ThunderML: A Toolkit for Enabling AI/ML Models on Cloud for Industry 4.0

Shrey Shrivastava(✉), Dhaval Patel, Wesley M. Gifford, Stuart Siegel, and Jayant Kalagnanam

IBM Research, 1101 Kitchawan Road, Yorktown Heights, NY, USA
shrey@ibm.com, {pateldha,wmgifford,stus,jayant}@us.ibm.com

Abstract. AI, machine learning, and deep learning tools have now become easily accessible on the cloud. However, the adoption of these cloud-based services for heavy industries has been limited due to the gap between general purpose AI tools and operational requirements for production industries. There are three fundamentals gaps. The first is the lack of purpose built solution pipelines designed for common industrial problem types, the second is the lack of tools for automating the learning from noisy sensor data and the third is the lack of platforms which help practitioners leverage cloud-based environment for building and deploying custom modeling pipelines. In this paper, we present ThunderML, a toolkit that addresses these gaps by providing powerful programming model that allows rapid authoring, training and deployment for Industry 4.0 applications. Importantly, the system also facilitates cloud-based deployments by providing a vendor agnostic pipeline execution and deployment layer.

Keywords: Cognitive computing · IoT sensor data · Machine learning · Deep learning · Purpose built AI pipelines

1 Introduction

The single biggest transformation on the horizon for heavy capital, large manufacturing and industrial companies is Industry 4.0 [14]. The promise of this transformation is to provide a digital semantic representation of the physical manufacturing world consisting of the production plant, the heavy capital assets, and the supply chain network (inventory and warehouse systems, and logistics). This representation is continuously replenished with real time sensor data using high bandwidth, low cost networks to provide up-to-date situational awareness of the enterprise. This, in turn, supports improved operational efficiency and production yields, as well as a deeper understanding of the enterprise's effect on the demand-supply dynamics of the manufacturing ecosystem. The key to leveraging the transformation to Industry 4.0 is developing models to utilize the available data to forecast and optimize enterprise operation.

J. Miller et al. (Eds.): ICWS 2019, LNCS 11512, pp. 163–180, 2019.
https://doi.org/10.1007/978-3-030-23499-7_11

Although AI technologies are now mature enough to provide industrial-strength solutions that bring better efficiencies and asset availability [3,5,20], adoption within the heavy industries still lags behind that of several other domains. There are three main obstacles to more widespread adoption of AI. The first is the lack of purpose-built AI solutions designed to improve industrial operations. The second is the lack of tools for automating the construction of AI models over IoT sensor data which are often highly non-linear, contain dynamic state transitions, exhibit lag relationships, and require a great deal of specialized feature extraction before being useful for AI modeling. The third is the lack of toolkits that help AI practitioners leverage cloud-based environments for building and deploying custom modeling pipelines.

The system we have developed, ThunderML, attempts to address these issues. Firstly, it provides pre-built solution templates in industrial domain for expediting the creation of AI models. Next, it aims to provide AI practitioners with a toolkit to translate their industrial problem statements into a well defined and executable pipeline. Lastly, it presents the settings to facilitate cloud-based AI environments including the management of training experiments and model deployment in a vendor agnostic way. In this paper we will focus almost exclusively on the latter two capabilities as we think these are the most broadly applicable.

1.1 Challenges of Using Existing Cloud-Based AI Platforms

While cloud-based AI platforms have done much to facilitate adoption of AI by alleviating many of the infrastructure provisioning and maintenance challenges associated with on-premises enterprise AI initiatives, they have not done enough to abstract away some of the complexity of running AI workflows in vendor agnostic ways. Current platforms expect practitioners to know a given vendor's means and methods of interacting with the computing resources without consideration given to providing a common programming model that makes the job of an AI practitioner easier. Cloud-based AI environments, by their very nature, push users towards batch training modes to facilitate data center resource management via a queued execution model. Such batch training modes are problematic for many data scientists who wish to see errors or results in real or near real time in order to make their modeling workflow more efficient.[1]

Another issue is that cloud-offerings typically approach AI from either a black-box perspective which offers users simplicity at the cost of flexibility or through a more complex runtime environment that requires users maintain code artifacts that often have nothing to do with the actual AI tasks at hand[2]. Even with a diverse set of offerings in the market, we feel a gap remains for the AI practitioner community. Cloud AI offerings should be easy to learn and use and provide the right level of complexity and flexibility AI practitioners need.

[1] https://cloud.google.com/blog/topics/research/new-study-the-state-of-ai-in-the-enterprise.

[2] https://aws.amazon.com/blogs/aws/iot-analytics-now-generally-available/.

In order to address these issues, we have developed ThunderML, a Python-based toolkit that makes the creation and deployment of purpose built AI models for industrial applications easier. ThunderML leverages many open source frameworks such as scikit-learn, Tensorflow, and Keras. The extension points are predominantly in terms of how we have built out a series of useful modeling functions and industrial solution templates to expedite the task of building and deploying AI for industrial applications. ThunderML is flexible enough to run on local hardware as well as providing an easier path to using common cloud service provider platforms for enhanced scalability in training and convenient model deployment services.

