because when there is a tie, the prediction from the model with the better accuracy is selected as the final prediction, i.e., inception_v3.

Parallel ensemble modeling by running one model per node (or GPU) is a straight-forward way to scale the system and improve the throughput. However, the latency could be high due to stragglers. For example, as shown in Figure 5, the node running *nasnet_large* would be very slow although its accuracy is high. In addition, ensemble modeling is also costly in terms of throughput when compared with serving different requests on different nodes (i.e., no ensemble). We can see that there is a trade-off between latency (throughput) and accuracy, which is controlled by the model selection. Note that the model selection is different to that in Section 4. Here we assume that multiple models have already been trained, and deployed on separate workers. We select some of them online for ensemble modelling. In other words, the models used for predicting the first batch of requests could be different to the models for processing the second batch of requests. If the requests arrive slowly, we simply run all models for each batch to get the best accuracy. However, when the request arriving rate is high, like in Section 5.1, we have to select different models (i.e., workers) for different requests to increase the throughput and reduce the latency. In addition, the model selection for the current batch also affects the next batch. For example, if we use all models for a batch, the next batch has to wait until at least one model finishes.

To solve Equation 4, we have to balance the accuracy and latency to get a single optimization objective. We move the latency term into the objective as shown in Equation 6. It maximizes a reward function R related to the prediction accuracy and penalizes overdue requests. β balances the accuracy and the latency in the objective. If the ground truth of each request is not available for evaluating the accuracy, which is the normal case, we have to find a surrogate accuracy.

[8]https://github.com/tensorflow/models/tree/master/research/slim/

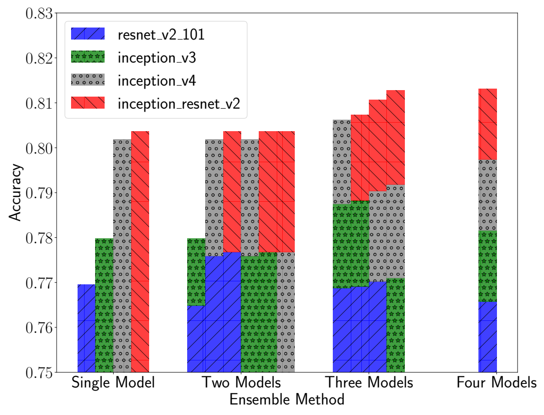


Figure 6: Accuracy of ensemble modeling with different models.

$$\max R(S) - \beta R(\{s \in S, l(s) > \tau\}) \quad (6)$$

Like the analysis for single inference model, we need to consider the batch size selection as well. It is difficult to design an optimal policy for this complex decision making problem, which decides both the model selection and batch size selection. In this paper, we propose to optimize Equation 6 using reinforcement learning (RL). RL optimizes an objective over a long term by trying different actions and entering the corresponding states to collect rewards. By setting the reward as Equation 6 and defining the actions to be model selection and batch selection, we can apply RL for our optimization problem.

RL has three core concepts, namely, the action, reward and state. Once these three concepts are defined, we can apply existing RL algorithms to do optimization. We define the three concepts w.r.t our optimization problem as follows. First, the state space consists of : a) the queue status represented by the waiting time of each request in the queue. The waiting time of all requests form a feature vector. To generate a fixed length feature vector, we pad with 0 for the shorter queues and truncate the longer queues. b) the model status represented by a vector including the inference time for different models (i.e., workers) with different batch sizes, i.e., $c(m, b), m \in M, b \in B$, and the left time to finish the existing requests dispatched to it. For multi-tenant environment, $c(m, b)$ varies as the workload of the worker (i.e., GPU) changes. Therefore, the latest $c(m, b)$ of each worker is recorded and used in RL. The two feature vectors are concatenated into a state feature vector, which is the input to the RL model for generating the action.

Second, the *action* decides the batch size b and model selection represented by a binary vector $\mathbf{v}$ of length $|M|$ (1 for selected; 0 for unselected). The action space size is thus $(2^{|M|} - 1) * |B|$. We exclude the case where $\mathbf{v} = \mathbf{0}$, i.e., none of the models are selected.

Third, following Equation 6, the *reward* for one batch of requests without ground truth labels is defined in Equation 7, $a(M[\mathbf{v}])$ is the accuracy of the selected models (ensemble modeling). In our experiment, we assume that there is no concept drift[4] from the validation data to the online request data. Hence, the validation accuracy reflects the performance of the model(s) and is used as the surrogate accuracy. In real applications, we can update the model accuracy periodically based on the prediction feedback [4]. In the experiment, we use the ImageNet's validation dataset to evaluate the accuracy of different ensemble combinations for image classification. The results are shown in Figure 6. The reward shown in Equation 7 considers the accuracy, the latency (indirectly represented by the number of overdue requests) and the number of requests.

$$a(M[\mathbf{v}]) * (b - \beta|\{s \in \text{batch}|l(s) > \tau\}|) \quad (7)$$

With the state, action and reward well defined, we apply the actor-critic algorithm [24] to optimize the overall reward by learning a good policy for selecting the models and batch size.

