Orchestrating the Development Lifecycle of Machine Learning-Based IoT Applications: A Taxonomy and Survey

BIN QIAN, JIE SU, ZHENYU WEN*, DEVKI NANDAN JHA, YINHAO LI, YU GUAN, DEEPAK PUTHAL, and PHILIP JAMES, Newcastle University, UK
RENYU YANG, University of Leeds, UK
ALBERT Y. ZOMAYA, The University of Sydney, Australia
OMER RANA, Cardiff University, UK
LIZHE WANG, China University of Geoscience (Wuhan), China
MACIEJ KOUTNY and RAJIV RANJAN, Newcastle University, UK

Machine Learning (ML) and Internet of Things (IoT) are complementary advances: ML techniques unlock the potential of IoT with intelligence, and IoT applications increasingly feed data collected by sensors into ML models, thereby employing results to improve their business processes and services. Hence, orchestrating ML pipelines that encompass model training and implication involved in the holistic development lifecycle of an IoT application often leads to complex system integration. This paper provides a comprehensive and systematic survey of the development lifecycle of ML-based IoT applications. We outline the core roadmap and taxonomy, and subsequently assess and compare existing standard techniques used at individual stages.

Additional Key Words and Phrases: IoT, Machine learning, Deep learning, Orchestration

1 INTRODUCTION

Rapid development of hardware, software and communication technologies boosts the speed of connection of the physical world to the Internet via Internet of Things (IoT). A report [1] shows that about 75.44 billion IoT devices will be connected to the Internet by 2025. These devices generate a massive amount of data with various modalities. Processing and analyzing such big data is essential for developing smart IoT applications. Machine Learning (ML) plays a vital role in data intelligence which aims to understand and explore the real world. ML + IoT type applications thus are experiencing explosive growth. However, there are unfilled gaps between current solutions and the demands of orchestrating the development lifecycle of ML-based IoT applications. Existing orchestration frameworks for example *Ubuntu Juju*, *Puppet* and *Chef* are flexible in providing solutions for deploying and running applications over public or private clouds. These frameworks, however, neglect the heterogeneity of IoT environments that encompasses various hardwares, communication protocols and operating systems. More importantly, none of them are able to completely orchestrate a holistic development lifecycle of ML-based IoT applications. The development lifecycle must cover the following factors: 1) *how* the target application is specified and developed, 2) *where* the target application is deployed, (3) *what* kind of information the target application is being audited. Application specification defines the requirements including the ML tasks, performance, accuracy and execution workflow. Based on the specification and the available computing resources, the ML models are developed to meet the specified requirements while optimizing the training processes in terms of the cost of time and computing resources. Next, the model deployment considers the difficulty of

*Zhenyu is the corresponding author
[1]https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/

the heterogeneity of the IoT environment for running a set of composed ML models. Finally, ML-based IoT applications closely connect with people's lives and some applications such as autopilot require high reliability. Therefore, essential monitoring information has to be collected to improve the performance of the application in the next iteration of the lifecycle.

In this survey, we present comprehensive research on orchestrating the development lifecycle of ML-based IoT applications. We first present the core roadmap and taxonomy, and subsequently summarize, compare, and assess the variety of techniques used in each step of the lifecycle. Previous efforts provided broad knowledge that can drive us to build the taxonomy. For instance, [260] discussed encountered challenges of developing the next generation of AI systems. [197, 310] gave comprehensive reviews of available deep learning architectures and algorithms in IoT domain. To the best of our knowledge, this is the first work that presents a comprehensive survey to illustrate the whole development lifecycle of ML-based IoT application, which paves the way for developing an agile, robust and reliable smart IoT application. Before introducing the roadmap and taxonomy, we provide a smart city example in the next subsection that illustrates ML-based IoT applications in real-world.

1.1 Smart City Applications

Smart city uses modern communication and information techniques to monitor, integrate and analyze the data collected from core systems running across cities. Meanwhile, smart city makes intelligent responses to various use cases, such as traffic control, weather forecasting, industrial and commercial activities. Fig. 1 represents a smart city which consists of various IoT applications with many of them using Machine Learning (ML) techniques. For example, a *smart traffic*

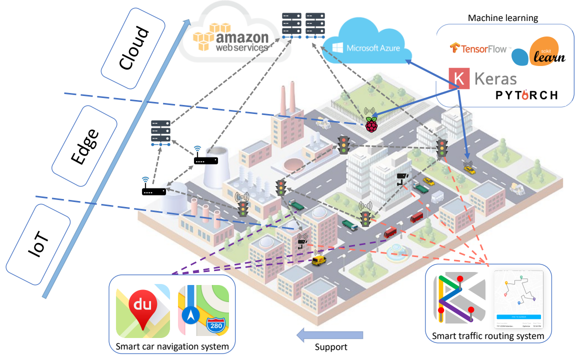

Fig. 1. Smart City

routing system consists of a large number of cameras monitoring the road traffic and a smart algorithm running on the cloud recommending the optimal routes for users [321]. On the other hand, a *smart car navigation system* [136] allows the passengers to set and change destinations via built-in car audio devices. The two systems work together to provide real-time interactive routing services. More specifically, the user's voice commands are translated in the car edge side and sent to the cloud where the *smart traffic routing* system works. The best route is translated back to voice guiding the users to their destinations. The above-mentioned applications involve various computing resources (e.g., cloud, edge, and IoT devices) and ML techniques, making the development of these ML-based IoT applications very challenging both for the ML models and the IoT system. To fill this gap, we orchestrate the development lifecycle of an ML-based IoT application. In the next subsection, we present a roadmap for the development lifecycle along with a comprehensive taxonomy that surveys the techniques relevant for developing the application.

1.2 Roadmap and Taxonomy

Roadmap. Fig 2 shows the roadmap of developing an ML-based IoT application. The roadmap starts with the requirements specification where the required computing resources (hardware and software) and ML models are specified. Based on the specification, we carefully design the infrastructure protocol, data acquisition approach and machine learning model development pipeline. Next, we implement and train the model with various ML algorithms. We also evaluate and optimize the models to achieve high efficiency without sacrificing too much accuracy. After the model development, an optimized deployment plan is generated based on the specified ML models and infrastructures. The deployed application must be audited while it is running on real IoT environments; the audit aims to explore the performance issues in terms of security, reliability and other QoS metrics. Finally, the audited issues will guide the corrections of orchestration details in the next iteration of the application development.

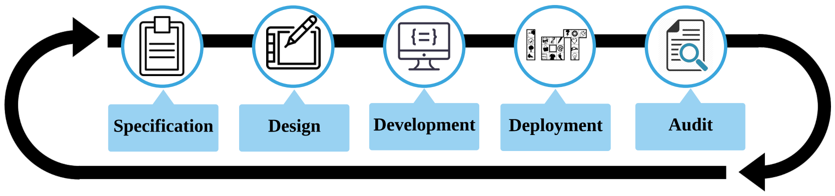

Fig. 2. The development lifecycle of an ML-based IoT application

Taxonomy. Fig. 4 depicts our taxonomy which systematically analyzes the core components in the orchestration of the development lifecycle of a ML-based IoT application. Note that the survey in [286] has reviewed cloud resource orchestration techniques. It outlines the key infrastructure orchestration challenges for cloud-based application as well as being extendable for IoT applications. Thus, in this survey, we focus more on the challenges of implementing ML models and orchestrating their IoT application development lifecycle. To this end, we extract the core building blocks of the development lifecycle relevant to ML and identify *four* main categories based on their specific functionality during the development process. The outline of the paper follows the structure of the taxonomy as well.

(1) **Model Development.** We propose a general pipeline for developing a ready-to-deploy ML model. We investigate the ML techniques to build each block of the pipeline (refer to §2).
(2) **Model Deployment.** In our work, we review the software deployment techniques and analyze the challenges of applying such techniques to deploy the ML models in IoT environments (refer to §3).
(3) **Model Audit.** Audit is one of the important dimensions in building a robust application. We survey the main security, reliability and performance issues in ML-based IoT applications (refer to §4).
(4) **Data Acquisition.** Data quality is important in building ML models. We identify three important dimensions throughout the data acquisition pipeline: data collection, data fusion and data preprocessing (refer to §5).

2 MODEL DEVELOPMENT

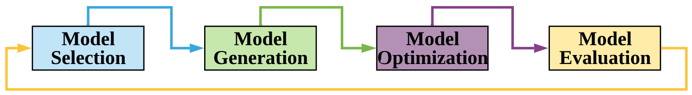

Fig. 3. A general pipeline of model development

One of the core components in this paper is machine learning (ML) models, which may be roughly divided into three categories: Traditional Machine Learning (TML), Deep Learning (DL) and Reinforcement Learning (RL). To develop ML models in the IoT environment, we propose a generic pipeline (see Fig. 3) including *model selection*, *model generation*, *model optimization* and *model evaluation* and explain with adaptive video streaming [183] as an example.

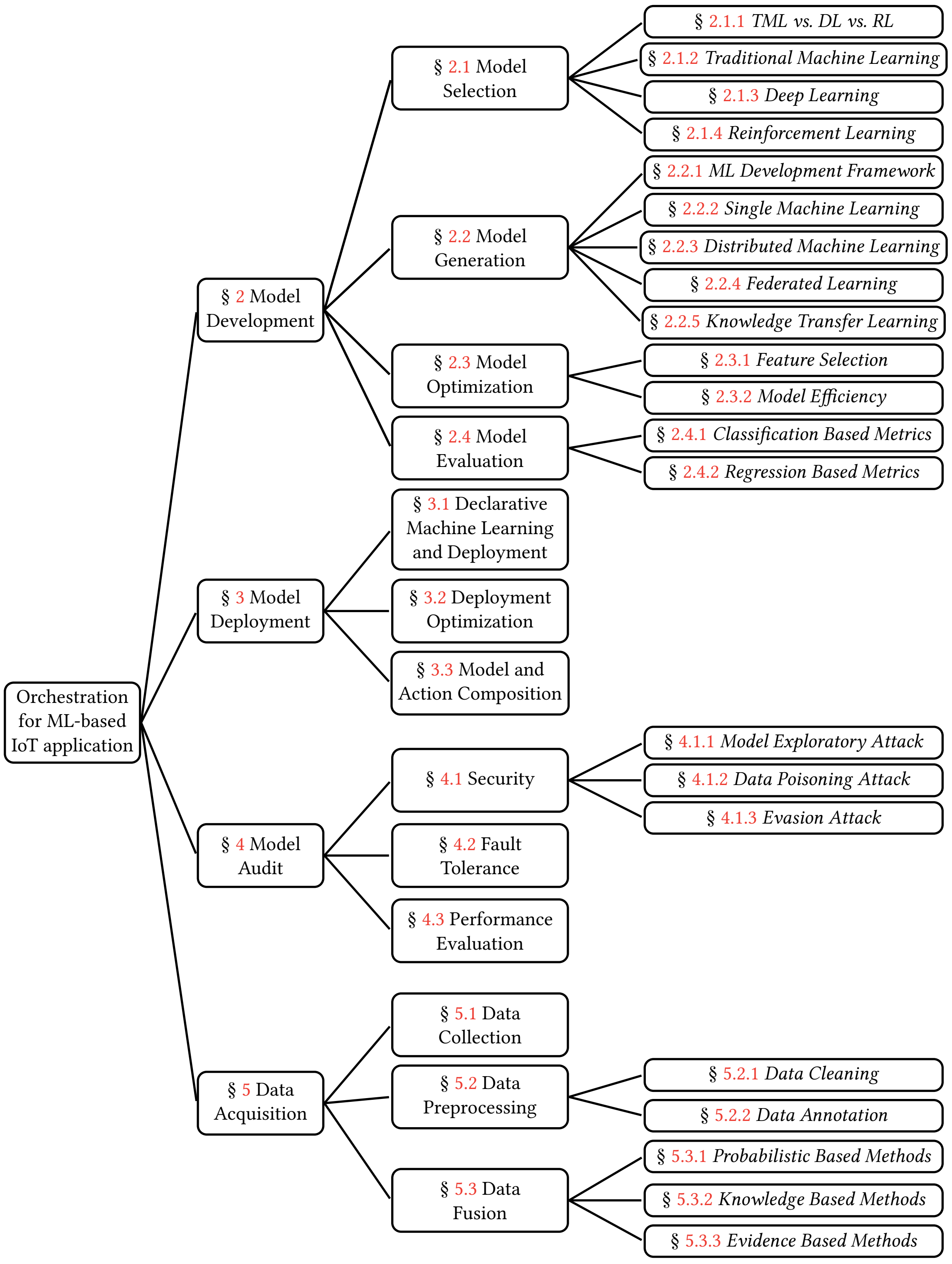
Orchestration for ML-based IoT application
§ 2 Model Development
§ 2.1 Model Selection
§ 2.1.1 *TML vs. DL vs. RL*
§ 2.1.2 *Traditional Machine Learning*
§ 2.1.3 *Deep Learning*
§ 2.1.4 *Reinforcement Learning*
§ 2.2 Model Generation
§ 2.2.1 *ML Development Framework*
§ 2.2.2 *Single Machine Learning*
§ 2.2.3 *Distributed Machine Learning*
§ 2.2.4 *Federated Learning*
§ 2.2.5 *Knowledge Transfer Learning*
§ 2.3 Model Optimization
§ 2.3.1 *Feature Selection*
§ 2.3.2 *Model Efficiency*
§ 2.4 Model Evaluation
§ 2.4.1 *Classification Based Metrics*
§ 2.4.2 *Regression Based Metrics*
§ 3 Model Deployment
§ 3.1 Declarative Machine Learning and Deployment
§ 3.2 Deployment Optimization
§ 3.3 Model and Action Composition
§ 4 Model Audit
§ 4.1 Security
§ 4.1.1 *Model Exploratory Attack*
§ 4.1.2 *Data Poisoning Attack*
§ 4.1.3 *Evasion Attack*
§ 4.2 Fault Tolerance
§ 4.3 Performance Evaluation
§ 5 Data Acquisition
§ 5.1 Data Collection
§ 5.2 Data Preprocessing
§ 5.2.1 *Data Cleaning*
§ 5.2.2 *Data Annotation*
§ 5.3 Data Fusion
§ 5.3.1 *Probabilistic Based Methods*
§ 5.3.2 *Knowledge Based Methods*
§ 5.3.3 *Evidence Based Methods*