Before we proceed, it's worth briefly giving a few examples of purpose built industrial solution templates available in ThunderML:

- *Time Series Prediction (TSPred)*: Flexible solution for forecasting time series from historical data in industries.
- *Failure Pattern Analysis (FPA)*: Predicting imminent failures for assets using IoT sensor data and past failure history data;
- *Root Cause Analysis (RCA):* Building interpretable models to assist plant operators track down the root causes for product quality deviances on batch or continuous process lines;
- *Anomaly Analysis:* Building unsupervised/semi-supervised models to identify anomalous behaviors of manufacturing assets;
- *Cognitive Plant Advisor (CPA)*: Combines advanced AI to build a predictive model of one or more key process outputs such as throughput and yield and uses these models within a business objective optimization problem to suggest optimal process settings to plant operators.

In summary, ThunderML can also help alleviate the skills gap issue that has hampered AI adoption in many industries. In our experience, technically adept (but not necessarily experts in AI personnel) can use ThunderML's industry templates and programming interface to enable industry AI models.

1.2 Contribution

Our contribution in this paper is the design and implementation of ThunderML. We elaborately discuss how ThunderML expedites the AI modeling workflow by giving practitioners an easier path for doing advanced modeling work leveraging cloud-based platforms for training and deployment. We then provide a use case to demonstrate the benefits of ThunderML in practice for a very general and widely applicable problem.

Figure 1 shows a high-level schematic representation of ThunderML's component architecture. At the lowest level, "Core AI Toolkits", we leverage common AI frameworks like scikit-learn [17], tensorflow, and keras for building modular AI functions that our purpose built solutions either utilize directly or extend. The next layer, "Pipeline Management", exposes an API that allows users to stitch these modular AI functions and compose multiple path pipelines. This

layer also manages the execution metadata like- the choice of target runtimes (e.g., a local machine, a spark cluster, or a remote cloud vendor execution environment), hyperparameter tuning, the maximum allowable training time, and the model scoring metric. In the next layer, "Pipeline Path Decomposer", the large number of possible modeling pipelines and hyperparameter combinations are decomposed to make them trainable in an parallel fashion (e.g., on a cloud vendor elastic compute service which can queue very large numbers of independent user jobs in an asynchronous fashion). In the final layer, "Cloud Vendor Interaction API", decomposed pipeline paths get packaged and transferred in a way suitable for job submission to a cloud-vendor's remote execution service. This layer handles the particulars of a vendor's platform API requirements to liberate users from the laborious task of learning the details of how to interact with a particular vendor's compute and storage services. It submits jobs, monitors progress, retrieves results, selects among the best of these (based on user-supplied metric) and manages the final deployment of models to the vendor deployment services.

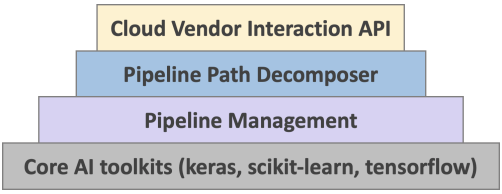

Fig. 1. High-level schematic representation of ThunderML's component architecture.

2 ThunderML: Background

In this section, we establish a few concepts at the abstraction level of activity being performed. ThunderML is based on the foundation of scikit-learn's framework [17] as it is the most commonly used framework in the community, both in academia and industry. We will use one of our purpose built industrial solution templates - **Time Series Prediction (TSPred)** as an example and give concrete examples of the terminology in the context of this problem.

The TSPred is a very pertinent and complex problem in the industry and forms the basis of other time series problems. It provides a good use case for ThunderML for two reasons. Firstly, it demonstrates how a AI problem statements can be converted into a AI pipelines using the ThunderML's unique and flexible programming model (Sect. 2). Secondly, we show how the TSPred benefits from the ThunderML's cloud based execution architectures in a vendor agnostic manner (Sect. 3).

2.1 AI Functions

A **function** is defined as the task surrounding the development of an AI model. This involves transformations on a dataset, training a model on the given dataset, and model performance evaluation. This usually involves the model, and the configuration associated with the modeling activity, like a single set of parameters. Since our focus for ThunderML is it being a toolkit for AI, functions are the fundamental units in higher-level AI tasks. We have created a programming model which allows users to easily define Directed acyclic graphs (DAGs) for controlling the sequence of operation for these AI functions. In the context of TSPred, the AI functions have been defined in four categories- transformers, models, model evaluation, cross-validation.