6. SYSTEM IMPLEMENTATION

In this section, we introduce the implementation details of Rafiki, including the cluster management, data and parameter storage, and failure recovery.

6.1 Cluster Management

Rafiki uses Kubernetes (or Docker Swarm) to manage the Docker containers for the masters, workers, data servers and parameter servers as shown in Figure 7. Docker container simplifies the environment setup. The training code, inference code, hyper-parameter tuning algorithms, and ensemble modeling approaches are deployed in separate Docker containers. Every time a new job is submitted, the corresponding master or worker containers are launched by the Rafiki admin (See Figure 2). The masters generate hyper-parameters for training or conduct ensemble modelling for inference. Rafiki prefers to locate the master and workers for the same job in the same physical node to avoid network communication overhead.

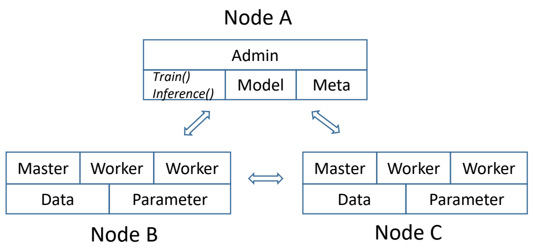


Figure 7: Rafiki cluster topology.

6.2 Data and Parameter Storage

Deep learning models are typically trained over large datasets that would consume a lot of space if stored in CPU memory. Therefore, Rafiki stores the training data in the distributed data storage, e.g., HDFS. Users upload their datasets via Rafiki utility functions, e.g., $rafiki.upload$ (see Figure 2). The training dataset is downloaded to a local directory before the training starts. In this way, Rafiki supports any format of data as long as the data is organized following the requirement (specified in documentation) of the training and inference code.

For model parameters, Rafiki implements a distributed parameter server based on Redis. During training, there is one database for each model, storing the best version of parameters from hyper-parameter turning. This database is kept in-memory as it is accessed and updated frequently. Once the hyper-parameter tuning finishes, Rafiki dumps the parameter database of each model onto disk. The path of the dumped file is stored in the meta database. Later, when we want to deploy the models of one application, Rafiki retrieves the dumped files from the meta database and loads them into memory for the inference workers to access.

6.3 Failure Recovery

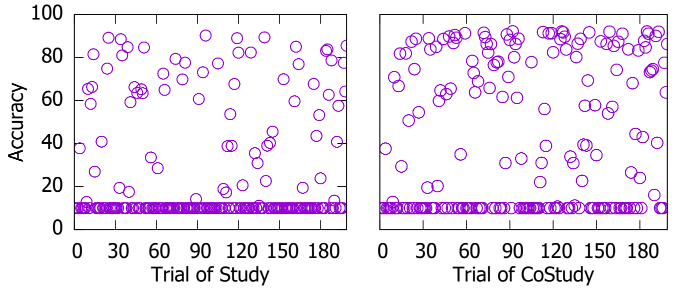
For both the hyper-parameter training and inference services, the workers are stateless. Hence, Rafiki admin can easily recover these nodes by running a new docker container and registering it into the training or inference master. However, for the masters, they have state information. For example, the master for the training service records the current best hyper-parameter trial. The master for the inference service has the state, action and reward for reinforcement learning. Rafiki checkpoints these (small) state information of masters for fast failure recovery.

7. EXPERIMENTAL STUDY

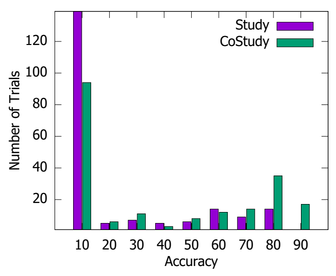
In this section we evaluate the scalability, efficiency and effectiveness of Rafiki for hyper-parameter tuning and inference service optimization. The experiments are conducted on three machines, each with 3 Nvidia GTX 1080Ti GPUs, 1 Intel E5-1650v4 CPU with 64GB memory connected by 1Gbps Ethernet card.