Fig. 4. A taxonomy for orchestrating ML-based IoT application development lifecycle

Adaptive video streaming. Video transmission between server and mobile devices employs http-based adaptive streaming techniques. In a typical video server (e.g., DASH [2]), videos are encoded and stored as multiple chunks at different bitrates. One video usually consists of several chunks with each containing seconds of content. To maximize video quality, the video player in a client (e.g., mobile device) usually employs adaptive bitrate (ABR) algorithms aiming to pull high-bitrate chunks from the server without compromising the latency. As shown in Fig. 5, ABR algorithms use simple heuristics to make bitrate decisions based on various observations such as the estimated network throughput and playback buffer occupancy. ABR algorithms require fine-grained tuning and can be hardly generalized to handle

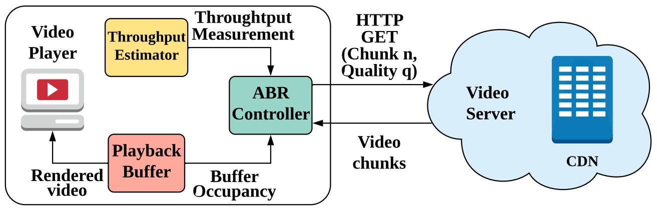

Fig. 5. Adaptive video streaming

various network conditions that fluctuate across time and different environments. Thus we are seeking to solve the problem using modern ML technologies.

To this end, we first need to perform **model selection** (§2.1) to find a subset of suitable models. In this scenario, the server must give a bitrate decision so that the client can return feedback that conveys whether the decision is satisfactory. Such interaction problems necessitate further use of RL and we will present proper choice of RL algorithms based on different selection criteria (§2.1.4). Next, we will choose a suitable development framework to implement the model and utilize different acceleration techniques to reduce the latency of **model generation** (§2.2). In this example, Tensorflow and A3C algorithm [196] are used as the development framework and distributed training protocol respectively for faster convergence. Once generated, the model has to be adapted into the real environment. Considering heterogeneity of IoT infrastructure, models need to be optimized according to the computing resources. This procedure is called **model optimization** (§2.3). In **model evaluation** (§2.4), model performance is measured to validate whether the model meets expected results. Particularly in this case, performance is evaluated by the total reward obtained from the simulated environment. The following subsections will discuss the the pipeline in detail.

2.1 Model Selection

Model selection aims to find the *optimal* ML model to perform user-specified tasks whilst adapting to the complexity of IoT environments. In this section, we discuss the model selection from three categories i.e., TML, DL and RL, and survey well-known models (or algorithms) in each category along with their corresponding criteria for model selection.

2.1.1 TML vs. DL vs. RL. Compared with the most popular DL , TML is relatively lightweight. It is a set of algorithms that directly transform the input data (to output), according to certain criteria. For supervised cases when a class label is available for training, TML aims to map the input data to the labels by optimising a model, which can be used to infer unseen data at the test stage. However, since the relationship between raw data and label might be highly non-linear, feature engineering— a heuristic trial-and-error process — is normally required to construct the appropriate input feature. The TML model is relatively simple, the interpretability (e.g., the relationship between the engineered features and the labels) tends to be high.

[2]https://github.com/Dash-Industry-Forum/dash.js

DL has become popular in recent years. With powerful capability for modelling complex non-linear relationships (between the input and output), DL does not require the aforementioned heuristic (and expensive) feature engineering process, making it a popular modelling approach in many fields such as computer vision and natural language processing. Compared with TML, DL models tend to have more parameters (to be estimated) and generally they require more data for reliable representation learning. However, it is crucial to guarantee the data quality and a recent empirical study[202] suggested the increasing number of noisy/less-representative training samples may harm DL's performance, making it less generalizable to unseen test data. Moreover, DL's multilayer structures make it difficult to interpret the complex relationship between input (i.e., raw features) and output. However, more and more visualisation techniques (e.g., attention map [313]) were used, which play an important role in understanding DL's decision-making process.

RL has become increasingly popular due to its success in addressing challenging sequential decision-making problems [265]. Some of these achievements are based on the combination of DL and RL, i.e., Deep Reinforcement Learning. It has shown its considerable performance in natural language processing [163, 296], computer vision [9, 48, 230, 264, 305], robotics [221] and IoT systems [182, 183, 320] and related applications like video games [9], visual tracking [230, 264, 305], action prediction [48], robotic grasping [221], question answering [296], dialogue generation [163], etc. In RL, there is usually one or more agent(s) interacting with the outside environment, where optimal control policies are learnt through experience. Fig. 6 illustrates the iterative interaction circle, where the agent starts without knowing anything about environment or task. Each time the agent takes action based on the environment states, and it receives a reward from the environment. RL optimises this process such that it learns to make decisions with higher rewards received.

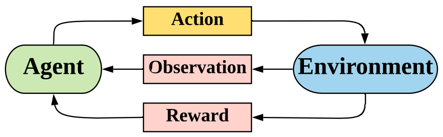

Fig. 6. Reinforcement Learning Paradigm

Discussion. In IoT environments, a variety of problems can be modelled by using the aforementioned three approaches. The applications range from system and networking [183] [182], smart city [320] [164], to smart grid [290] [235], etc. To begin with modeling, it is essential for users to choose a suitable learning concept at the first stage. The main selection criteria can be divided into two categories: *Function-based selection* and *Power Consumption-based selection.*

Function-based selection aims to choose an appropriate concept based on their functional difference. For example, RL benefits from its iterative environment ↔ agent interaction property, and can be applied to various applications which need interaction with environment or system such as smart temperature control systems, or recommendation systems (with cold start problem). On the other hand, TML algorithms are more suitable for modelling structured data (with high-level semantic attributes), especially when interpretability is required. DL models are typically used to model complex unstructured data, e.g., images, audios, time-series data, etc. and are an ideal choice especially with high amount of training data and low requirement on interpretability.

Power Consumption-based selection aims to choose an appropriate model given constraints in computational power or latency. In contrast to TML, the powerful RL/DL models are normally computationally expensive with high overhead. Model compression techniques were developed, making RL/DL models for efficient some IoT applications. However, on some mobile platforms with very limited hardware resources (e.g., power, memory, storage), it is still challenging to employ compressed RL/DL models, especially when there are some performance requirements (e.g., accuracy, or real-time inference) [51]. On the other hand, lightweight TML may be more efficient, yet reasonable accuracy can only be achieved with appropriate features (e.g., high level attributes derived from the time-consuming feature engineering).

2.1.2 *Traditional Machine Learning.*

Given different tasks, TML can be further divided into *Supervised Learning* and *Unsupervised Learning*. Herein we contrast two categories (algorithm details are available in TML method **Appendix B**), and discuss the criteria for choosing the TML algorithms.

Supervised Learning. Supervised learning algorithm (i.e., Fig. 7) can be used when both the input data X and the corresponding labels Y are provided (for training), and it aims to learn a mapping function such that $Y :\longleftarrow f(X)$. The

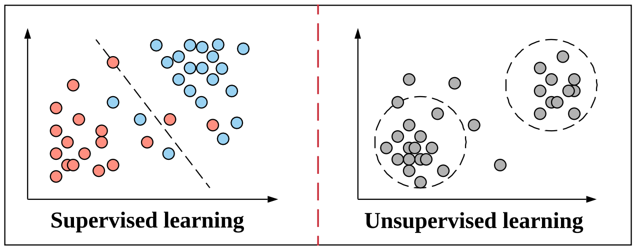

Fig. 7. Examples of Supervised Learning (Linear Regression) and Unsupervised Learning (Clustering)

most representative algorithms include *Logistic Regression (LR)*, *Artificial Neural Networks (ANN)*, *Decision Tree (DT)* [222] and *Random Forest (RF)* [36], *Support Vector Machine (SVM)* [63]. The TML algorithms are based on mathematics and statistics modelling which give more interpretability for model itself.

Unsupervised Learning. The unsupervised learning algorithm (see Fig. 7 right) aims at learning the inherent relationship between the data when only input data X exists (without class label Y). For example, the clustering algorithm can be used to find the potential patterns of some unlabelled data and the obtained results can be used for future analysis. *K-Means*[116] and *Principal Component Analysis (PCA)* [246] are the two most popular unsupervised learning algorithms. *K-means* aims to find K group patterns from data by iteratively assigning each sample to different clusters based on the distance between the sample and the centroid of each cluster. *PCA* is normally used for dimensionality reduction, which can de-correlate the raw features before selecting the most informative ones.

Discussion. A common principle for IoT application is to select the algorithm with the highest performance in terms of effectiveness and efficiency. One can run all related algorithms (e.g., supervised, or unsupervised), before selecting the most appropriate one. For effectiveness, one has to define the most suitable evaluation metrics, which can be task-dependent, e.g., accuracy or mean-f1 score for classification tasks, or mean squared errors for regression, etc. Before model selection, a number of factors should be taken into account: data structure (structured data, or unstructured data that need extra preprocessing), data size (small or large), prior knowledge (e.g., class distribution), data separability (linearly, or non-linearly separable which may require additional feature engineering), dimensionality (low, or high which may require dimensionality reduction), etc. There may also exist additional requirements from the users/stakeholders, e.g., interpretability for health diagnosis. Additionally, it is necessary to understand the efficiency requirement specific to an IoT application and one has to consider how the training/testing time grows with respect to data size. Time complexity shown in Table 2 in **Appendix B** provides more insights. Take *KNN* as an example: although no training time is taken, *KNN*'s inference time can be very high (especially with a large training set), and thus unsuitable for certain time-critical IoT applications. Also, the deployment environment is another non-negligible factor when developing IoT applications since many applications run (or partially run) on low power computing resources.

2.1.3 *Deep Learning.*

In this section, we primarily introduce three classical deep models (i.e., *Deep Neural Networks* (DNN)/ *Multilayer Perceptron* (MLP), *Convolutional Neural Networks* (CNN) and *Recurrent Neural Networks* (RNN)) for supervised learning tasks on unstructured data such as image, video, text, time-series data, etc. We also brief two popular unsupervised models: *Autoencoder* (AE), and *Generative Adversarial Networks* (GAN).

Supervised DL. We contrast the features of three basic supervised DL models: DNN, CNN and RNN.

Deep Neural Networks (DNN). As previously mentioned, a deep neural network (*DNN*) is an *ANN* with more than one hidden layer, and hence it is also called multilayer perceptron (*MLP*). Compared with *ANN* with a single hidden layer, *DNN* has more powerful modelling capabilities and its deep structure makes it easier for it to learn higher-level semantic features, which is crucial for classification tasks on complex data. However, for high-dimensional unstructured input data (such as images), there may be many model parameters to be estimated, and in this case, overfitting may occur if there is not enough labelled data. Nevertheless, generally *DNN* has decent performance when input dimensionality is not extremely high, and it has been successfully applied to various applications, for example human action recognition [274], traffic congestion prediction[75] and healthcare[203].