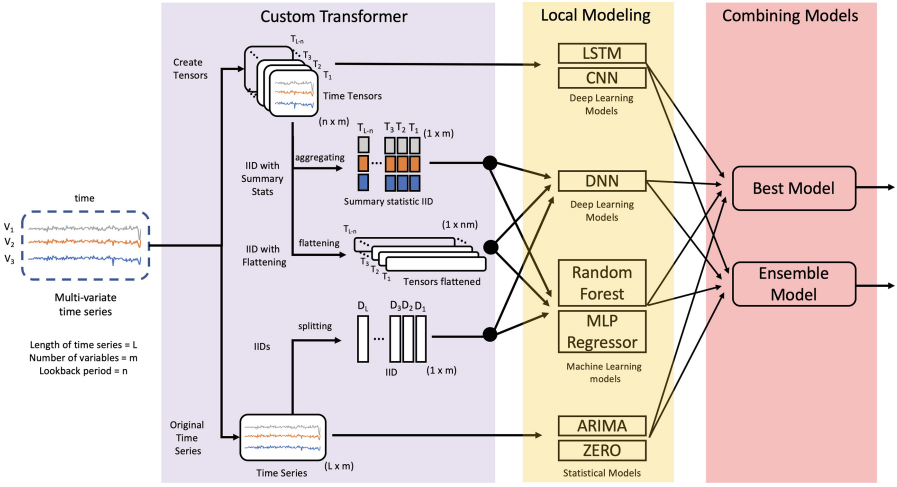

Fig. 2. Architecture of AI pipeline for time series prediction problem.

Custom Transformations: These transformation functions appropriately prepare the data for the modeling activity and it becomes specially important in the TSPred since training time series models requires well-defined time-ordered input and output training data. The time series prediction problem uses a fixed-width sliding window called the **lookback window** and tries to predict the value for future timestamps. For example, in a time series, we can take a sliding window of length n from time t to $t-n+1$ (inclusive) and predict the value at the future timestamp $t+1$ for the case of 1-step ahead prediction. In this case, the 3-dimensional vector of multivariate sliding windows is referred to as a **time tensor** and is the input to the model. Whereas the vector consisting of values at $t+1$ is the output for training purposes. The custom transformer has the job of creation of such input-output pairs for training models in time series prediction problems brings new challenges. The two main challenges which we address are:

1. The temporal nature of the data needs to be retained when creating features from the time series.
2. Different models require different formats of input and output data during modeling.

The Custom Transformer handles the first challenge by considering multiple feature-processing AI functions for time series data. As mentioned above, we have implemented five types of transformer function: identity mapping, time-tensor preparation, window-based feature summary extraction, window-based flattened feature vectors, and time series as-it-is, as shown in Fig. 2. These cover elaborate cases for the use case of time series feature extraction. Depending on the model input, we provide different features to the models. It gives users the flexibility to try different pre-processing combinations for different models and choose the best one suited for their data.

The second challenge with time series is the restriction on each model to ingest the time series data. Since statistical, machine learning and deep learning models have their respective data ingestion policies, we use ThunderML's flexible DAG based programming models to define the flow between the transformers and models show in Fig. 2. This way, we are able to automate the task of data processing and model training easily with discrete functions defined in the pipeline configuration. In TSPred, this eliminates the need for users to deal with the time series data transformations for each model. This process can become even more cumbersome with new feature engineering methods, increasing variety of models and other time series settings. For example, statistical models like zero order and ARIMA take the entire time, whereas other models like CNN, LSTM, etc. require different lookback windows depending on the setting.

Models: In TSPred, we utilize various time series prediction models such as Zero Order model, ARIMA model, Long Short-Term Memory (LSTM) model [9,13], Multilayer Perceptron (MLP) Regressor [18], Random Forest Regressor, Deep Neural Network (DNN), Convolutional neural network (CNN) [12], WaveNet [19] and SeriesNet [11].

Model Evaluation: We also use some AI functions which provide model evaluation capabilities. Our framework leverages scikit-learn's as well as custom model evaluation methods and the pipeline is flexible to use any of them if the models in the pipeline support it.

Cross Validation: We implement the usage of different cross validation (CV) techniques in the form of AI functions. Similar to specifying models and transformers, CV objects can also be provided by the user. CV is critical in the case of time series data as we need to maintain the sequential order. We have implemented several time series CV functions which are illustrated in Fig. 3

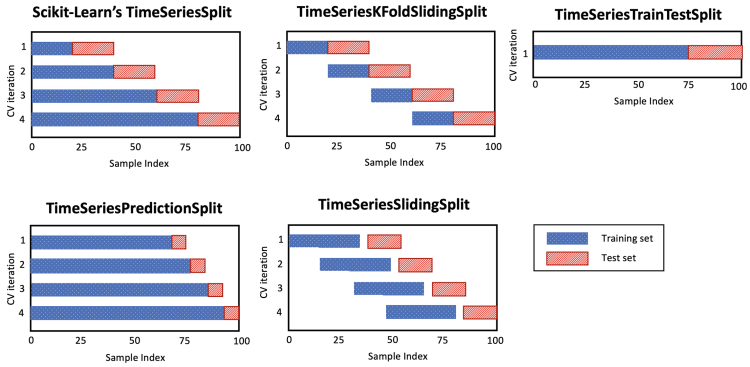

Fig. 3. Comparison of cross validation approaches used for time series models

2.2 AI Flows

An **AI flow** extends the functionality of the AI functions by connecting sequences of tasks like preprocessing the data and post processing the results in 'flows'. Although similar to the machine learning pipelines defined in scikit-learn's framework, AI Flow expands this definition by introducing ThunderML's unique DAG based programming model, providing very high flexibility while defining pipelines. This includes modeling functions, and just like scikit-learn's pipeline, it includes other steps like pre-processing (dimension reduction, scaling, etc.). The AI flow usually consists of multiple sets of parameters for each transformer and estimator, as well as a scoring method (classification-accuracy, ROC, AUC, etc.; regression-r^2, mean absolute error, etc.) based on which the best set of parameters are chosen. Each path from start to end in the Fig. 2 is an example of a AI flow in the context of TSPred.