7.1 Evaluation of Hyper-parameter Tuning

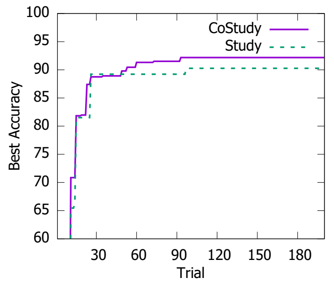
Task and Dataset We test Rafiki's distributed hyper-parameter tuning service by running it to tune the hyper-parameters of deep



(a)

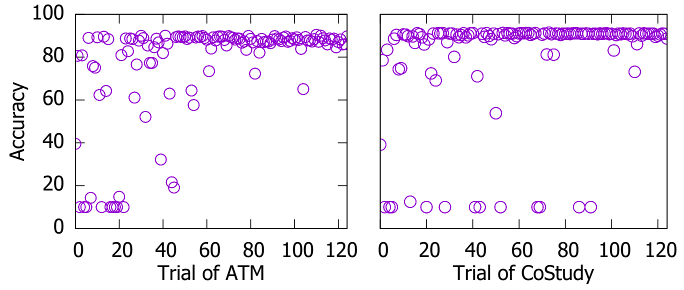


(b)

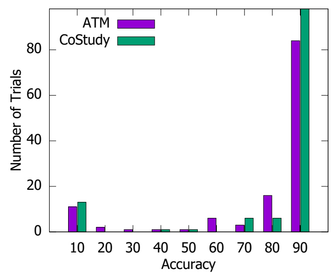


(c)

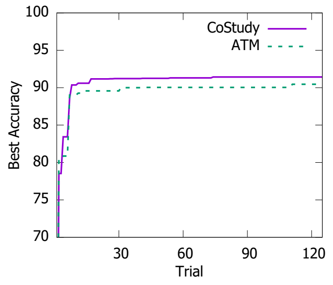
Figure 8: Hyper-parameter tuning based on random search.



(a)



(b)



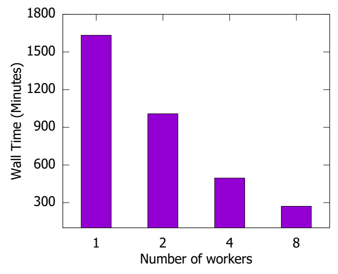
(c)

Figure 9: Hyper-parameter tuning based on Bayesian optimization.

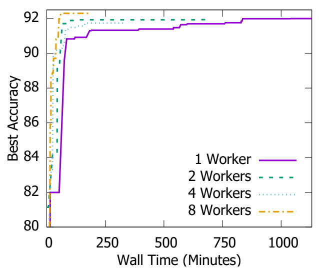
ConvNets over CIFAR-10 for image classification. CIFAR-10 is a popular benchmark dataset with RGB images from 10 categories. Each category has 5000 training images (including 1000 validation images) and 1000 test images. All images are of size 32x32. There is a standard sequence of preprocessing steps for CIFAR-10, which subtracts the mean and divides the standard deviation from each channel computed on the training images, pads each image with 4 pixels of zeros on each side to obtain a 40x40 pixel image, randomly crops a 32x32 patch, and then flips (horizontal direction) the image randomly with probability 0.5.

We fix the ConvNet architecture to be the same as shown in Table 5 of [28], which has 8 convolution layers. The hyper-parameters to be tuned are from the optimization algorithm, including momentum, learning rate, weight decay coefficients, dropout rates, and standard deviations of the Gaussian distribution for weight initialization. We run each trial with early stopping, which terminates the training when the validation loss stops decreasing.

We compare the naïve distributed tuning algorithm, i.e., *Study* (Algorithm 1) and the collaborative tuning algorithm. i.e., *CoStudy* (Algorithm 2) with different *TrialAdvisor* algorithms. Figure 8 shows the comparison using random search [2] for *TrialAdvisor*. In particular, each point in Figure 8a stands for one trial. We can see that the top area for CoStudy is denser than that for Study. In other words, CoStudy is more likely to get better performance. This is confirmed in Figure 8b, which shows that CoStudy has more trials with high accuracy (i.e., accuracy >50%) than Study, and has fewer trials with low accuracy (i.e., accuracy $\leq$50%). Figure 8c illustrates the tuning progress of the two approaches, where each point on the line represents the best performance among all trials conducted so far. We can observe that CoStudy is faster than Study and achieves better accuracy than Study. Notice that the validation accuracy is very high (>91%). Therefore, a small difference (1%) indicates a significant improvement.



(a)



(b)

Figure 10: Scalability test of distributed hyper-parameter tuning.

Figure 9 compares Study and CoStudy using Gaussian process based Bayesian Optimization (BO)[9] for TrialAdvisor. Study with BO is actually the same as ATM [30]. Comparing Figure 9a and Figure 8a, we can see there are more points in the top area of BO figures. In other words, BO is better than random search, which has been observed in other papers [27]. In Figure 9a, CoStudy has a few more points in the right bottom area than ATM (i.e., Study). After doing an in-depth inspection, we found that those points were trials initialized randomly (α is large) instead of from pre-trained models. For ATM, it always uses random initialization, hence the BO algorithm has a fixed prior about the initialization method. However, for CoStuy, its initialization is from pre-trained models for most time. The random initialization trials change the prior and thus get biased estimation about the Gaussian process, which leads to poor accuracy. Since we are decaying α to reduce the chance of random initialization, there are fewer and fewer points in the right

[9]https://github.com/scikit-optimize

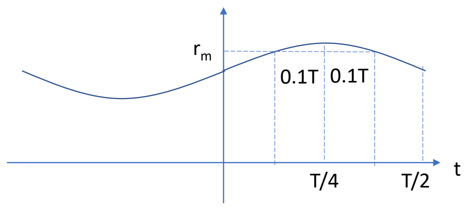


Figure 11: Sine function for controlling the request arriving rate.

bottom area. Overall, CoStudy achieves better accuracy as shown in Figure 9b and Figure 9c.