Convolutional Neural Network (CNN). When it comes to high-dimensional unstructured data such as images, in visual recognition tasks it is hard to directly map the raw image pixels into target labels due to the complex non-linear relationship. The traditional way is to perform feature engineering, which is normally a trial-and-error process, and may require domain knowledge in certain circumstances, before TML is applied. This heuristic approach is normally time-consuming, and there exist substantial recognition errors even in simple tasks since it is very challenging to hand-engineer the high-level semantic features. *CNN*, a deep neural network with convolutional layers and pooling layers, can address this issue effectively. The convolution operation can extract the higher level features while the pooling operation can keep the most informative responses and reduce the dimensionality. Compared with *DNN*, the weight sharing concept (of the convolution operation) enables *CNN* to capture the local pattern without suffering from the "curse of high-dimensionality" from the input. These operations and the hierarchical nature make *CNN* a powerful tool for extracting high-level semantic representations from raw image pixels directly, and successfully applied to various recognition tasks such as object recognition, image segmentation [89] and object detection [117]. Because of the decent performance on various visual analysis tasks, *CNN* is usually considered as the first choice for some camera-based IoT applications, for example traffic sign detection [248].

Recurrent Neural Networks (RNN). Nowadays, with the increasing amount of generated stream and sequential data from various sensors, time series analysis has become popular among the machine learning (ML) community. *RNN* is a sequential modelling technique that can effectively combine the temporal information and current signal into the hidden units for time-series classification/prediction. An improved *RNN* named Long Short Term Memory (LSTM) [122], including complex gates and memory cells within the hidden units for "better memories", became popular in various applications such as speech recognition [104], video analysis [273], language translation [177], activity recognition [108] etc. Since data streaming is most common in the IoT environment, RNN (LSTM) is deemed as one of the most powerful modelling techniques, and there are various IoT applications such as smart assistant [91, 274], *smart car navigator system* [136], malware threat hunting [111], network traffic forecasting [225], equipment condition forecasting [315], energy demand prediction system [200], load forecasting [152], etc.

Unsupervised DL. Two unsupervised DL models are hereby introduced:*Autoencoder (AE)* [15] and *Generative Adversarial Network (GAN)* [100]. Without requiring any label information, *AE* can extract compact features and reconstruct original (high-dimensional) data with the extracted features. It is normally used for dimensionality reduction, latent distribution analysis or outlier detection. *GAN*, on the other hand, applies an adversarial process to learn the "real" distribution from the input data. More precisely, *GAN* consists of two parts, namely generator and discriminator. The generator aims at generating indistinguishable samples compared to the real data. While the discriminator works adversarially to distinguish the generated fake samples from the real data. With iterative competition process, GAN will eventually reach to a state where the generated samples are indistinguishable from the real data. The learnt "real" distribution can be used to generate samples for different purposes. Both *AE* and *GAN* are powerful and promising tools for computer vision as well as IoT applications. *AE* can be used for diagnosis/fault detection tasks [57, 205] or simply as a preprocessing tool (i.e., feature extraction/dimensionality reduction). *GAN* has been used for studies on generating rare category samples, and this upsampling approach may further improve the model performance [318, 319].

Discussion. The aforementioned DL models can be effective tools for processing different unstructured data types. The way of applying them is generally very flexible, and they can be used jointly to process the complex data from various sources in the IoT environments. For example, although *CNN*/*RNN* could be used in an end-to-end manner (e.g., as image/time-series classifiers), they could also be used as feature extractors, based on which one can easily aggregate features extracted from different sources (e.g., audio, images, sensor data). With high-dimensional video data, one can either model by training *CNN*+ *LSTM* jointly [278], or use *CNN*/*AE* as feature extractors, before the sequential modelling (e.g., using *LSTM*). However, when modelling the data with limited labels (e.g., rare event), one needs to consider the potential overfitting effect when using DL directly. One may go back to the TML approaches or use some upsampling techniques (e.g., *GAN*) to alleviate this effect.

2.1.4 *Reinforcement Learning (RL).*

In this section, we first introduce the strategies used to formulate the aforementioned video streaming example (see §2) with Reinforcement Learning (RL). As mentioned earlier, in RL an *agent* interacts with the *environment*, learning an optimal control policy through experience. It requires three key elements, *observation*, *action*, and *reward*. Based on these, we can formulate the adaptive bitrate streaming problem. Specifically, *observation* can be the buffer occupancy, network throughput, etc. At each step, the *agent* decides the bitrate of the next chunk. A *reward* (for example the quality of service feedback from the user) is received after the *agent* takes *action* (chunk bitrate). The algorithm proposed in [183] collects and generalizes the results of performing the past decisions and optimizes its policy from different network conditions. This RL-based algorithm can also make the system robust to various environmental noises such as unseen network conditions, video properties, etc.

As shown in Fig. 8, there is a plethora of algorithms in the whole reinforcement learning family. More details of these RL algorithms can be found in the **RL methods in Appendix B**, and here we focus on selecting appropriate RL algorithms based on different selection criteria.

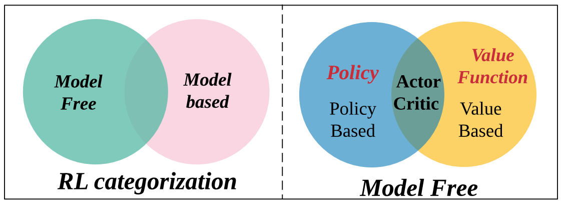

Fig. 8. Reinforcement Learning Categorization

Environment Modelling Cost In RL modelling, sample efficiency is one of the major challenges. Normally the RL agent can interact either with the real world or a simulated environment during training. However, it can be difficult to simulate the heterogeneous IoT environments and complex IoT devices. RL models can also be trained directly in real world IoT environments, yet one major limitation is the heavy training cost, which may range from seconds to minutes for each step. The model-based RL method, a method that can reduce the sample complexity, can decrease the training time significantly. It first learns a predictive model of the real world, based on which the decisions can be made. When compared with model-free approaches, model-based methods are still in their infancy, and because of the efficiency property, they may attract more attention in the near future.

Action Space: The action space of RL algorithms can be either continuous or discrete. For those RL algorithms with discrete action space, they choose from a finite number of actions at runtime. Take the video streaming task for example, the action space is different bitrates for each chunk. Another task formulated in discrete action space can be found in [182], where the action space is the "schedule of the job at i-th slot". Available algorithms for discrete action space tasks most reside in the policy gradient group, for example DQN, DDQN. The continuous action space, on the other hand, is infinite for all possible actions. Relationships exist between the actions that are usually sampled from certain distributions such as Gaussian distribution. For example, in an energy-harvesting management system, PPO algorithm [238] is used to control IoT nodes for power allocation. The action space, as stated in [201], is sampled from a Gaussian distribution to denote the load of each node ranging from 0% to 100%. Similarly, in another work [7] that studied energy harvesting WSNs, the Actor-Critic [150] algorithm is implemented to control the packet rate during transmission. One advantage of continuous action space lies in its ability to accurately control the system, thus a higher QoE is expected.

2.2 Model Generation

Based on the user requirement and task specification, we have selected a variety of models. Next, the models need to be developed and implemented. In this section, we will introduce the available tools for the development. We will also present the approaches that can be utilized to accelerate the training process.

2.2.1 Machine Learning Development Framework. The training and execution of ML models can be tricky and it may require numerous engineering efforts. Efforts have been devoted to developing frameworks to support the model development. These frameworks have their own strengths and weaknesses in terms of the supported models, usability, scalability, etc. In this section, we will review several development frameworks.

For TML, the most famous development framework is Sci-kit learn. It is a free ML library with Python interface. Sci-kit learn supports almost all main-stream machine learning models and is a popular tool for fast prototyping. For DL, we list some of the most popular DL frameworks and discuss their pros and cons in Table 1. Users can choose the most suitable frameworks based on their needs.

DL frameworks	Core language	Interface	Pros	Cons
Tensorflow (2)	C++	Python, Javascript, C++, Java, Go	- Effective data visualization - Distributed learning - Efficient model serving - On-device inference with low latency for mobile devices - Eager Execution with TF2, easy to debug	- Steep learning curve (migration from TF 1 to TF 2) - Poor results for speed
Pytorch	C/C++	Python, C++	- Simple and transparent modeling - Eager execution	- Hard to serve even with ONNX support
Caffe (2)	C++	Python, C++	- Fast, scalable, and lightweight - Server optimized inference	- Limited community support - Limited in implementing complex networks
Mxnet	C++	Python, C++, Java, Julia, R, Perl, Clojure	- Fast, flexible, and efficient in terms of running DL algorithms - Run on any device - Easy model serving - Highly scalable	- Smaller community compared with Tensorflow or Pytorch
DL4J	Java	Java, Clojure, Kotlin	- Robust, flexible and effective - Works with Apache Hadoop and Spark	- Robust, flexible and effective - Works with Apache Hadoop and Spark

Table 1. Comparison of Deep Learning Frameworks

When IoT comes into context, more challenges arise with edge computing as it is trying to move the computation close where the data is generated [245]. The device heterogeneity of edge computing has made the development of the DL models more complicated. There are many portable Edge computing devices, each optimized with different inference engines. For example Nvidia Jetson series GPU computing unit compiles models with TensorRT inference engine while TensorFlow Lite is specially optimized for Google coral TPU. These inference engines optimize the model graph and quantize the model parameters to lower precision, thus delivering low latency and high-throughput for on-device inference. Some attempts [269] have been made to integrate both inference engines but the compatibility issue still exists. TVM [49] breaks the boundaries among diverse hardware, aiming at cross-framework and cross-device end-to-end optimization of DL models.

2.2.2 Single Machine Learning (Centralized). Model training via single machine is a common strategy for ML model generation. By placing the learning-related computation in the same place, the model learns from the data and updates its parameters. In this subsection, we highlight two approaches that leverage hardware for the training process acceleration: *Computation Optimization*, *Algorithm Optmization.*

Computation Optimization The basic computation unit of neural networks consists of vector-vector, vector-matrix and matrix-matrix operations. Efficient implementation of computations can accelerate the training and inference process. The Basic Linear Algebra Subprogram (BLAS) standardizes the building blocks for basic vector, matrix operations. A higher level linear algebra library such as cuBLAS implements BLAS on top of NVIDIA CUDA and is efficient in utilizing the GPU computation resource. Intel Math Kernel Library (MKL) on the other hand, maximizes performance on Intel processors and is compatible with BLAS without the change of code.

Different DL architectures (e.g., DNNs, CNNs and RNNs) may require different optimizations in terms of basic computations. The DNN computation is usually basic matrix-matrix multiplication and the aforementioned BLAS libraries can efficiently accelerate the computations with GPU resources. The CNNs and RNNs are different in their convolution and recurrent computations. Convolutions can not fully utilize the multi-core processors and the acceleration can be achieved by unrolling the convolution [45] to matrix-matrix computation or computing convolutions as point-wise product [185]. For RNN (LSTM), the complex gate structures and consecutive recurrent layers differ from the DNNs and CNNs in that these computation units can not be split and deployed directly at different devices. This has made parallel computation difficult to apply. Optimization is possible though, with implementations on top of NVIDIA cuDNN [52]. Computations among the same gates can be grouped into larger matrix operations [8] and save intermediate steps. We can also accelerate by caching RNN units' weights with the GPU's inverted memory hierarchy [76]. The weights are reusable between time steps, making a maximum 30× speed up on a TitanX GPU.

Algorithm Optimization Apart from the resource utilization optimization, the algorithmic level optimization is another important research direction for efficient model training and faster convergence. Optimization algorithms aim at minimizing/maximizing a loss function that varies for different machine learning tasks. They can be divided into two categories: *First Order Optimization* and *Second Order Optimization.*

First Order Optimization methods minimizing/maximizing the loss function with the gradient values with respect to the model parameters. Gradient Descent is one of the most important algorithms for neural networks. After back-propagation from the loss function, the model parameters are updated towards the opposite direction of the gradient. Gradient descent approaches fall into local optima when the absolute value is either too big or too small. Also it updates the gradient of the whole data set at one time, memory limitation is always a big problem. Variants have been proposed to address the aforementioned problems, including Stochastic gradient descent [32], mini-batch gradient descent [73]. Also, much famous research enables faster model convergence: Momentum [218], AdaGrad [82], RMSProp [119], ADAM [148]. *Second Order Optimization* methods take second order derivative for minimizing/maximizing loss function. Compared to the *First Order Optimization*, it consumes more computation power and is less popular for machine learning model training. However, *Second Order Optimization* considers the surface curvature performance and is less likely to get stuck on saddle points. Thus it sometimes outperforms the *First Order Optimization*. Famous *Second Order Optimization* methods include [12, 40, 118, 198, 206]. For more systematic survey on the optimization methods for machine learning training, one can refer to [33].

2.2.3 *Distributed Machine Learning.*

Modern ML models such as neural networks require a substantial amount of data for the training process. These data are usually aggregated and stored in the cloud server where training happens. However, when the training process of large volume data outpaces the computing power of a single machine, we need to leverage multiple machines available in the server cluster. This requires the development of novel distributed ML systems and parallel training mechanisms which distribute and accelerate the machine learning workload.