The process by which the parameters are chosen, or hyperparameter optimization, is conducted by one of the following methods: complete grid search, random grid search and RBFOpt. The parameter grid chosen are model dependent and it consists of values that are known to give better performance. One example of parameter grid specification for ThunderML is described below:

Listing 1.1. Param grid sample

```
keras_param_grid = {'optimizer':[{'adam':{'amsgrad':[False],
                    'beta_1':[0.9],
                    'beta_2':[0.999],
                    'decay':[0.0],
                    'epsilon':[None],
                    'lr': [0.001, 0.01]}}],
                    'kernel_initializer':['glorot_uniform'],
                    'dropout_rate':[0, 0.2, 0.5],
                    'loss':['mean_squared_error']}
```

2.3 AI Pipelines

We will define our **AI pipeline** as a high-level experiment which is capable of exploring and learning from the data by building the best model among a large combination of AI flows. This means trying different AI Flows with parameter space exploration, along with diverse data transformation techniques and evaluation strategies. With the flexible programming model in ThunderML, it is very easy to define and organize large combinations of AI flows very easily. The results from these experiments gives the best AI flow along with the best set of parameters for the given dataset. Apart from TSPred, ThunderML provides AI pipelines for general problems like supervised learning (classification, regression etc.) and unsupervised learning (clustering, outlier detection, etc.) as well as purpose built pipelines mentioned above. The TSPred AI pipeline consists of three major stages:

1. The **custom transformation** stage
2. In the **local modeling stage** stage, the model for different flows run independent from each other and produce independent scores.
3. In the **model combining** stage, we output the results either by providing the top scoring AI flow or the combination of top AI flows using ensemble techniques.

In summary, there are many different ways to build models for time series prediction problems, and the best method may vary from dataset to dataset. Thus, a tool for automatic discovery of best prediction model is needed, which AI practitioners can easily configure based on their problem statement.

2.4 Practical AI Using ThunderML

The ThunderML toolkit leverages popular machine learning frameworks like scikit-learn, Tensorflow, Keras etc. to allow users to define distinct AI flows and quickly build complex AI pipelines. Along with the pipelines, the users can create AI solution templates by defining other stages surrounding the AI pipelines in a modular fashion. To demonstrate the flexibility and easy of ThunderML, a simple pseudo code for TSPred (Fig. 2) has been provided below:

Listing 1.2. Pseudo code for Fig. 2

```
def purpose_built_pipeline():
    Task = AIPipeline()

    Task.add_flow(flow_name='temporal'
                  [[TimeTensor()], #Transformer AI Functions
                  [LSTM(), DNN(), CNN()]] #Model AI Functions
                 )
    Task.add_flow(flow_name='iid',
                  [[IdentityMapping(), WindowFeatures(),
                  WindowFlattening()],
```

```
                    [DNN(), RandomForest(), MLP()]]
                )
    Task.add_flow(flow_name='statistical',
                    [[NoOperation()],
                    [ARIMA(), Zero()]]
                )

    Task.scoring(['best', 'stacking_ensemble'])
    Task.create_graph()
    return Task

TSPred = purpose_built_pipeline()
TSPred.execute()
```

The next advantage ThunderML provides the user is in terms of the flexibility of execution. Once an AI practitioner pragmatically defines their problem statement in the form of AI Pipelines as shown above, ThunderML provides the flexibility of using local hardware for execution as well as option to going through the cloud route with common cloud service provider platforms for enhanced scalability. These are described in greater detail in the next Section.

3 ThunderML: Cloud-Based Execution Architectures

ThunderML **is one** of the first systems which allows you to interact with enterprise cloud services to perform comprehensive machine learning experiments in a programmatic way. The system **developed** wraps the interfaces between the user and the cloud services API, allowing users to use the cloud services transparently and train models easily.

In any AI tasks, the basic components are the dataset and the AI pipeline definitions. With the introduction of cloud, it adds another requirement for the cloud services. Hence in a cloud based architecture for ThunderML, these are the main components:

- **Cloud Service:**
  - ML training and deployment services: With the addition of cloud platform as the machine learning runtime, the user need to have an account on the respective enterprise cloud offerings and valid 'user-credentials' to access the cloud runtime services.
  - Cloud storage service: Most of the ML services use an internal cloud storage service where the results of the training experiment are stored.
- **Component provided by the user:**
  - AI pipeline definition and configuration: An AI pipeline definition provides details on how to execute the AI flows in the cloud runtime environment. The ThunderML enables users to define a complex AI pipeline in a programmatic manner.
  - User data: This can be provided through local execution environment or it can be uploaded to the cloud storage service depending the cloud ML service specification.

- Credential to cloud services: This is the access key to the cloud service and needs to be provisioned by the user once the services have been created.