We study the scalability of Rafiki by varying the number of workers. Figure 10 compares the jobs running over 1, 2, 4 and 8 GPUs respectively. Each point on the curves in the figure represents the best validation performance among all trials that have been tested so far. The x-axis is the wall clock time. We can see that with more GPUs, the tuning becomes faster. It scales almost linearly this is because the communication cost is very small. Workers do not communicate with each other and worker-master communication happens once per trial.

7.2 Evaluation of Inference Optimization

We use image classification as the application to test the optimization techniques introduced in Section 5. The inference models are ConvNets trained over the ImageNet [13] benchmark dataset. ImageNet is a popular image classification benchmark with many open-source ConvNets trained on it (Figure 5). It has 1.2 million RGB training images and 50,000 validation images.

Our environment simulator randomly samples images from the validation dataset as requests. The number of requests is determined as follows. First, to simulate the scenario with very high peak arriving rate and very low peak arriving rate, we run two sets of experiments by generating the arriving rate based on the maximum throughput r_u and minimum throughput r_l respectively. Specifically, the maximum throughput is the sum of all models' largest throughput, which is achieved when all models run asynchronously to process different batches of requests. In contrast, when all models run synchronously, the slowest model's minimum throughput is just the overall minimum throughput. Second, we use a sine function to define the request arriving rate, i.e., $r = \gamma sin(t) + b$. The slope γ and intercept b are derived by solving Equation 8 and 9, where T is the cycle period which is configured to be $500 \times \tau$ in our experiments. The first equation is to make sure that more requests than r_u (or r_l) are generated for 20% of each cycle period (Figure 11, $T \times 20\% = 0.2T$). In this way, we are simulating the scenario where there are overwhelming requests coming at times. The second equation is to make sure that the highest arriving rate is not too large, otherwise the request queue would be filled up very quickly and new requests have to be dropped. Finally, a small random noise ϕ is applied over r to prevent the RL algorithm from remembering the $sine$ function. To conclude, the number of new requests is $\delta \times (\gamma sin(t) + b) \times (1 + \phi), \phi \sim \mathcal{N}(0, 0.1))$, where δ is the time span between the last invocation of the simulator and the current time. In the following figures, the dashed curve represents the request arriving rate.

We compare the greedy algorithm (Algorithm 3), the RL algorithm from Section 5.2 and Clipper [4] in terms of the overall accuracy and latency (measured by the number of overdue requests). Clipper selects all models for ensemble modeling as they give the best accuracy in our experiments. It prefers large throughput (i.e., batch size) as long as the processing time $c(m, b)$ is smaller than τ. Note that it does not consider the request waiting time. Since the processing time of the largest batch size, i.e., b=64, is smaller than τ in our experiments, Clipper always selects the largest batch size and uses all available models for ensemble modeling. For the greedy and RL algorithm, they consider both the queuing time and the processing time for the latency, which is more realistic.

$$k \times sin(T/2 - 0.2 \times 2T/2) + b = r_u \text{ or } r_l \quad (8)$$

$$k \times sin(T/2) + b = 1.1 \times r_u \text{ or } r_l \quad (9)$$

7.2.1 Single Inference Model

We use inception_v3 trained over ImageNet as the single inference model. The state of the RL algorithm is defined to be the same as that in Section 5.2 except that the model related status is removed since there is only a single model. The batch size list is $B = \{16, 32, 48, 64\}$. The maximum throughput is $\max b/c(b) = 64/0.23 = 272$ images per second and the minimum throughput is $\min b/c(b) = 16/0.07 = 228$. We set $\tau = c(64) \times 2 = 0.56$s. The computation time for the RL algorithm to make a decision is about 0.02s, which is far less than the model inference time $c(b)$ and τ. This is because the RL's decision model is a simple multi-layer perceptron model.

First, we compare all algorithms with the arriving rate defined based on r_u. We can see from Figure 12 that after a few iterations, RL performs similarly as the greedy algorithm when the request arriving rate r is high. When r is low, RL performs better than the greedy algorithm. This is because there are a few requests left when the queue length does not equal to any batch size in Line 7 of Algorithm 3. These left requests are likely to overdue because the new requests are coming slowly to form a new batch. Clipper always selects $b = 64$, which results in a long queuing time when the arriving rate is low. Hence, there are overdue requests. When the arriving rate is high, the optimal selection is $b = 64$, and the performance (measured by the number of overdue requests) is the same as the greedy algorithm.