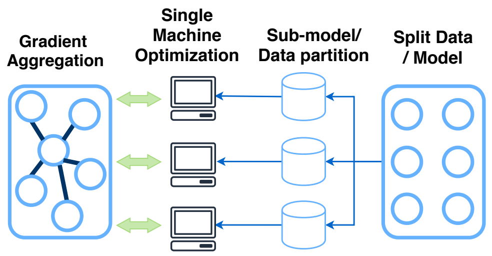

Fig. 9. Distributed Machine Learning Pipeline

Fig 9 shows the schematic diagram of a distributed ML pipeline. It has multiple components which are engaged in *Training Concurrency*, *Single Machine Optimization*, and *Distributed System*. In *Training Concurrency*, either the models or the data are split into small chunks and placed on different devices. With *Single Machine Optimization* (which shares similar techniques as conventional ML, see § 2.2.2) that accelerates the training process, we get all local gradient updates. Finally, *Distributed System* discusses strategies that efficiently aggregate the gradient updates.

Training Concurrency in Distributed ML. In the distributed machine learning, the selection of parallel strategy depends on two factors: data size and model size. When either the datasets or the model parameters are too big for single-machine processing, it is straightforward to consider partitioning them into smaller chunks for processing at different places. Here we first introduce two basic methods *data parallel*, *model parallel*. We also introduce *pipeline parallel* and other hybrid approaches that take advantage of both approaches.

Data Parallel. In a multi-core system where a single core can not store all the data, data parallel is considered by either splitting the data samples or the features. Data parallel has been successfully applied to numerous machine learning algorithms [58] with each core working independently on a subset of data. It can be used for training ML algorithm for example decision trees and other linear models where the features are relatively independent. Parallel with the split of data features, though, it can not be used directly with neural networks because different dimensions of the features are highly correlated.

In deep learning, data parallel works by distributing the training dataset across different GPU units. The dominant data parallel approach is *batch parallelism* where mini-batch SGD is employed to compute local gradient updates on a subset of the data. A central server is responsible for aggregating all local updates to global parameter and pushing new models back to the working units. One of the earliest works trained with GPUs can be found in [224] where the authors implemented distributed mini-batch SGD unsupervised learning concurrently with thousands of threads in a single GPU. By varying the batch size [102, 255, 302], this method is effective in reducing the communication cost without too much accuracy loss. In the next paragraph, we will discuss more about the parallel SGD algorithms [187, 259, 304, 312, 325] for improving the communication efficiency, which can be seen as one way of improving the performance of data parallelism. Another type of data parallel that addresses the memory limit on single GPU is *spatial parallelism* [141]. *Spatial parallelism* considers partitioning spatial tensors into smaller subdivisions and allocating them to separate processing units. It thus differs from *batch parallelism* in that the latter puts the groups of data in the same process. *Spatial parallelism* approach has proven to show near linear speedup on modern multi-GPU systems.

Model Parallel. Data parallel suffers from the infeasibility of dealing with very large models especially when it exceeds the capacity of a single node. Model parallel addresses this problem by splitting the model with only a subset of the whole model running on each node [35, 72, 147, 160]. The computation graphs can be divided within the layers (horizontal) or across the layers (vertical). Mesh-tensorflow [244] allows linear within-layer scaling of model parameters across multiple devices after compiling a computation graph into a SPMD program. However, this approach requires high communication cost as it needs to split and combine model updates across a large number of units. [134] introduced decoupled parallel backpropagation to break the sequential limitation of the back-propagation between the nodes, greatly increasing the training speed without much accuracy loss. For CNN, as each layer can be specified as five dimensions including: samples, height, width, channels, and filters, existing literature [79, 80] studies the split of models among dimensions. Another research direction that optimizes the communication overhead is by searching the optimal partition and device placement of computation graphs via reinforcement learning [193]. The literature [140, 284] followed this idea and shows interest in automatic search of optimal parallel strategies.

Pipeline Parallel. Although model parallel has proven successful in training extremely large models, the implementation is complicated due to the complexity of the neural network structure. This is especially true for CNNs since the convolution operators are highly correlated. Also, GPU utilization is low for model parallel. Due to the gradient interdependence between different partitions, usually only one GPU is in use each time. To solve the aforementioned problems, pipelining has been studied [213, 293] for speeding up the model training. With pipeline parallel, models are partitioned and displayed across different GPUs. Then mini-batches of training data are injected to the pipeline for concurrent processing of different inputs at the same time. Fewer worker GPUs are idle in the pipeline parallel setting as each node is allocated jobs, without waiting for other nodes to finish their work. According to the synchronization strategy we discussed earlier, gradients are aggregated by either synchronous pipeline model (GPipe [132]) or asynchronous pipeline model (PipeDream [115], SpecTrain [46], XPipe [107]). Theoretical analysis of pipeline parallel optimzation has also been studied and with Pipeline Parallel Random Smoothing (PPRS) [60], convergence rates can be further accelerated.

Hybrid. Data and model parallel are not mutually exclusive. Hybrid approaches that combine the benefits of both methods are effective in further accelerating the training process. Pipeline parallel [115, 132] can be seen as an approach built on top of data parallel and model parallel. Apart from that, [154] proposed combining data parallel and model parallel for different types of operators. With data parallel for CNN layers and model parallel for DNN layers, it achieved a 6.25× speed up with only 1% of accuracy loss on eight GPUs. Another implementation MAPS-Multi [18] borrows the idea of [154] and automates the partitioning of workload among multiple GPUs, achieving 3.12× speed up on four GTX 780 GPUs. Other forms of data parallel and model parallel hybrids exist in the literature [53, 72, 95, 96] that reduce the overall communication and computation overhead.

Distributed ML System. When we have acquired a local model update with partial data slice, multi-node and multi-thread collaboration are important for effectively updating the model. Network communication plays an important role in sharing the information across the nodes. In this section, we present the three most important features in network communication: 1) network topology, 2) synchronization strategy and 3) communication efficiency.

Network Topology. The network topology defines the node connection approach in the distributed machine learning system. When the data and models are relatively simple, it is common to utilize existing Message Passing Interface (MPI) or MapReduce infrastructure for the training. Later when the systems are becoming more and more complex, new topologies should be designed to facilitate the parameter update.

The Iterative MapReduce (IMR) or AllReduce approaches are commonly used for synchronous data parallel training. Typical IMR engines (for example the Spark MLlib [190]) generalizes MapReduce and enables the iterative training required by most ML algorithms. Synchronous training can also be implemented by AllReduce topology. MPI (Message Passing Interface) supports AllReduce and is efficient for CPU-CPU communication. Many researchers implement their own version of AllReduce for example Caffe2 Gloo, Baidu Ring AllReduce. In the ring-Allreduce topology, all nodes connect to each other without a central server, just like a ring. The training gradients are aggregated through their neighbors on the ring. To provide more efficient communication for DL workload in the GPU cluster, libraries such as Nvidia NCCL [62] are developed and support the AllReduce topology. In NCCL2 [138], the multi-node distribution feature is also introduced. Horovod [240] replaces the Baidu ring-Allreduce backend with NCCL2 for efficient distribution.

A Parameter Server (PS) infrastructure [166] is usually composed of a set of worker nodes and a server node which gathers and distributes computation from worker nodes. As asynchronous training of PS neglects stragglers, it provides better fault tolerance capability when some of the nodes break down. Parameter server also features high scalability and flexibility. Users can add nodes to the cluster without restarting the cluster. Famous projects such as DMTK Microsoft Multiverso [84], Petuum [295] and DistBelief [72] enable training of even larger networks.

Synchronization Strategy. In distributed ML, model parameter synchronization between worker nodes is cost-extensive. The trade-off between the communication and the fresher updates has great impact on the parallelism efficiency.

Bulk Synchronous Parallel (BSP) [186] is the simplest strategy for ensuring model consistency of all worker nodes. For each training iteration, all nodes wait for the last (slowest) node to finish the computation and the next iteration does not start before the all the model updates are aggregated. Total Asynchronous Parallel (TAP) [72] approaches are proposed to address the problem of the stragglers within the network. With TAP, all worker nodes access the global model via a shared memory. They can pull and update global model parameters any time when the training is finished. As there is no update barrier for this approach, the system fault tolerance is greatly improved. However, stale model updates can not guarantee convergence to global optimum. Many famous frameworks use the TAP strategy, including Hogwild! [229] and Cyclades [207].

Stale Synchronous Parallel (SSP) [121] compromises between fully-synchronous and asynchronous schemes. It allows a maximum staleness by allowing faster working nodes to read global parameters without waiting for slower nodes. As a result, the workers spend more time doing valuable computation, thereby improving the training speed greatly. But when there is too much staleness within the system, the convergence speed can be significantly reduced. Many state-of-the-art distributed training systems implement BSP and SSP for efficient parallelism, for example tensorflow [1], Geeps [67], Petuum [295].

In contrast to the SSP which limits the staleness of the model update, the Approximate Synchronous Parallel [127] (ASP) limits the correctness. In Gaia [127], for each local model updates, the global parameter is aggregated only if the parameter change exceeds a predefined threshold. This “significance" only strategy eliminates unnecessary model update and is efficient in utilizing the limited bandwidth. However, the empirical determination of threshold only considers the network traffic and is insufficient for dealing with dynamics in the IoT environment. [285] has addressed this problem by also considering resource constraints for efficient parallelism.

Communication Efficiency. Communication overhead is the key and often the bottleneck in distributed machine learning [167]. The sequential optimization algorithms implemented in the worker nodes require frequent read and write from the global shared parameters which poses great challenge on balancing network bandwidth and communication frequency. To increase the communication efficiency, we can either reduce the size of the model gradient (communication content) or the communication frequency.

Communication content. The gradient size between working nodes is correlated to both the model size itself and the gradient compression rate. We have reviewed four types of *model compression* techniques in §2.3.2 which are effective in reducing the overall gradient size. Hereby we focus on the techniques that compress the gradient before transmission, discusses the gradient *quantization* and *sparsification*.

Gradient quantization differs from the weight quantization (§2.3.2) as the former compresses the gradient transmission between worker nodes while the latter focuses on faster inference via smaller model size. Works that reduce the gradient precision [71] have been proposed and 1-bit quantization [239, 261] is effective in greatly reducing the computation overhead. Based on the idea, QSGD [6] and Terngrad [288] consider stochastic quantization where gradients are randomly rounded to lower precision. Additionally, weight quantization and gradient quantization can also be combined [125, 133, 292, 311, 323] for efficient on device acceleration.

The weights of the DNNs are usually sparse and due to the large number of unchanged weights in each iteration, the gradient updates are even more sparse. This sparsification nature of the gradient transmission has been utilized for more efficient communication. Gradient sparsification works by sending only important gradients when exceeding a fixed threshold [261] or adaptive threshold [81]. Gradient Dropping [3] uses layer normalization to keep the convergence speed. DGC [171] uses local gradient clipping for sending important gradients first while the less important ones are aggregated with momentum correction for later transmission.

Communication Frequency. Local (Parallel) SGD [187, 259, 304, 312, 325] entails performing local updates several times before parameter aggregation. Motivated by reducing the inter-node communication, this approach is also called model averaging. One-shot averaging [187, 325] considers only one aggregation during the whole training process. While [312] argues that one-shot averaging can cause inaccuracy and proposes more frequent communications, many works [170, 216, 303, 314] prove the applicability of the model averaging approach in various deep learning applications. In an asynchronous setting, the communication frequency can also be maneuvered through the push and pull operations in the worker nodes. DistBelief [72] has adopted this approach with a larger push interval compared to the pull interval.

2.2.4 Federated Learning. In traditional distributed machine learning, the training usually happens on the cloud data center with aggregated training data generated by collecting, labelling and shuffling raw data. The training data is thus considered *identical and independent distributed (IID)* and balanced. This facilitates the training process as one only needs to consider distributing the training task across various computation units and updating the model by aggregating all local gradient updates. However, this is not the case when IoT comes into play. The ML-based IoT applications differ from the traditional ML applications in that they usually generate data from heterogeneous geo-distributed devices (e.g., user behavior data from mobile phones). These data can be privacy-sensitive as users usually prefer not to leak personal information, making conventional distributed ML algorithms infeasible for solving such problems. Thus novel optimziation techniques are required to enable training in such scenarios.

Federated learning (FL) [151] is a type of distributed machine learning research that moves the training close to the distributed IoT devices. It learns a global model by aggregating local gradient updates and does not require the movement of the raw data to the cloud center. FederatedAveraging (FedAvg) [188] is a decentralized learning algorithm specifically designed for the FL. It implements synchronous local SGD [47] on each device with a global server averaging over a fraction of all the model updates per iteration. FedAvg is capable of training high-accuracy models on various datasets with many fewer communication rounds. Following this work, [151] proposed two approaches: *Structured updates* and *sketched updates* for reducing the communication cost, achieving higher communication efficiency. Further research addresses the privacy limitation of FL by Differential Privacy [189] and Secure Aggregation [30]. Finally, [29] delivers system-level implementation of FL based on previously mentioned techniques. It is able to train deep learning models with local data stored on mobile phones.