The steps for setting up the cloud services (services for training, deployment and storage) are generally very well documented in all enterprise cloud offerings. Once the setup for the above components is done correctly, the user can start defining and executing of their AI pipelines with their data seamlessly with ThunderML's execution architecture. The interaction diagram of how ThunderML leverages enterprise cloud service is shown in Fig. 4.

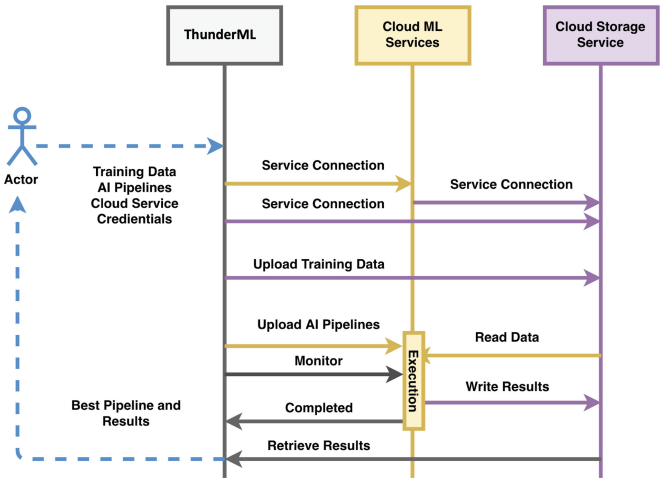

Fig. 4. Interaction diagrams shows how the users can leverage cloud services to run AI pipeline with the help of ThunderML toolkit

The first step in the interaction diagram is providing ThunderML with the training data, AI pipeline and cloud service credentials. ThunderML uses the credentials to connect with the cloud services and creates a client for both the ML and storage services in local execution environment. Using the cloud storage client, ThunderML uploads the user's data to the cloud storage space so that the ML service can use it. Once the data is uploaded, ThunderML configures the AI pipeline to be executed at the ML instance with different settings. There are two architectures proposed here, which provide different execution styles for running AI pipelines. The basic difference between these two architectures are in the following aspects: time, cost, and flexibility. The two execution styles, **coarse** and **fine** execution, are described in greater detail below. ThunderML processes the pipeline configuration before sending them to the ML service. Once the pipeline configuration reaches the ML service, it obtains the data from the storage system and executes the AI pipeline. The training results are generated and stored in the cloud storage. These include the result summary files, log files, trained AI function (for getting trained model weights in the case of deep learning models)

and the parameters. Once this process is completed, ThunderML retrieves the training results from the storage service. Lastly, the cleaning of the cloud storage, removal of any temporary files as well as terminating the kernel is handled by ThunderML.

The results of the AI pipeline is returned from the cloud storage and stored locally for the user. The details of the entire process are hidden from the user; they never have to directly interact with the cloud, and the process is essentially the same as running local python code.

3.1 Coarse Execution

Coarse execution refers to executing a AI pipelines in a more traditional way, i.e. **sequentially** on the cloud's processing power. As shown in Fig. 5, the ThunderML takes the pipeline and its configurations and bundles them as a single package at step 3. This package contains the AI pipeline definition along with the supporting configurations.

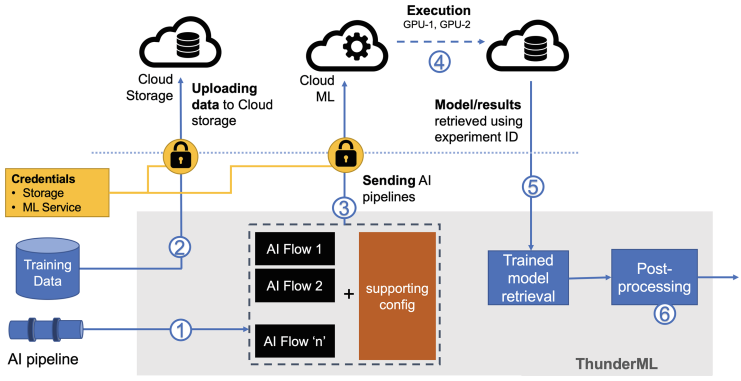

Fig. 5. Coarse execution

Once the AI pipeline definitions reach the ML service, the ThunderML configures the ML service to instantiate a container with on the cloud for executing the pipelines. The container has specific resources allocated to it as defined by the ML service. The pipeline runs in this container and each of the AI flows run one after other in a sequential fashion as shown by the step 4 in Fig. 5.

3.2 Fine Execution

The fine execution architecture leverages the cloud's processing power in a **parallel** manner. ThunderML takes the AI pipeline and splits them into multiple parts, one for each AI flow along with it's associated configurations and creates individual packages for them as shown in step 3 of Fig. 6. Then ThunderML

configures the ML service to instantiate multiple containers, one container per AI flow on the cloud. Since each AI flow receives independent resources, the execution process is expedited as shown in step 4 of Fig. 6. The improvement in execution time might come at the cost of less efficient resource management at the enterprise cloud back-end. Although the resource utilization does not differ much due to the fact that number of calculations in the AI pipeline are independent of the type of execution.