Second, we compare all algorithms with the arriving rate defined based on the minimum throughput r_l. We can see from Figure 13 that since the arriving rate is smaller than that in Figure 12, there are fewer overdue requests. In addition, RL learns to select the proper batch size adaptively and thus has fewer overdue requests (nearly 0) than the greedy algorithm. Clipper performs worst as it wastes much time on waiting for enough requests (64) to form a complete batch.

7.2.2 Multiple Inference Models

In the following experiments, we select inception_v3, inception_v4 and inception_resnet_v2 to construct the model list M. The maximum throughput r_u and minimum throughput r_l are 572 requests per second and 128 requests/second respectively. In the following experiments, RL selects a subset of models (one per worker) and a batch size for ensemble modeling. The computation time for decision making is about 0.024s, which is a bit higher than that for single model inference because the decision space is larger. Clipper always uses all models and the largest batch size. The selected workers run synchronously. The greedy algorithm adopts different strategies (explained below) to select models for ensemble modeling. Algorithm 3 is applied for batch size selection.

We first generate the request rate using the maximum throughput, i.e., r_u. For this case, the greedy algorithm runs all workers asynchronously to process different batches. There is no ensemble modeling. We can see that RL has better accuracy (Figure 14a versus 14b) and fewer overdue requests (Figure 14c versus 14d) than

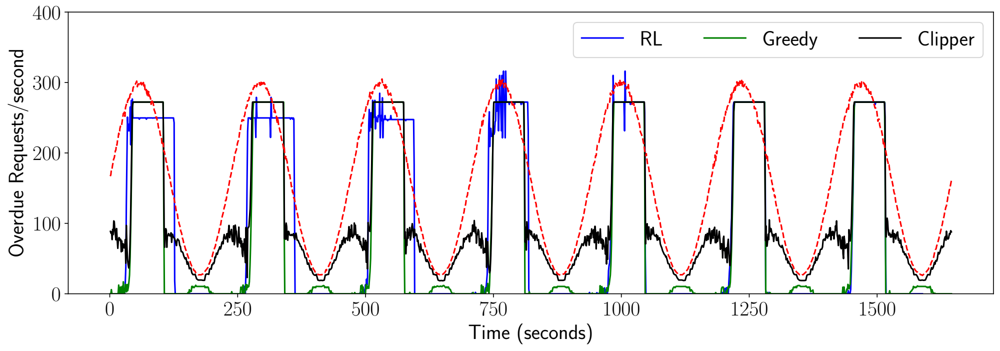


Figure 12: Comparing number of overdue requests for single model inference (arriving rate generated based on r_u)