FL is still developing rapidly with many challenges remaining to be solved. On the one hand, FL shares similar challenges as in conventional distributed machine learning methods in terms of more efficient communication protocol, synchronization strategy as well as parallel optimziation algorithms. On the other hand, the distinct setting of FL requires more research preserving the privacy of training data, ensuring the fairness and addressing bias in the data. For a more thorough survey on details of FL, one can refer to [144].

2.2.5 Knowledge Transfer Learning. The knowledge learnt from trained models can be transferred and adapt to new tasks. This is especially helpful in IoT environments where usually limited data/labels are available. In this section we introduce four types of knowledge transfer learning (KTL) approaches: Transfer learning, Meta learning, Online learning and Continual learning.

Transfer Learning [266] — transferring knowledge across datasets— is the most popular KTL approach. It trains a model in the source domain (with adequate data/labels, e.g., on ImageNet [74] for general visual recognition tasks), and fine-tunes the model parameters in the target domain to accommodate the new tasks (e.g., medical imaging analysis on rare diseases). The rationale behind is that low-level and mid-level features can be representative enough and thus shared across different domains. In this case, only the parameters related to high-level feature extraction need to be updated. This mechanism does not require a large amount of data annotation for learning reliable representation in the new tasks, which could be useful in cases when annotations are expensive (e.g., medical applications). **Meta Learning** [277] is another popular KTL approach; instead of transferring knowledge across datasets, it focuses on knowledge transfer across tasks. Meta learning means learning knowledge or patterns from a large number of tasks, then transfer this knowledge for more efficient learning of new tasks. When with continuous data streaming, it is also desirable to update the model with incoming data, and in this case **Online Learning** [124] can be used. However, it is difficult to model when the incoming data is from a different distribution or a different task. Most recently, **Continual Learning**[210] was proposed to address this issue. Not only can it accommodate the new tasks or data with unknown distribution, it can also maintain the performance on the old/historical tasks (i.e., no forgetting [145]), making it a practical tool for real-world IoT applications. These four KTL approaches are similar in concept yet have different use cases. Transfer/Meta learning are focused on knowledge transfer across datasets or tasks (irrespective of data types), while online/continuous learning are more suitable for data streaming and can transfer the knowledge continuously to the new incoming data or tasks.

2.2.6 Discussion. Effort has been devoted to implementing the distributed machine learning on top of modern deep learning frameworks. Remarkable results have been achieved where with proper implementation [102] with Tensorflow, the training time of the state-of-art ImageNet can be reduced from days to one hour. Compared with Tensorflow, more efficient implementation such as Horovod can increase the GPU utilization for even more acceleration. Horovod has already been incorporated in various deep learning framework ecosystems (e.g., Pytorch, Mxnet).

Deep learning ecosystems free the researcher from heavy implementation effort. There are however, challenges for model generation in a distributed setting: (1) *The choice of hardware.* The same implementation can have different performance on different devices. One would have to be aware of the device features for efficient acceleration. (2) *Parallel hyperparameter tuning strategy.* Compared with single machine training, the distributed system is more complex and it is thus more difficult to find an optimal structure. (3) *Effective work of DL frameworks with other big data application like Hadoop/Spark.* Existing big data frameworks (e.g., Spark/Hadoop) can also be applied for effectively distributing the DL training pipeline, and a deeper integration of both frameworks is urgently required.

2.3 Model Optimization

We have discussed the model selection and model generation where a model is generated catering to the specific needs of IoT applications. There are, however, still things to be considered before model deployment. The IoT application differs significantly from other areas in terms of deployment devices and data sources. The limited computational budget of edge devices requires smaller models for small-scale computational workload to ensure low inference latency. Also, heterogeneous data sources in IoT environments usually contain redundant information that can even mislead the decision of the ML models. It is important to select only relevant and informative features or to perform model compression for performance optimization. In this section, we discuss these two topics.

2.3.1 Feature Selection. The high-dimensional data in IoT environments poses challenges on the training of the ML algorithms. Noisy and redundant signals exist and may consume substantial computational power. Feature selection can help reducing the computational complexity, improve the performance in terms of both effectiveness and efficiency—crucial factors in the limited-resource IoT environments. Briefly, feature selection is the process of preserving relevant features while discarding irrelevant/redundant features. There are generally three categories of feature selection, namely the *Filter* approach [16, 69, 112], the *Wrapper* approach [98, 110, 143, 184] and the *Embedded* approach [37, 223].

For future research, one could extend the single-object optimization to multi-object optimization. For example in IoT systems, optimal feature selection algorithms assist the machine learning models to optimize the execution time. We can explore the modifications of the feature selection algorithm to minimize the energy consumption of routing decisions as well [83]. One can also study how to detect the dynamics within the data flow and then adaptively apply the search algorithms accordingly to further improve the performance of the feature selection algorithms.

2.3.2 Model efficiency. The state-of-the-art DL models often require high computational resources beyond the capabilities of IoT devices. Models that perform well on large CPU and GPU clusters may suffer from unacceptable inference latency or even be unable to run on edge devices (e.g. Raspberry Pi). Tuning the deep neural network architectures to increase the efficiency without sacrificing much accuracy has been an active research area. In this section, we cover three main optimization directions: *Efficient architecture design*, *Neural architecture search* and *Model compression.*

Efficient architecture design. There exist neural networks that can specifically match the resource and application constraints. They aim to explore highly efficient basic architecture specially designed for platforms such as mobiles, robots as well as other IoT devices. MobileNets [126] is among the most famous works and proposed to use depth-wise separable convolutions [250] to build CNN models. By controlling the network hyper-parameters, MobileNets can strike an optimal balance between the accuracy and the constraints (e.g., computing resources). Later in MobileNetv2 [237], the inverted residual with linear bottleneck architecture was introduced to significantly reduce the operations and memory usage. Other important works include Xception [56] ShuffleNet [317], ShuffleNetv2 [178], CondenseNet [131]. These neural networks optimize on-device inference performance via efficient design of building blocks, achieving much less computational complexity while keeping or even raising accuracy on various computer vision datasets. Some work even outperforms the neural architectures generated through exhaustive automatic model search. Also, different building blocks can be combined together for even lighter models.

Neural architecture search (NAS). Another research direction named neural architecture search aims at searching an optimal network structure in a predefined search space. There are usually three types of algorithms: reinforcement learning approach [172, 214], Genetic Algorithm (GA) based [173, 228], and other algorithms [13, 39].

The models generated by these methods are normally constrained to smaller model sizes. Model size and operation quality are the two most common metrics to be optimized, over other metrics such as inference time or power consumption. Representative works including MONAS [128], DPP-Net [77], RENA [324], Pareto-NASH [85] and MnasNet [267] are interested in finding the best model architectures to meet these constraints. These approaches are more straightforward as they optimize directly over real world performance. However, one drawback of NAS is the extensive computing power required for finding the optimal neural architectures. Thus, the already generated architectures can be utilized as guidance for future design for more efficient neural network architecture.

Model Compression. As modern state-of-art DL models can be very large, reducing the model computation cost is crucial for deploying the models on IoT devices, especially for those latency-sensitive real-time applications. Model compression methods can be divided into four categories: 1) *Parameter pruning and sharing* that removes the redundant parameters [64, 109, 227, 276]. 2) *Low-rank factorization* that decomposes the CNN or DNN tensors to lower ranks [159]. 3) *Transferred/compact convolutional filters* that reduces the memory consumption by implementing special structural convolutional filters [161, 243, 307]. 4) *Knowledge distillation* that learns a new, more compact model that mimics the function presented by the original complex DL model [14, 120, 234, 306].

Types of model compression techniques have their own strengths and weaknesses and thus optimal choice is based on specific user requirements. *Parameter pruning and sharing* methods are the most commonly applied techniques for compression models from original models. It is stable as with proper tuning, this approach usually delivers no or few accuracy losses. On the other hand, *Transferred/compact convolutional filters* methods address the compression from scratch. This end-to-end efficient design for improving the CNN performance approach shares similar insights to the efficient *neural architecture design* approach as we discussed earlier. *Knowledge distillation* methods are promising when working with relatively small datasets as the student model can benefit from the teacher model with less data. All these methods are not mutually exclusive, we can make combinations based on specific use cases to optimize the models that are more suitable for low-resource IoT devices.

2.4 Model Evaluation

After the models have been trained, based on suitable metrics their performance should be evaluated before deployment. Accuracy is one of the most popular evaluation metrics in classification tasks, yet it faces several problems in different scenarios. For example, it is an overall measure without indicating the recognition capability for each class, which may be heavily biased if there exists a significant class imbalance problem. There are various evaluation metrics and it is key to select the most appropriate one. For the rest of this section we investigate several widely used metrics for classification and regression tasks. For classification/regression tasks, one aims to construct a model (i.e. $f(\cdot)$) that can predict the value of dependent variable Y from independent variable X. The difference between these two tasks is the fact that the dependent variable Y is numerical for regression and categorical for classification.

2.4.1 *Classification Problem based metric.*

In classification tasks, one of the most effective evaluation metrics is a confusion matrix [271]. As demonstrated in Table 2 for a binary classification task, in a confusion matrix the row represents the predicted class and the column represents the ground truth (actual class). The entries True Positive (tp) and True Negative (tn) represent the correctly classified positive and negative samples, while the entries False Negative (fn) and False Positive (fp) denote the misclassified positive and negative samples, respectively.

	Actual Positive Class	Actual Negative Class
Predicted Positive Class	True Positive (tp)	False Negative (fn)
Predicted Negative Class	False Positive (fp)	True Negative (tn)

Table 2. Confusion Matrix for Classification

Based on the confusion matrix, several evaluation metrics can be derived. The *accuracy* (i.e., $\frac{tp+tn}{tp+tn+fp+fn}$) and *error rate* (i.e., $\frac{fp+fn}{tp+tn+fp+fn}$ or $1 - accuracy$) are the most commonly used metrics because it is more understandable and intuitive for humans. However, these two metrics are powerless in terms of class-wise informativeness [179], which may neglect the minority class [44] (if there is a class imbalance problem).

The metrics *precision* and *recall* can be used to measure the performance irrespective of the class imbalance problem (more definitions of classification evaluation metrics can be found in **Table 4 in Appendix C**). In binary classification problems, as mentioned earlier tp, fn, fp are defined as the the number of "positives correctly classified as positives", "positives incorrectly classified as negatives" , "negatives incorrectly classified as positives" respectively. Then we can see *recall* (i.e., $\frac{tp}{tp+fn}$) indicates the ability of a classifier to detect (true) positives out of all positive instances, while *precision* (i.e., $\frac{tp}{tp+fp}$) is the percentage of detected (true) positives out of all the detected ones. Since the binary classification's decision may highly depend on the threshold, there is a trade-off between precision and recall. For example, if a high threshold has been chosen—the similarity scores (the model outputs) have to be higher to give positive decisions—the classifier tends to have high fn and low fp, yielding low recall and high precision. Similarly, reducing the value of the threshold may increase recall and decrease precision accordingly. For different applications, one needs to consider the optimal threshold for their requirements. For example, forensic applications may prioritize high precision (i.e., low in fp) while a medical diagnosis may prioritize high recall (i.e., low in fn).

In some tasks when both recall and precision are important, the *F1-score* (i.e., $2\frac{precision \cdot recall}{precision+recall}$)—a measure that can balance the precision/recall trade-off— is normally used. It is worth noting that for multi-class cases, the multi-class confusion matrix can be calculated, and the aforementioned precision/recall/F1-score can be extended to measure the class-wise performance. Depending on the data/applications, the overall performance can be measured by aggregating all the class-wise metrics. Two popular aggregation operations are averaging, and weighted averaging, e.g., mean F1-score, or weighted F1-score (over all the class-wise F1-scores).

2.4.2 *Regression Problem based metric.*

For regression problems, the evaluation metrics are different from the classification ones. Popular evaluation metrics include Mean Squared Error (MSE), Mean Absolute Error (MAE), Mean Percentage Error (MPE), etc. Details and formulas of these metrics can be found in **Table 5 Appendix C**.

Mean Absolute Error (MAE) and *Mean Squared Error (MSE)* are the simplest metrics for regression evaluation. They denote the expected model errors defined in terms of absolute difference and squared difference (between the predicted value and the ground truth), respectively. Alternatively, the *Mean Absolute Percentage Error (MAPE)* and *Mean Percentage Error (MPE)* can also be applied to regression problems. The MAPE is similar to MAE but more intuitive as it shows percentage. The MPE lacks the absolute term on MAPE, which means the positive and negative errors will cancel out. In this case, the MPE can not be directly used to measure the performance of a model. However, it can be used to check whether the model systematically underestimates (more negative errors) or overestimates (more positive errors).