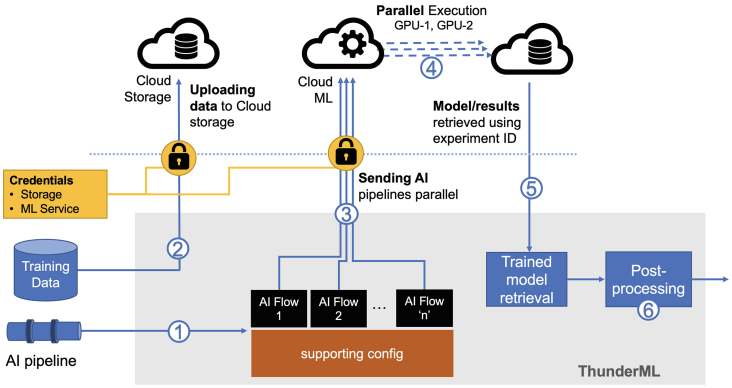

Fig. 6. Fine execution

4 Case Study: Time Series Prediction

In this section, we dive deeper into our implementation of the Time Series Prediction(TSPred) pipeline and provide a concrete example of its utility. We will formally define the time series prediction problem providing some background, then we discuss the different experiments we conducted and the corresponding results.

4.1 Background and Motivation

A time series is a sequence of real valued observations that are captured over a period of time. The time series prediction problem typically takes the form of predicting the future values of some observations over some time horizon, given historical values of those observations. The inputs and outputs may be multivariate or univariate in nature and additional control variables may be present. For simplicity we will focus on the univariate case, but the proposed framework can easily adapt to other settings.

Among all the interesting work on time series analysis such as anomaly detection to root cause analysis, time series prediction (forecasting) is an extremely

common problem studied in numerous domains, including weather forecasting, web traffic forecasting, and stock market prediction. With the increasing deployment of IoT sensors for monitoring industrial assets and manufacturing processes, a huge amount of in-operation time series sensor data is now available.

One distinct characteristic of time series data is that they generally do not follow the IID (independent and identically distributed) assumption, i.e., the data points in time series are correlated to each other in the time domain. The non-IID assumption for time series data makes existing off-the-shelf machine learning solutions such as Google's Cloud AutoML[3], Auto-scikit-learn [7], TPOT [16] less applicable, since extra care needs to be taken when performing cross validation and feature engineering for time series data. Thus, there is a need for a framework that specifically supports time series prediction problems.

4.2 Experiment 1: Coarse and Fine Execution

The purpose of the experiment is to analyze the performance of the two cloud based execution architectures discussed in the Sect. 3 on out TSPred pipeline. Since the purpose of this experiment is to compare the two architectures and not evaluate the accuracy, we will apply some constraints to our experiment. We only use 12 models in the pipeline (listed in Table 1), we run this experiment with only two univariate time-series datasets and each model uses default parameters only. Models with '_10' in their name utilize a lookback window of 10 otherwise it is a lookback of 1, which is basically a model which treats the previous day's reading as an IID point discarding the temporal characteristics. The cross-validation strategy used here is $TimeSeriesTrainTestSplit$ (as shown in Fig. 3) with fraction of train, validation and test portions set to 0.8, 0.1, 0.1. The scoring method for evaluating the model performance in the validation set as well as test set is done using Mean Absolute Error. The results of this experiment are provided in the Table 1.

One thing to consider in this experiment is that while using cloud based services, there is an additional cost associated with packaging the AI pipeline and dataset (if the data is not on cloud services initially) and sending these packages to the cloud ML services through rest API calls. The fine execution splits each AI pipeline into a different package as compared to the coarse execution which just sends a single package for the entire pipeline. This means that fine execution has greater time cost associated with managing execution. From our experiments on the three datasets, we have seen that for uploading the package for an AI pipeline with 12 models, the fine execution takes around 80–90 s, whereas the coarse execution takes around 12–14 s which only sends one package. There is a difference of 70–80 s when running 12 models. Similarly, a cost is added while retrieving the results for each AI flow post execution and coarse execution and adding a difference of 70–80 s. This suggests that the fine execution has an overhead of 150 s when compared to coarse execution of AI pipeline with 12 flows. We run this experiment with 2 epoch values for deep learning models - 500 and

[3] https://cloud.google.com/automl/.

Table 1. Coarse vs. fine performance results

Univariate dataset 1 (length = 229)				
	Epochs = 500			
Model	Coarse (609 s)		Fine (515 s)	
	MAE	Time (s)	MAE	Time (s)
simple_LSTM_10	0.5	39.35	0.51	38.8
Zero model	**0.91**	0.01	0.91	0.01
mlpregressor	0.56	0.11	0.56	0.12
randomforestregressor	0.57	0.02	0.57	0.05
DNN	0.56	2.91	0.56	2.62
DNN_10	0.55	3.24	0.55	2.66
Wavenet_CNN_10	**0.21**	172.36	0.21	168.36
Simple_CNN_10	0.33	7.05	0.34	4.74
deep_CNN_10	0.23	12.91	0.21	11.75
SeriesNet_CNN_10	0.48	75.61	0.51	72.0
mlpregressor_10	**0.22**	**0.2**	**0.22**	**0.19**
randomforestregressor_10	0.55	0.03	0.55	0.06
Univariate dataset 2 (length = 494)				
	Epochs = 500			
Model	Coarse (770 s)		Fine (696 s)	
	MAE	Time (s)	MAE	Time (s)
simple_LSTM_10	0.44	21.26	0.41	22.16
Zero model	0.52	0.0	0.52	0.0
mlpregressor	0.4	0.1	0.4	0.11
randomforestregressor	1.07	0.03	1.07	0.05
DNN	0.3	2.93	0.31	2.5
DNN_10	0.24	2.89	0.24	2.44
Wavenet_CNN_10	0.28	78.71	0.25	81.62
Simple_CNN_10	0.28	5.61	0.29	4.49
deep_CNN_10	0.36	9.65	0.35	6.44
SeriesNet_CNN_10	0.73	49.46	0.71	49.9
mlpregressor_10	0.23	0.15	0.25	0.17
randomforestregressor_10	0.96	0.03	0.96	0.05