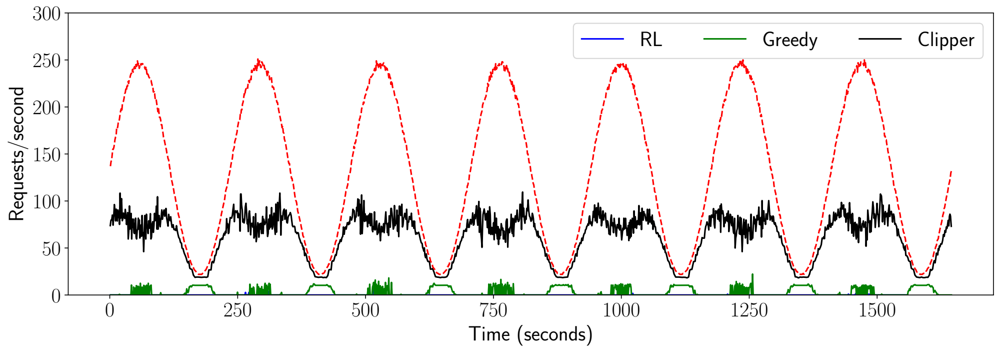
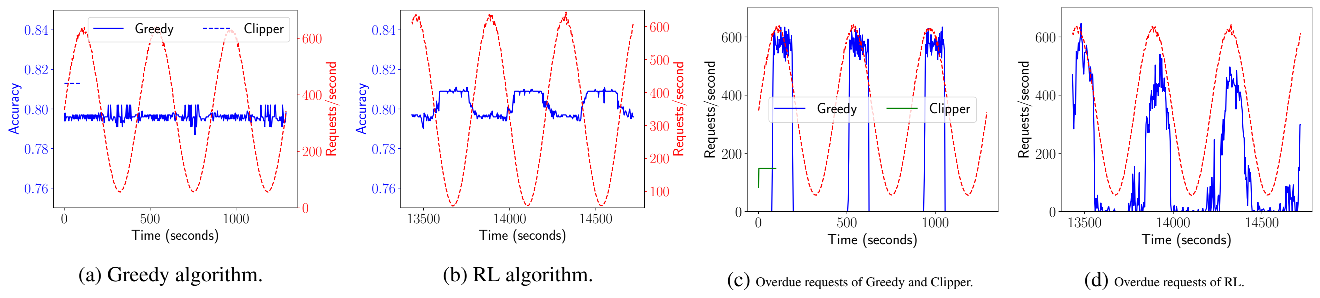
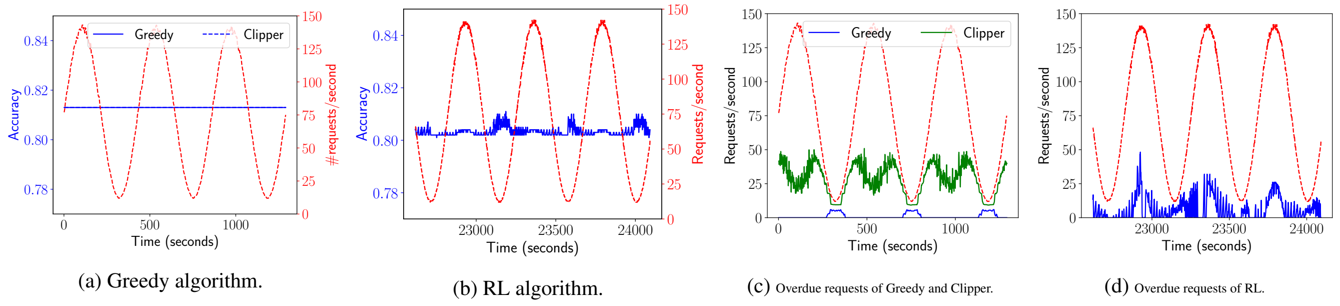


Figure 13: Comparing number of overdue requests for single model inference (arriving rate generated based on r_l).



(a) Greedy algorithm. (b) RL algorithm. (c) Overdue requests of Greedy and Clipper. (d) Overdue requests of RL.

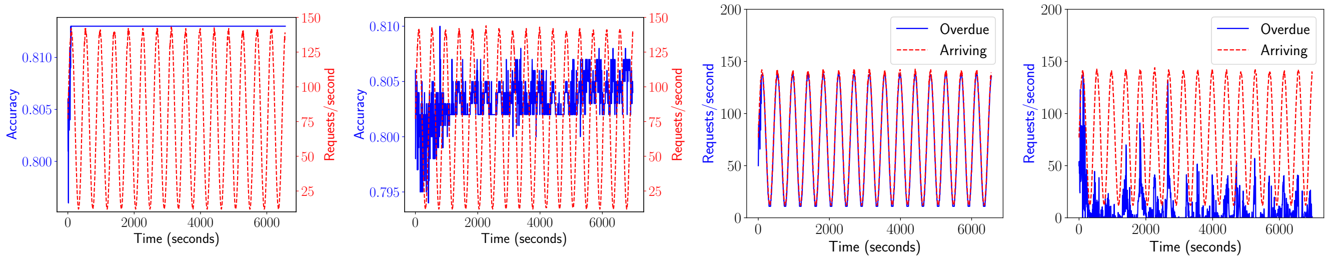
Figure 14: Multiple model inference with the arriving rate generated based on r_u.



(a) Greedy algorithm. (b) RL algorithm. (c) Overdue requests of Greedy and Clipper. (d) Overdue requests of RL.

Figure 15: Multiple model inference with the arriving rate generated based on r_l.

the greedy algorithm. This is because it is adaptive to the request arriving rate. When the rate is high, it uses fewer models to serve the same batch to improve the throughput and reduce the overdue requests. When the rate is low, it uses more models to serve the same batch to improve the accuracy. Clipper uses all models for ensemble modeling, which results in high accuracy as shown in Figure 14a. However, the throughput is low. Therefore, the queue overflows quickly and the number of overdue requests increases



(a) $\beta = 0$. (b) $\beta = 1$. (c) $\beta = 0$. (d) $\beta = 1$.

Figure 16: Comparison of the effect of different β for the RL algorithm.

quickly, which makes the program crash after running for a short period.

Second, we generate the request rate using the minimum throughput, i.e., r_l. For this case, the greedy algorithm runs all workers synchronous, i.e., selecting all models for ensemble modeling. From Figure 15a and Figure 15b, we can see that the greedy algorithm and Clipper have the same fixed accuracy since they use all models for ensemble modeling. The RL algorithm's accuracy is high (resp. low) when the rate is low (resp. high). Because it uses fewer models to do ensemble modeling when the arriving rate is high. Since the peak request arriving rate is very low, the greedy algorithm is able to handle almost all requests. Similar to Figure 13, some overdue requests of the greedy algorithm is due to the mismatch of the queue size and the batch size in Algorithm 3. Clipper has to wait for enough requests for processing, which wastes some time for the early arrived requests and results in many overdue requests. RL is better than Clipper as it adapts the model and batch size selection based on the arriving rate.