All of the above metrics can be applied to the regression problem, but it is important to consider the property of the dataset beforehand. For example, some fields may (or may not) be more prone to outliers, and the corresponding (effective) evaluation metrics may be different.

3 MODEL DEPLOYMENT

When the ML model development process is finished, the developed models are to be deployed and composed as an application in the complex IoT environments. To simplify the deployment, the ML models and underlying infrastructure need to be specified (§3.1). Next, the optimization techniques can be applied to generate the deployment plans that select the suitable ML models for the deployment, optimizs the resource utilization of the model deployment and improve the reusability of the deployed models (§3.2). Once the deployment plans are generated, the models will be deployed over the specified infrastructure and the deployed models will be composed as defined in the plan (§3.3).

3.1 Declarative Machine Learning and Deployment

Declarative ML. Declarative ML aims to use high-level language to specify ML tasks by separating the applications from the underlying data representation, model training and computing resources. There are *three* general properties of declarative ML. First, the high-level specification only considers data types of input, intermediate results and output. They are exposed as abstract data types without considering the physical representation of the data or how the data is processed by the underlying ML models. Second, the ML tasks are specified as high-level operations through well-defined semantics. The basic operation primitives and their expected accuracy levels (or confidence interval) are defined accordingly. Based on the operation semantics, declarative ML systems select the features and underlying ML models automatically or semi-automatically, optimize the model performance and accuracy for varying data characteristics and runtime environments. Notably, the selection is based on the available models, provided as services. Finally, the correctness of the ML models must be satisfied when a given model produces the equivalent results in any computing resources with the same input data and configurations. As a result, the declarative ML enables execution of the ML models over various hardware and computation platforms (such as Apache Spark) without any changes. Besides, these specification languages also bring flexibility and usability in the ML model deployment stage.

SystemML [28] is an implementation of declarative ML on Apache Spark. Through domain-specific languages, it specifies the ML models as abstract data types and operations, independent of implementation. The system is able to specify the majority of ML models: matrix factorizations, dimension reduction, classification, descriptive statistics, clustering and regression. There is also other state-of-the-art research on declarative ML, including TUPAQ [256] and Columbus [309]. They utilize language specification and modelling technologies to describe the ML models for automatic model and feature selection, performance and resource optimization, model and data reuse.

Declarative Deployment. Hardware in the IoT environment consists of three basic types of device: *data generating* devices, *data processing* devices and *data transferring* devices. *Data generating* devices are also called "Things" (e.g., sensors, CCTV) and are used to collect environmental data. *Data transferring* devices such as router, IoT gateway, base station are used to transfer the generated data to the *data processing* devices. *Data processing* devices are used to run the analytic jobs. They can be GPU, CPU and TPU servers running in cloud or ARM based edge device such as Raspberry Pi and Arduino. An ML-based IoT application is usually running across a fully distributed environment, such that it requires correct specification of the component devices as well as the precise interoperation between these devices. [252] lists fundamental aspects that may simplify the hardware specification, i.e., processor, clock rate, general purpose input/output (GPIO), connectivity methods (Wi-Fi, Bluetooth, wired connection) and communication protocols (serial peripheral interface), universal asynchronous receiver-transmitter (UART).

Regarding the software, it is often categorized into three groups based on operation levels: *operating system (OS)*, *programming language* and *platform*. IoT *OS* allows users to achieve the basic behavior of a computer within internet-connected devices. The choice of OS in different layers of the IoT environment depends on the hardware properties such as memory and CPU. The *programming language* helps the developers to build various applications in different working environments with diverse constraints. The choice depends on the capability of devices and the purpose of the application [41]. The IoT software *platform* is a system that simplifies the development and deployment of the ML-based IoT application. It is an essential element of a huge IoT ecosystem which can be leveraged to connect new elements to the system. For more details of the most popular OSs, programming languages and platforms in IoT domain, one can refer to **Appendix A**. The ML development *platforms* have been discussed in § 2.2.1.

The heterogeneity of IoT infrastructures makes the deployment very complicated and difficult to automate. To overcome this issue, the infrastructure must be described and specified by machine understandable languages. Then, the declarative deployment systems are able to automatically map the ML models to the infrastructures and generate the deployment plans that optimize the performance and the accuracy.

The declarative TOSCA model [68] is able to specify the common infrastructures such as Raspberry Pis and cloud VM (hardware), MQTT and XMPP (communication protocol). The deployment logic can be defined through *TOSCA Lifecycle Interface* that allows users to customize the deployment steps. However, this declarative model is still very basic and can not handle complex deployments such as specifying the details of ML based application. Moreover, the IoT applications consist of installing devices and sensors which require *human tasks*. These tasks are not natively supported by any available declarative deployment [38]. The imperative tool (e.g., kubectl commands) allows the technical experts with diverse knowledge of different deployment systems and APIs to interact with a deployment system and decide what actions should be taken. However, current imperative frameworks such as Juju, Kubernetes still do not support interactions such as sensor installation. In future, declarative deployment systems should interact with declarative ML systems to deploy a complex application over the heterogeneity of IoT infrastructure while supporting the *human* tasks through a more human centered imperative deployment model.

3.2 Deployment Optimization

When the infrastructures and deployment workflow of the ML models are specified, the deployment optimization problem can be formed as a mathematical expression subject to a set of system constraints. Then, resource allocation algorithms can be used to efficiently and precisely find the best solution for the given mathematical expressions. Moreover, the optimization objectives are a set of QoS parameters including storage and memory, budget, task execution time and communication delay etc,. These algorithms can be divided into *two* classes based on whether an optimal solution can be guaranteed: *meta-heuristic method* and *iterative method* (or *mathematical optimization*). Nowadays, ML methods are becoming popular and being applied to solve these resource allocation problems by learning "good" solutions from the data. We investigate the representative works in resource allocation based on these *three* classes.

Iterative-based method. This class of algorithm generates a sequence of improved approximate solutions with each driven by previous solutions. Eventually, the solutions will converge to an optimal point proved by a rigorous mathematical analysis. The heuristic-based iterative methods are also very common, categorized as *meta-heuristic based method*. The most popular algorithms of this class include newton's method [180, 181], gradient method [17] and ellipsoid method [176]. To apply and adapt iterative-based algorithms to optimize resource allocation requires strong mathematical background, which can be an obstruction for software developers to utilize these algorithms to optimize their deployment. Furthermore, the algorithms perform for differently for different problems in terms of efficiency and accuracy. As a result, more algorithms from iterative-based methods need to be studied and simplified by the system researchers, providing toolkits (or solvers) to tackle different optimization problems in IoT application deployment.

Meta-heuristic based method. The optimization problems in IoT applications can have large search spaces or be time-sensitive. The *meta-heuristic* based method is faster than *iterative-based method* in finding a near-optimal solution. This type of method consists of two subclasses: *trajectory*-based method and *population*-based method. The *trajectory*-based method finds a suitable solution with a trajectory defined in the search space. First, the resource allocation problems are mapped into a set of search problems such as variable neighborhood search, iterated local search, simulated annealing and tabu search. Then, the *meta-heuristic* algorithms are used to find the solutions. Many survey papers [114, 174, 253] have reviewed the algorithms applied for resource allocation in IoT, cloud computing, mobile computing. Additionally, *population*-based methods aim to find a suitable solution in the search space described as the evolution of a population of solutions. This method is also called evolutionary computation and the most well-known algorithm is the genetic algorithm. [308] investigates the resource allocation problems solved by evolutionary approaches in cloud computing.

Machine learning based method. ML based method is inspired by the ability of data to represent the performance and utilization of the contemporary systems. The ML based methods are used to build data-driven models that allows the target systems to learn and generate an optimized deployment plan. The proposed algorithms have been used to optimize various QoS parameters such as latency [183, 299], resource utilization [194], energy consumption [21] and many others. Zhang et al. [310] have given a comprehensive survey of the ML based methods used for resource allocation in mobile and wireless networking.

Deployment (or resource allocation) optimization problems have been studied for decades, and remain a huge legacy for overcoming the optimization problems in deploying ML-based IoT applications. Instead of developing new optimization algorithms, more efforts are required to model the complex optimization problems, in which the system scale, conditions and diversity have been amplified significantly.

3.3 Action and Model Composition

Deployment of ML models in a pipeline requires proper model composition to maximize the user QoS. As shown in §1.1, a smart car navigator system comprises multiple ML models, including speech recognition, text classification, text generation and text-to-speech (TTS) model.

Action composition is defined by composing a set of basic actions for complex decisions. In a self-driving car operating system, actions can be accelerating, braking, turning left and right, etc. The combination of various action spaces increases the difficulties of learning optimal decisions in such complex systems. Hierarchical abstract machines (HAM)

[254] are well studied in the context of reinforcement learning [211, 270] by allowing agents to select from a constrained list of action spaces, speeding up the learning and adaptation to the new environment.

Model composition aims to create a ML-based IoT application by using reusable, portable, self-contained modules via inserting new components or removing existing components. Apache Airflow is an open-source platform for creating, scheduling and monitoring workflows in Python. The Valohai operator is an extension of Airflow that utilizes the docker container to build self-contained modules for each model while providing the flexibility for users to define the steps to execute. [146] reported the following challenges for chaining the ML models:

- How to allocate computing resources automatically for different models. An application is chained by various ML models require different computing resources across heterogeneous infrastructures. It is challenging to provision the computing resources efficiently for the chained ML models while meeting their performance requirements.
- How to chain the dependent models. Each individual ML model has its own specification and data format of the inputs and outputs. The challenge is to design a data messaging system to orchestrate the data flow across different models while considering their required specification and data format.
- How to meet the security requirements. The inferences are performed through various components, with each deployed across different computing resources. This introduces a set of challenges including privacy, verification of outputs of each model, changing the security policies of components, etc.
- How to monitor failure. The composed application consists of a set of ML models that needs to be monitored, ensuring that everything is streamlined and executed as anticipated.

Apart from challenges mentioned above, the literature discusses the techniques to improve the performance of individual models via system configuration, including *model batch size*, *model replica* and *system buffering*.

Per-Model Batch Size. Batching the received user queries optimizes throughput by fully utilizing the features of the pre-trained models, which is faster than processing one query at a time. However, batching query can potentially increase latency because the the model will wait for a whole batch of queries to come before it starts to proceed. The first query is not returned until the final query is processed [65]. The choice of the per-model batch is challenging due to the sequential composition between the models.

Model Replica. In heavy or bursting loads, a system must quickly respond to the query fluctuations to meet the latency requirements. To alleviate the system congestion and achieve high throughput, it is critical to identify the bottleneck, which can be challenging due to the system dynamics. The bottleneck models can be resolved by replicating the model instances across multiple devices [66], therefore balancing the workload. However, distributing the queries across more model replica in a parallel setting [66] is also challenging since the optimal placement depends on the model performance and the device capacity.

System Buffering. Serving system as a stream processing system comprises components across multiple devices. These devices usually process at different speeds, making system buffering across nodes necessary. Message queues are usually implemented to ensure smooth running within the system. However, buffering mechanism would increase the latency based on various system configurations [66]. It is thus challenging to design proper strategies to balance the message queue overhead and the system latency.

4 MODEL AUDIT

Audit aims to evaluate whether the application is operating effectively, safely and reliably with the collected evidence. To this end, we must know what we should audit. Most work focuses on monitoring or debugging the issues caused by infrastructure failures [155], implementation bugs [78, 153] and deployment errors [263]. In this section, we investigate the security, reliability and performance issues caused by ML models, especially DL models.

4.1 Security

There are many surveys regarding IoT security issues and challenges. The security of IoT standardized communication protocols were evaluated in [103] based on their proposed model. [249] categorized the security issues of IoT into *eight* domains including authentication, access control, confidentiality, privacy, trust, secure middleware, mobile security and policy enforcement. [233] studied the main challenges and solutions of designing and deploying security mechanisms in centralized and distributed IoT architectures. [169] discussed the security features of IoT and categorized the attacks into *four* layers. i.e., the perception layer, the network layer, and the application layer.

In this subsection, we discuss security issues for deep learning based IoT applications: *Model exploratory attack*, *Data poisoning attacks* and *Evasion attacks*. *Model exploratory attacks* do not happen during training, instead the attacker tries to discover information from the trained model including the model itself and training data. *Data poisoning attacks* happen during the training phase, where the attacker attempts to shift the boundary of DL models in their favor by polluting the training data. Finally, *evasion attacks* maliciously craft the inputs for the deep learning based IoT application to trigger abnormal model behavior. Interestingly, the development of the research on adversarial learning has started an arms race between adversaries and defenders.

The following subsection summarizes the most popular attacks (The defenses can be found in **Apendix D**) of these attacks. We also propose research directions for development of robust IoT applications.