1000 and see if any difference is present. The observations from this experiment are noted below:

– **Model Performance**: The score for a model for a given dataset in both Coarse and Fine execution is similar, suggesting that execution architectures

do not affect the accuracy of the models. Sometimes in deep learning models, the scores of a model are drastically different (performing like the baseline model). This can be attributed to random weight initialization in deep learning models.

– **Time of Execution**: The coarse execution take more time than the fine execution in each case. For smallest dataset, the difference between coarse and fine execution (total time) is 80–95 s. With larger datasets this time increase to 90–130 s. Along with the overhead of fine execution, the fine execution is faster than the coarse execution by 220 s–280 s which is one-third of the execution time of the average execution time. This difference will increase with increased data size and larger training run.

4.3 Experiment 2: Univariate Time Series Prediction

In this section we focus on the *univariate time series prediction problem*, where the task is to predict the value of the time series at the next time step. This demonstrates the ability of the ThunderML toolkit to run flexible AI pipelines for a particular learning problem. Since the primary objective is to demonstrate the utility of the pipeline, we will not aggressively search for the best algorithm for the time series prediction problem. Rather, we will demonstrate how multiple models can be explored with the AI pipeline. With proper parameter tuning and training (larger number of epochs for deep learning models), it would be easier to run the same pipeline for surveying which model is the best for the particular problem. We also want to comment on the size of the lookback window which is best for time series prediction for the datasets.

We run this experiment on 15 univariate time-series datasets and 28 models (CNNs, LSTMs, DNNs, Random Forest Regressors, MLP Regressors). Each model is using default parameters and we choose model with either lookback window of 5, 10 or 20 timestamps. Otherwise it is lookback of size 1, which means treating previous day's reading as an IID point and discarding the temporal characteristics. The cross-validation strategy used here is $TimeSeriesTrainTestSplit$ (as shown in Fig. 3) with fraction of train, validation and test portions set to 0.8, 0.1, 0.1. The scoring method for evaluating the model performance in the validation set as well as test set is done using Mean Absolute Error (MAE). The summary of the results is provided in the Fig. 7. The _5, _10 and _20 indicate respective lookback window size.

We observed the top five models for each dataset based on the validation score. Since the difference among the top 5 models for each dataset was very small (MAE difference of around 0.01–0.1), we decided to evaluate them on the basis of the top 5 ranking models for a dataset. The observations made from this experiment are intriguing since they allow users to get a deeper understanding of the problem and their data. The main observations that can be made from this experiment are:

Model	Rank-1	Rank-2	Rank-3	Rank-4	Rank-5	Total
Wavenet_CNN_10	3	0	0	3	1	7
Simple_CNN_20	0	2	2	2	1	7
mlpregressor_10	1	0	1	3	2	7
deep_CNN_20	1	2	1	0	2	6
Wavenet_CNN_20	0	1	1	2	2	6
DNN_10	1	1	1	2	0	5
mlpregressor_20	1	3	1	0	0	5
SeriesNet_CNN_20	1	0	1	0	3	5
simple_LSTM_20	3	0	1	0	0	4
simple_LSTM_10	0	1	2	0	1	4
DNN	0	1	2	0	0	3
Wavenet_CNN_5	0	0	1	0	1	2
simple_LSTM_5	0	1	0	0	1	2
mlpregressor	2	0	0	0	0	2
Deep_LSTM_5	0	1	0	1	0	2
Simple_CNN_5	0	2	0	0	0	2
Simple_CNN_10	1	0	0	1	0	2
randomforestregressor_20	0	0	1	0	0	1
randomforestregressor_5	0	0	0	0	1	1
Deep_LSTM_20	0	0	0	1	0	1
SeriesNet_CNN_10	1	0	0	0	0	1
Deep_LSTM_10	0	0	0	0	0	0
randomforestregressor_10	0	0	0	0	0	0
mlpregressor_5	0	0	0	0	0	0
deep_CNN_10	0	0	0	0	0	0
DNN_5	0	0	0	0	0	0
SeriesNet_CNN_5	0	0	0	0	0	0
randomforestregressor	0	0	0	0	0	0

Fig. 7. Results of experiment 2: model-wise occurrence in top-5 ranks from cross-validation score

- Deep learning models outperform traditional machine learning models.
- It is seen that greater historical window size have higher rankings. The top 10 models in the rankings have window size of 20 or 10. The deep learning models with smaller lookback are at the bottom of the table with the reason being overfitting to local trends.
- It can be seen that with sufficient temporal information, even machine learning models can perform better than their IID counterparts.