We also compare the effect of different β, namely $\beta = 0$ and $\beta = 1$ in the reward function, i.e., Equation 7. Figure 16 shows the results with the requests generated based on r_l. We can see that when β is smaller, the accuracy (Figure 16a) is higher. This is because the reward function focus more on the accuracy part. Consequently, there are many overdue requests as shown in Figure 16c. In contrast, when $\beta = 1$, the reward function tries to avoid overdue requests by reducing the number of models for ensemble. Therefore, the accuracy and the number of overdue requests are smaller (Figure 16b and Figure 16d).

8. CASE STUDY ON USABILITY

Rafiki has been deployed to support various applications, including food image recognition as a component of a nutrition tacking App[10], disease progression modelling [34] for an integrative healcare analytics system [18], etc. In this section, we demonstrate the end-to-end journey of a Rafiki user, i.e., an application developer, using image classification as an example. We have a database (*food*) storing the food images. Each image record contains the ID and age of the user who uploaded the image, the image taken time and location, and the image path (*img_path*) on the server. The task is to predict the food name from the image and aggregate the food of each user to get his/her food preference or intake summary.

First, the application developer prepares a training dataset, e.g., crawling food images from the internet and labeling them manually. The image classification task requires the images of the same class (i.e., the same food) to be placed under one sub-folder of the dataset directory. Let us denote the dataset directory as *train/*.

[10]www.foodlg.com

Second, the application developer uses Rafiki's training service to train machine learning models for food image classification. The code is shown in Figure 2 (*train.py*), which indicates that there are 120 types of food and the input image will be resized to 256 (height) times 256 (width) before feeding to the model.

Third, the application developer uses Rafiki's inference service to deploy the model(s) after training. The code is shown in Figure 2 (*infer.py*). Once the model is deployed, the application developer executes the SQL query below for the food analytic task, which calls a user-defined function (UDF) *food_name()* to predict the name of the food in each image. The UDF function is similar to the application user code in Figure 2. Note that the function is executed only on the images of the rows that satisfy the condition (age is greater than 60). It saves much time when the table is large.

```
select food_name(img_path) as name, count(*)
from food
where age > 60
group by name;
```

9. ACKNOWLEDGEMENT

This research is supported by the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Programme (Grant No. NRF2016NCR-NCR002-020), National Natural Science Foundation of China (No. 61832001), National Key Research and Development Program of China (No. 2017YFB1201001). The work of Meihui Zhang is supported by China Thousand Talents Program for Young Professionals (3070011 181811).

10. CONCLUSIONS

Complex analytics has become an inherent and expected functionality to be supported by big data systems. However, it is widely recognized that machine learning models are not easy to build and train, and they are sensitive to data distributions and characteristics. It is therefore important to reduce the pain points of implementing and tuning of dataset specific models, as a step towards making AI more usable. In this paper, we present Rafiki to provide the training and inference service of machine learning models, and facilitate complex analytics. Rafiki supports effective distributed hyper-parameter tuning for the training service, and online ensemble modeling for the inference service that is amenable to the trade off between latency and accuracy. The system is evaluated with various benchmarks to illustrate its efficiency, effectiveness and scalability. We also conduct a case study that demonstrates how the system enables a database developer to use deep learning services easily.

11. REFERENCES

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning. In *OSDI 16*, pages 265–283, GA, 2016. USENIX Association.

[2] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.*, 13:281–305, Feb. 2012.

[3] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. P. Kuksa. Natural language processing (almost) from scratch. *CoRR*, abs/1103.0398, 2011.

[4] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica. Clipper: A low-latency online prediction serving system. In *NSDI*, pages 613–627, Boston, MA, 2017. USENIX Association.

[5] C. Curino, E. Philip Charles Jones, R. Popa, N. Malviya, E. Wu, S. Madden, H. Balakrishnan, and N. Zeldovich. Relational cloud: A database-as-a-service for the cloud. *CIDR*, pages 235–240, April 2011.

[6] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[7] J. Gao, W. Wang, M. Zhang, G. Chen, H. V. Jagadish, G. Li, T. K. Ng, B. C. Ooi, S. Wang, and J. Zhou. PANDA: facilitating usable AI development. *CoRR*, abs/1804.09997, 2018.

[8] D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J. E. Karro, and D. Sculley, editors. *Google Vizier: A Service for Black-Box Optimization*, 2017.

[9] H. Hacigumus, B. Iyer, and S. Mehrotra. Providing database as a service. In *ICDE*, pages 29–38, Feb 2002.