4.1.1 Model exploratory attack. This type of attack is usually performed on open-source frameworks such as PredictionIO and cloud-based machine learning service. This ML-as-a-service may allow users to input partial feature vectors while still being able to receive confidence values in addition to prediction results. Thus, the attacker can leverage this feature to either extract the model or the sensitive information underlying the model. *Model stealing* and *Membership leakage* are *two* main types of model exploratory attack. *Model stealing* attack aims to duplicate the functionality of the model that allows the attacker to evade detection by the stolen model [10, 204]. [272] proposed a method that learns the target models via a prediction API. Evaluations show that this method successfully extracts the models including logistic regression, SVM, neural network and decision tree from BigML and Amazon Web Services. More attack methods can be created based on the extensive literature on learning theory, e.g., PAC learning [275] and its variants [19]. *Membership leakage* attacks are interested in stealing the information from the training data which may not be publicly available and may contain some sensitive information such as trade secrets, medical records etc. In this type of attack, an attacker is able to infer the members of the population or the members of the training dataset. Attacking the members of the population means that the types of data are used to create the model. Therefore, the target model has not been generalized for the adversary, because he/she has the sample of the entire population of the training dataset. The attacks were successfully performed in the dataset including voice, handwritten images, network traffic, online shopping, record of hospital stays etc [10, 247]. The members of the training dataset attack aim to identify the individuals whose data are used for training the model, which causes a serious privacy issue. For example, if an attacker knows that a patient's medical record was used to train a disease detection model, it also reveals that the patient has this disease. The experiments in [272] show that the attacks are able to extract the training dataset when the model is based on kernel logistic regression.

4.1.2 *Data poisoning attack.*

Unlike *model exploratory attack*, an adversary performs the attacks during the model training phase. These attacks insert carefully constructed poison instances into the training dataset to manipulate the performance of a system. We introduce types of data poisoning attacks both in traditional machine learning and deep learning.

Data poisoning attack in machine learning. (1) supervised learning. A causative attack was proposed by Xiao et al. against SVMs which utilizes *label flipping* to poison the training data [294]. A *label flipping* attack attempts to add a noise label to the training data. These flipping labels are able to cause some malicious samples to be labeled as legal, or make legal samples appear to be malicious. To improve the efficiency of the attack, Biggio and Laskov [23] utilized the gradient descent algorithm to find the best attack points to flip the labels.

(2) unsupervised learning. The poisoning attack has been demonstrated against various clustering algorithms. The idea is to introduce carefully crafted data points to the training dataset to cause clusters to merge. In [24, 231], the authors assumed that the attacker has full knowledge of the clustering algorithm and then reduced the attack to an optimization problem. The evaluations show that the clustering algorithms are compromised significantly with a very small percentage of poisoned input data.

Data poisoning attack in deep learning. There are very few data poisoning attacks in neural networks. [258] showed that a deep learning model lost 11% accuracy after modifying 3% training data. Moreover, if the attacks are focus on attacking the specific test instances, the successful rate, time consumption and required resources (the number of modified samples) can be reduced significantly [50, 106, 258]. [242, 262] targeted real-world scenarios where the labels are examined by human reviewers and malware detectors. The authors aimed to overfit the deep learning models by poisoning the training data. Thus, the target instants (trained models) would not perform well during inference time.

4.1.3 *Evasion attack.*

With the explosive development of machine learning, evasion attacks are becoming the most prevalent type of attack in machine learning, attracting people's attention from both academia and industry. Fig. 10 shows the arms race between the attacker and defender. It shows that the attacker attempts to confuse the defender with a crafted adversarial example, while the defender aims to strengthen its ability to filter out illegitimate input.

During both training and inference, the attacker can generate *adversarial examples* by modifying the samples. The training phase modification is similar to *data poisoning attacks* in that the decision boundary of the defender classifier is modified by insertion, modification or deletion of the training dataset. There are two approaches for generating adversarial samples, *white-box* or *black-box*. In the *white-box* setup, the adversarial samples are crafted based on the attacker who has access to both the training data and the targeted model. Therefore, an adversary is able to obtain the boundaries of the targeted model by carefully modifying the training data. To be more explicit, as shown in Fig. 10, the defender aims to stop using the illegitimate input X to train the itself. In order to fool the defender, the attacker attempts to learn the boundaries of the defender by adding the perturbations to X and then performing the attack. This process is repeated until the adversarial samples break the boundaries. The most representative techniques [101, 157, 209] are based on the attacker who has knowledge of both target model and instance of data. In the *black-box* setup, the attack introduced in [208] is not aware of the training data and the targeted model. The only observation of the targeted model is the inputs and their labels given by the targeted model. Based on this, a local model is trained to replace the target DNN. 84% of the adversarial examples generated by the local substitute model are misclassified by the targeted DNN.

4.1.4 *The system challenges of building a secure ML-based IoT application.*

Most of the attacks and defenses reviewed in previous sections focus on developing the algorithms for *functional tasks* such as computer version, natural language processing, audio speech processing etc. To build a secure IoT application, we must consider the security issues from

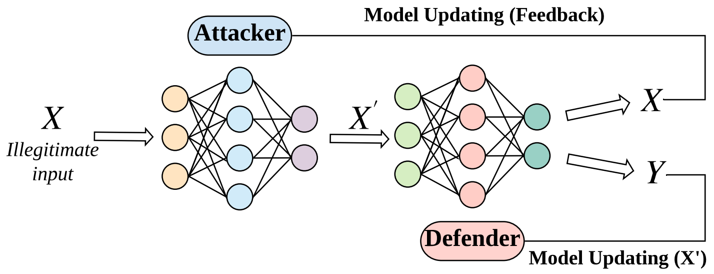

Fig. 10. Arms race game between the attacker and the defender. The attacker takes an illegitimate example (X) as the input of his/her neural network and generates an adversarial example (X'). This example attempts to fool the defender which is a classification neural network. If the classifier recognizes the adversarial example as Y which belongs to the legitimate input, it means the attacker wins the game. On the contrary, the defender wins the game if the adversarial example is classified as X. When the attacker fails, it will try to update its model to generate a stronger adversary example based on the feedback. Similarly, the defender will enhance its model based on the lesson learned from the successful attack.

the system perspective, as the *functional tasks* cannot perform well when the system is under attack. We discuss two system challenges to improve the security of ML-based IoT application.

Developing new attacking and defending models. ML has been widely used in IoT system developments including network engineering [168, 183], resource allocation [162, 298], system debugging [43, 251], network intrusion detection [149, 175, 280] and network operations[130]. These systems can be exposed to the aforementioned attacks as well. The literature [2, 87, 113] has revealed successful examples of attacks and defenses. Further to the efforts on *functional tasks*, more research and development (R&D) work is required to improve the security of ML-based IoT applications.

Developing new security platforms/frameworks. We have discussed the arms race game between the attacker and defender (see Fig. 10). This can be utilized to ensure the resilience of IoT applications to various attacks. At a high level, an ideal platform would be able to launch various attacks via a predefined deployment pipeline to attack the experimental group. Meanwhile, the attack behaviors and system performance will be monitored to reinforce the capacity of the defender. To this end, three research questions need to be answered. 1) *How to automate the attacks.* Unlike the traditional software deployment problem, deploying attacks is much more complicated. For example in an evasion attack, the proposed platform must be able to use various ML models to craft the adversarial examples. It is very difficult to automate this process. Due to the difference between the model inputs and outputs, the models may need to be retrained based on the observation of the real world to generate better adversarial examples. 2) *How to monitor the attacks.* As discussed in previous sections, attacks can happen in data collection, model training and model inference. Therefore, the traditional log system is not able to handle this complexity. In a data poisoning attack, for instance, the traditional log system is unable to capture the impact caused by fake data points injection into the system, thus the training of a defender is unfeasible. 3) *How to coordinate the attacker with the defender.* At the high level, the arms race game between the attacker and the defender is very logical. The challenge here is to continuously select the suitable attacks and thereby improve the defender's performance. This can be formalized as an optimization problem where one of the objective is to maximize the ability of a system in defending against certain types of attack.

4.2 Fault Tolerance

Distributed system fault tolerance has been studied for decades. Many works have been proposed to handle the failures including system architecture [226] and algorithm design [42]. In the IoT environment, the probability of failure increases significantly with many faults hard to detect. Our previous papers [94, 291] reviewed the state-of-the-art research and discussed key research directions. In this subsection, we introduce some common faults in IoT-ML applications.

4.2.1 Faults in ML. Generalisation is crucial for ML models, which measure the prediction capacity on unseen test data [34]. Generally, ML training can be regarded as an optimisation process. For example, the model can be trained by minimising a certain loss function. However, overfitting may occur when ML models are trained on less representative, noisy or small data, and in this case, trivial error patterns may be learned, causing lack of generalisation (i.e., faults) at the test stage. There are many ways to reduce the overfitting effect, such as regularization (e.g., with regularization terms such as L1/L2 norm), *Stochastic gradient descent (SGD)* [31] or dropout [257] (for DL models), early stopping [20] (stop training when validation error starts to increase), etc.

Data imbalance may also cause overfitting. The model may mainly learn patterns from the majority classes while it may easily ignore the contributions from the minority classes (with limited training samples), yielding severe faults at the inference stage. Various approaches have been proposed for mitigation including data augmentation (e.g.,[281]), data upsampling (e.g., GAN-based data generation [268], [279]), cost-sensitive learning (which will impose a larger penalty on training errors with minority classes), transfer learning, etc.

In addition to overfitting effect, faults can also be attributed to the optimisation process. For example, with very deep DL models or with RNN, gradient vanishing/explosion may occur during the optimisation process, causing representation learning to be challenging or even infeasible. There are also several approaches to address this issue, e.g., *Batch Normalization (BN)*[135] (through normalizing the gradients in each layer), residual connection structure in DL (to preserve the gradient across many layers). For federated learning or distributed learning, RSA(i.e., Byzantine-Robust Stochastic Aggregation)[165] has also beenproposed to prevent the incorrect gradient aggregation.

4.2.2 Fault tolerance in neural networks. At a high level of abstraction, neural networks can be viewed as a distributed system. The failure can happen in *neuron* or *synapse*. In [192], Mhanmdi and Guerraoui proposed a general model to describe the fault model of neural networks. The *neuron* may stop computing (Crash) or generate some abnormal outputs (Byzantine). Similarly, the failures of *synapse* can be abstracted as Crash and Byzantine. Crash represents that the transmission has not succeeded, and Byzantine is that the incorrect messages are sent from the source *neuron* to the destination *neuron*. Thus, we assume that a given neural network $\mathcal{N}$ performs an expected output $F_{\mathcal{N}}(X)$, and $F_{\mathcal{N}_{fault}}(X)$ is the output of the faulty network obtained from $\mathcal{N}$. The distance ϵ between $F_{\mathcal{N}}(X)$ and $F_{\mathcal{N}_{fault}}(X)$ represents the fault tolerance of $\mathcal{N}$, when there are at most n faulty components (including *neuron* and *synapse*):

$$\parallel F_{\mathcal{N}}(X) - F_{\mathcal{N}_{fault}}(X) \parallel \leq \epsilon \tag{1}$$

where X is the training dataset, applied to both $\mathcal{N}$ and $\mathcal{N}_{fault}$. In order to guarantee the robustness of the neural model, the designer needs to ensure that the error (left hand in Equation 1) is below a predefined threshold (right hand in Equation 1). The threshold depends on the performance of the network and its intended application [192, 217].

Like that in distributed system, the fault tolerance in neural networks also has two types: *Passive* and *Active*.

In **passive** fault tolerance, no diagnostics, relearning, or reconfiguration is required thereby avoiding fault detection and location. The most common *passive* fault tolerance approach which is also one of the important features of neural network is inserting redundancy. Such methods learn a small network from the given input/output, and then add the replicated hidden neurons to share the load of the critical nodes, after the model has been trained. Representative works [55, 59, 86] addressed the fault tolerance by adding extra links or nodes to the well trained neural network. The authors in [54] proposed a solution that adds artificial faults to the network during the training time. Therefore, the network can tolerate the specific faults. However, this approach requires that the neural network designers are aware of all the faulty scenarios while building the network. Also, adding redundancies makes the models very complex and huge, which brings the challenges of deploying them over lightweight and low-power IoT devices.

Active fault tolerance aims to recover the neural model from faults by resetting the neural network into a fault-free state. However, it does not attract too much attention from research, a common strategy is to utilize high-performance computation resources to re-compute the lost work when the hardware fails [1, 287]. Notably, Qiao et al. proposed a checkpoint-based fault tolerance for deep learning in [219]. This new method partially recovers the model from the checkpoints based on the priority of the checkpoints thereby significantly reducing the cost of recomputing.

4.3 Performance Evaluation

In this section, we consider several performance criteria that need to be considered for evaluating the efficiency of the obtained ML models. The criteria is identified as *two* main dimensions: *model precision* and *execution latency*.