5 Related Work

In the recent years, there have been many AI and ML frameworks proposed to relieve the pain points of data scientists. Frameworks like Scikit-learn [17], TensorFlow [1], Keras, PyTorch based on Torch [6], Theano [4], etc. provide API to define AI and ML experiments in python. Whereas cloud vendors like IBM [10], Amazon [2], Google [8], and Microsoft [15] provide infrastructure platforms and related APIs to execute machine learning jobs as a service on their respective platforms. These solutions provide many benefits to data scientists, however, they don't bridge the aforementioned gaps that are present for adoption of AI in Industries.

Our Solution, ThunderML, bridges these gaps by providing AI practitioners in the industry with a flexible and easy-to-use framework for defining high-level AI solutions built on top of other widely used open source frameworks like

Sklearn, Tensorflow, etc. It also provides a common interface for executing these AI solutions on different cloud vendors like IBM Cloud, Google Cloud, etc. with relative ease.

6 Conclusion

We have proposed ThunderML, a versatile toolkit for bridging fundamental gaps present in the adoption of AI in heavy industries. ThunderML aims to provide an easy yet flexible programming model which allows data scientists and AI practitioners to convert their unique problem statements into a purpose built and scalable AI pipelines quickly. It also helps practitioners leverage cloud-based environment for enabling AI pipelines with greater control using the two execution architectures. Finally, the case study attempts to provide evidence of the feasibility of ThunderML as a powerful and flexible toolkit which can prove vital in the hands of AI practitioners, as the time series prediction which forms the backbone of many industry solutions.

References

1. Abadi, M., et al.: TensorFlow: a system for large-scale machine learning. In: 12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 2016), pp. 265–283 (2016)
2. Amazon: Machine Learning on AWS. https://aws.amazon.com/machine-learning/
3. Ardagna, C.A., Bellandi, V., Ceravolo, P., Damiani, E., Bezzi, M., Hébert, C.: A model-driven methodology for big data analytics-as-a-service. In: IEEE International Congress on Big Data, BigData Congress, pp. 105–112 (2017)
4. Bergstra, J., et al.: Theano: a CPU and GPU math compiler in python. In: Proceedings 9th Python in Science Conference, vol. 1, pp. 3–10 (2010)
5. Cheng, Y., Hao, Z., Cai, R., Wen, W.: HPC2-ARS: an architecture for real-time analytic of big data streams. In: IEEE International Conference on Web Services, ICWS, pp. 319–322 (2018)
6. Collobert, R., Bengio, S., Mariéthoz, J.: Torch: a modular machine learning software library. Technical report, Technical report IDIAP-RR 02–46, IDIAP (2002)
7. Feurer, M., Klein, A., Eggensperger, K., Springenberg, J., Blum, M., Hutter, F.: Efficient and robust automated machine learning. In: Cortes, C., Lawrence, N.D., Lee, D.D., Sugiyama, M., Garnett, R. (eds.) Advances in Neural Information Processing Systems 28, pp. 2962–2970 (2015)
8. Google: Cloud AI Products. https://cloud.google.com/products/ai/
9. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Comput. **9**(8), 1735–1780 (1997)
10. IBM: Watson Machine Learning. https://www.ibm.com/cloud/machine-learning
11. Kristpapadopoulos, K.: SeriesNet (2018)
12. Krizhevsky, A., Sutskever, I., Hinton, G.E.: ImageNet classification with deep convolutional neural networks (2012)
13. LeCun, Y., Bengio, Y., Hinton, G.: Deep learning. Nature **521**(7553), 436 (2015)
14. Liao, Y., Deschamps, F., de reitas Rocha Loures, E., Ramos, L.F.P.: Past, present and future of industry 4.0 - a systematic literature review and research agenda proposal. Int. J. Prod. Res. **55**(12), 3609–3629 (2017)

15. Microsoft: Microsoft Azure Machine Learning Studio. https://azure.microsoft.com/en-us/services/machine-learning-studio/
16. Olson, R.S., Bartley, N., Urbanowicz, R.J., Moore, J.H.: Evaluation of a tree-based pipeline optimization tool for automating data science. In: Proceedings of the Genetic and Evolutionary Computation Conference 2016, GECCO 2016, pp. 485–492 (2016)
17. Pedregosa, F., et al.: Scikit-learn: machine learning in Python. J. Mach. Learn. Res. **12**(Oct), 2825–2830 (2011)
18. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., Salakhutdinov, R.: Dropout: a simple way to prevent neural networks from overfitting. J. Mach. Learn. Res. **15**(1), 1929–1958 (2014)
19. Van Den Oord, A., et al.: WaveNet: a generative model for raw audio. In: SSW, p. 125 (2016)
20. Zhang, P., Wang, H., Ding, B., Shang, S.: Cloud-based framework for scalable and real-time multi-robot SLAM. In: IEEE International Conference on Web Services, ICWS, pp. 147–154 (2018)