[10] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[11] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size. *CoRR*, abs/1602.07360, 2016.

[12] D. Kang, J. Emmons, F. Abuzaid, P. Bailis, and M. Zaharia. Noscope: Optimizing neural network queries over video at scale. *PVLDB*, 10(11):1586–1597, Aug. 2017.

[13] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, pages 1106–1114, 2012.

[14] L. I. Kuncheva and C. J. Whitaker. Measures of diversity in classifier ensembles and their relationship with the ensemble accuracy. *Mach. Learn.*, 51(2):181–207, May 2003.

[15] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[16] T. Li, J. Zhong, J. Liu, W. Wu, and C. Zhang. Ease.ml: Towards multi-tenant resource sharing for machine learning workloads. *PVLDB*, 11(5):607–620, Jan. 2018.

[17] Y. Li. Deep reinforcement learning: An overview. *CoRR*, abs/1701.07274, 2017.

[18] Z. J. Ling, Q. T. Tran, J. Fan, G. C. H. Koh, T. Nguyen, C. S. Tan, J. W. L. Yip, and M. Zhang. Gemini: An integrative healthcare analytics system. *PVLDB*, 7(13):1766–1771, Aug. 2014.

[19] V. Mnih, A. Puigdomènech Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. *ArXiv e-prints*, Feb. 2016.

[20] A. N. Modi, C. Y. Koo, C. Y. Foo, C. Mewald, D. M. Baylor, E. Breck, H.-T. Cheng, J. Wilkiewicz, L. Koc, L. Lew, M. A. Zinkevich, M. Wicke, M. Ispir, N. Polyzotis, N. Fiedel, S. E. Haykal, S. Whang, S. Roy, S. Ramesh, V. Jain, X. Zhang, and Z. Haque. Tfx: A tensorflow-based production-scale machine learning platform. In *KDD 2017*, 2017.

[21] B. C. Ooi, K. Tan, S. Wang, W. Wang, Q. Cai, G. Chen, J. Gao, Z. Luo, A. K. H. Tung, Y. Wang, Z. Xie, M. Zhang, and K. Zheng. SINGA: A distributed deep learning platform. In *ACM Multimedia*, pages 685–688, 2015.

[22] H. Pham, M. Y. Guan, B. Zoph, Q. V. Le, and J. Dean. Efficient Neural Architecture Search via Parameter Sharing. *ArXiv e-prints*, Feb. 2018.

[23] C. Ré, D. Agrawal, M. Balazinska, M. I. Cafarella, M. I. Jordan, T. Kraska, and R. Ramakrishnan. Machine learning and databases: The sound of things to come or a cacophony of hype? In *SIGMOD*, pages 283–284, 2015.

[24] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[25] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, and M. Young. Machine learning: The high interest credit card of technical debt. In *SE4ML: Software Engineering for Machine Learning (NIPS 2014 Workshop)*, 2014.

[26] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[27] J. Snoek, H. Larochelle, and R. P. Adams. Practical Bayesian Optimization of Machine Learning Algorithms. *ArXiv e-prints*, June 2012.

[28] J. Snoek, O. Rippel, K. Swersky, R. Kiros, N. Satish, N. Sundaram, M. M. A. Patwary, Prabhat, and R. P. Adams. Scalable Bayesian Optimization Using Deep Neural Networks. *ArXiv e-prints*, Feb. 2015.

[29] I. Sutskever, J. Martens, G. Dahl, and G. Hinton. On the importance of initialization and momentum in deep learning. ICML'13, pages III–1139–III–1147. JMLR.org, 2013.

[30] T. Swearingen, W. Drevo, B. Cyphers, A. Cuesta-Infante, A. Ross, and K. Veeramachaneni. ATM: A distributed, collaborative, scalable system for automated machine learning. In *2017 IEEE BigData 2017, Boston, MA, USA, December 11-14, 2017*, pages 151–162, 2017.

[31] W. Wang, X. Yang, B. C. Ooi, D. Zhang, and Y. Zhuang. Effective deep learning-based multi-modal retrieval. *The VLDB Journal*, pages 1–23, 2015.

[32] W. Wang, M. Zhang, G. Chen, H. Jagadish, B. C. Ooi, and K.-L. Tan. Database meets deep learning: Challenges and opportunities. *ACM SIGMOD Record*, 45(2):17–22, 2016.

[33] H. Zhang, L. Zeng, W. Wu, and C. Zhang. How good are machine learning clouds for binary classification with good features? *CoRR*, abs/1707.09562, 2017.

[34] K. Zheng, W. Wang, J. Gao, K. Y. Ngiam, B. C. Ooi, and J. W. L. Yip. Capturing feature-level irregularity in disease progression modeling. In *CIKM*, pages 1579–1588, 2017.