Model Precision. In a typical IoT application, the software performance is assumed stable after deployment. However, this is not the case for ML application where precision degradation is always expected after deployment. Precision degradation can happen as various unexpected external changes lead to shift in data distribution. Device location change, time and the weather are all important factors that may decrease the model performance. Therefore, it is critical that the model performance is monitored and new data is introduced continuously for retraining of the model. In ML, we define lifelong learning [210] as continually acquiring data and extracting new information without catastrophic forgetting of past knowledge. Lifelong learning keeps the model precision at a steady level.

Execution Latency. Many IoT applications are latency-sensitive depending on their tasks. For example, in the aforementioned smart transportation system (see §1.1) where sensors monitor and detect car accidents, instant decisions have to be made to warn the drivers of potential hazards. Various factors, listed below, have to be evaluated to ensure seamless communication among the distributed components of a smart IoT application.

Bandwidth Usage. In distributed IoT networks, large scale IoT sensors are generating a huge amount of data all the time. It is not possible to send all the data to the cloud for data analysis. Fog computing proposed to move the computing close to the sensors to reduce the data transmission over the IoT network. However, the bandwidth of sensor network and edge network are still limited, some nodes may experiences high latency due to the network congestion. This may cause huge latency for the whole system as well. We need to monitor and evaluate this network dynamic [182] in order to provide solutions to alleviate the congestion in the networks.

Resource Consumption. Hardware in IoT applications varies in computing power, memory and storage capacity. For any resource-intensive tasks, for example those computation-heavy or memory-heavy ones, resource exhaustion in one node may lead to unacceptable latency for the whole application. It is thus necessary to design efficient resource management systems [199, 301] to monitor and optimize task allocation for these physical devices,

System Throughput. The ML-based IoT applications may be developed to serve millions of people, for example, the *smart traffic routing application* mentioned in 1.1. This massive number of users may send the requests simultaneously. Responding to these requests quickly without losing user satisfaction is still an unsolved problem in cloud computing. However, this issue is amplified in ML-based IoT applications, in which the queries may be performed on various devices and models. Some database optimization techniques such as caching frequent queries, batching queries and approximate computing are applied [66, 212]. There are remaining gaps in optimizing the query plans by considering heterogeneousness of the computing resources, uncertainty of the network, and diversity of ML models.

5 DATA ACQUISITION

Data is one of the most important constituents in developing a ML model as the prediction accuracy of the model is highly correlated with the quality of the input data [195]. To provide high quality data for ML-based IoT applications, we orchestrate the data acquisition process into several steps. First, raw data are collected from various data sources (§5.1). With proper preprocessing techniques (§5.2) to remove redundant information and annotate the data, we are capable of performing several different ML tasks. While we have more data sources during the development process, we can also fuse (§5.3) them to provide more consistent and useful information. The following subsections will focus on the mentioned steps and discuss how data acquisition can support development of a robust ML-based IoT application.

5.1 Data Collection

The IoT data can be broadly categorized into *Structured data* and *Unstructured data* based on its representation. *Structured data* can be represented in a pre-defined format (rows and columns). The meaning of each field is explicit which eases the analysis and storage of the data. Examples of structured data include employee register information, visiting logs, etc. On the other hand, *Unstructured data* lacks any specific structure or format. Varying from text, audio, video to mails and messages, it accounts for a large proportion of IoT data. These two types of data are generated in *three* formats: *signal data*, *log data* and *packet data*. The *signal data* collects the daily life signal through various hardware such as sensors, sound recorders, cctv cameras, etc. The *log data* is usually used to capture the system status. Finally, the *packet data* is the data sent over the network and each unit transmitted consists of a header and the actual data. To collect these data, *three* important factors need to be considered: 1) *Data exchange*, 2) *Resource consumption* and 3) *Concept Drift*.

Data exchange. Data generated from IoT devices is sent to an edge (sink) node or other IoT devices, eventually collected and stored in the cloud. The computation power of gateways and edge nodes is improving, which brings an opportunity to remove data redundancy while saving the energy and bandwidth required for transferring data to downstream nodes [289]. This aggregation requires application of various data summarization techniques [61] including sampling, sketching, histograms, wavelets and adaptation of these techniques to meet the constraints of the hardware and the time-varying channel conditions. Henriette et al. [232] investigated the state-of-the-art stream processing systems that can be used to implement these data summarization techniques and execute them in a parallel and elastic manner. However, it still requires a lot of effort to develop new data summarization techniques and stream processing systems to handle the difficulty of processing high volumes data from various sources with multi-modality.

Resource consumption. As mentioned earlier, IoT devices are very limited by resources such as processing capability, storage capacity, wireless bandwidth and battery power. Thus, it is very critical to optimize the resource utilization while processing, storing or transferring data to the edge device or cloud. To this end, we need to consider *three* issues: *resource allocation*, *energy control* and *task allocation*. *Resource allocation* in the context of data collection is to assign computing, storage or bandwidth resources to the data generated by IoT devices before transferring to edge or cloud. Sending streaming data drains the battery at a faster rate while limited storage capacity does not enable large data storage. *Energy control* focuses on optimizing the energy consumption when the IoT data is processed and transferred over the devices. *Task allocation* aims to balance the resources consumption in IoT devices while minimizing the overall latency. These *three* factors are sometimes considered together and most of the available algorithms are based on market-enabled pricing schemes, which dynamically exchange the resources among the devices in IoT infrastructure by creating an artificial market [88, 139]. In ML-based IoT application, the ML models should be considered as the special tasks that are running on extremely heterogeneous computing resources in a distributed manner, and these tasks are usually compute-intensive, data-intensive and network-intensive. As a result, it is crucial to develop new market models to describe these special resource consumption problems and new algorithms to solve the problems.

Concept Drift. Due to the dynamicity of the IoT environment, data distribution becomes very uncertain and changes frequently over time leading to *concept drift* [93]. Changes can occur abruptly or gradually correlated with the occurrence of other events. Additionally, the change can be in different forms i.e., input data characteristics or relation between input data and target variables with single or multiple occurrence (constant or variable recurrence). For the successful execution of IoT applications, these drifts need to be predicted, distinguished from noise and handled properly. Numerous algorithms are proposed for managing concept drift. [70, 129] review the generic algorithms to handle the concept drift. There are two main detection methods, *performance-based* and *data distribution-based*. The former can work well if the data is labeled which may not be possible for all cases while the latter is able to detect only a subset of available drifts. Since IoT-based ML application data are not always labeled and high accuracy is desired, it is essential to develop new algorithms which are able to detect and manage the concept drift.

5.2 Data Preprocessing

The real-world data collected from heterogeneous IoT devices usually contains outliers or is incomplete in nature, which makes it difficult to feed it into ML models directly. Data preprocessing deals with these anomalies and improves the data quality and practicality. There are several things that need to be considered, namely *data cleaning*, *data annotation* and *feature engineering*. We have discussed the details of *feature engineering* in §2.3.1, will not consider it in this section.

5.2.1 Data cleaning. Much data contains noise that is bound to confuse the ML models and reduce the accuracy of the prediction results. Data cleaning resolves this problem by completing several routine tasks such as *filling missing values*, *smoothing noise data* and *removing outliers* [5, 27]. Empty records in the data set can be replaced manually by a specific value, for example the attribute mean or the most common attribute in the set. It can also be marked with "unknown" or just ignored if the dataset is large enough. Noisy data, though, can be smoothed by grouping first and then averaging over each group. Data outliers can also be detected during this process if the value exceeds a predefined threshold. There are other common practices such as data normalization [220], which is used to scale dimensions of data to a specific range. This is very efficient when there is high variation for different dimensions of the data.

5.2.2 Data Annotation. As discussed in §2.1, data annotation is necessary for supervised learning based ML models, in which both the data and the corresponding target act as the input sample. The model is trained with the labeled data which is used to predict the target for new unseen data. This is usually costly and complex due to the requirement for a large volume of labeled data needed for the training. The following investigates different annotation methods that can be applied according to the size of the data to be annotated and the cost of annotation per data.

Manual Annotation: At the initial stage of a ML project, quick prototyping of a workable model requires only few labeled data. In this sense, the developers can manually annotate the collected data to create a small dataset. This is usually done by reviewing the data samples and attaching labels following the annotation guidelines. Manual annotation by the engineers is quick and precise without any professional training, and the data quality is usually great. the problem with this approach is the lack of scalability.

Crowdsourcing Annotation: Crowdsourcing annotation is a scalable and cost-effective method. It is usually orchestrated by an online platform that provides access to a workforce of people to complete the annotation tasks. Famous crowdsourcing platforms include Amazon Mechanical Turk (MTurk). Compared to manual annotation, this approach can be scaled to large dataset labeling. However, the crowdsourcing method requires delicate design on quality control mechanisms to ensure the annotation quality, and the incentives or rewards for the crowds.

Active Learning: Active learning [97, 241] aims to design a system capable of choosing and learning from less training data while still achieving the same or even higher accuracy. An active learning system consists of two components: a *learning module* that trains a model with the current training sample and a *sample selection module* that selects the most informative samples from the unlabeled samples. The selected samples will then be annotated manually and added to the training set. The iterative process continues until the training converges. The key here is the sample selection module which can be approximately subdivided into *five* categories, *risk reduction*, *uncertainty*, *diversity*, *density* and *relevance* according to the selection criteria [282]. These criteria can be used either single-handedly (e.g., risk reduction [90], uncertainity [142], relevance [11]) or in a combination. In [283], *uncertainty*, *diversity*, *density* and *relevance* are combined for multi-modality video annotation. Similarly, work in [123] combines *uncertainty*, *diversity* and *density* metrics and the evaluation proves the combination performs well on medical image classification tasks.

All the above explained methods work well for the case of static machine learning scenarios with batch data available beforehand, However, this may not be suitable for IoT-specific streaming data imminent with high concept drifts. In this case, the model needs to learn continuously with the upcoming data. Since the new data does not have any label, multiple delayed learning concepts [99, 105, 215] are proposed to handle the non-negligible delay in data labeling. These methods are adequate for the scenario where labeling takes a constant time and latency is not a determining factor. For IoT data with variable constant drifts, cleaning and labeling may not take uniform time. Also, latency is one of the deciding factors for IoT-based ML applications. Thus, new sets of methods are essential for data labeling which considers the fluctuating IoT data with minimum possible delay.

5.3 Data Fusion

Data fusion aims to combine the data from multiple sources to provide more accurate and useful information. It offers numerous advantages for ML-based application by enhancing the data quality (finding the missing values), detecting any anomalies, conducting the prediction and finding any correlations among the distributed dataset [25, 158, 300]. However, there are multiple challenges in combining heterogeneous IoT data [4] such as data frequency, data imperfection, data correlation, data alignment and dynamic iterative process. To handle these challenges effectively, numerous data fusion methods are available in the literature. It is mainly categorized into *three* groups as given below.

5.3.1 *Probabilistic Data Fusion Algorithms.*

This group consists of the algorithms that use density function or probability distributions as a core method for data fusion. It includes Bayesian techniques [26], Markov models [156], evidential reasoning [297]and other methods. These methods are simple and widely used in different applications to express the co-relation and dependency between numerous datasets. However, there are certain drawbacks with probabilistic data fusion methods highlighted in [4]. First, it can not scale with the size and modality of the data. Second, uncertain and noisy data can not be handled properly. Finally, prior probabilities and density functions are difficult to obtain.

5.3.2 *Knowledge-based Data Fusion Algorithms.*

To overcome the uncertainty of data and increase the accuracy of fusion methods, knowledge-based data fusion methods accumulates knowledge from the imprecise big data and apply over the fusion process. Different aggregation techniques and ML methods are used for the data fusion process. For example, [22, 191] (supervised learning method) and [92, 316] (unsupervised learning method) are used to discover the distribution of the complex datasets. However, the complexity of this type of method is higher than the probabilistic methods. This class of method, thus, may consume more computing resources and cost more time to process.

5.3.3 *Evidence-based Data Fusion Algorithms.*

This group of methods is based on Demster-Shafer Theorem (DST) and recursive operations. As compared to probabilistic methods, where there are only two states (happening or non-happening) of an event, DST includes an unknown state to capture real-world uncertainty. [137, 236] are the applications of DST for data fusion. However, increasing the data evidence also increases the complexity of this method. Therefore, this method is not suitable for the applications running on less powerful computing resources.

5.4 Discussion

In this section, we reviewed core components in the data acquisition process and discussed how they can contribute to generation of high quality, ready-to-use data for IoT-ML application. Multiple research directions can be considered to leverage others' efforts, thereby improving the performance of ML model. 1): *Data reuse*: with the scaling of the data volume, past data is stored and usually ignored after use. However, it can be reused and mined for more values. For example it can be used for boosting semi-supervised data annotation [322], or it can be integrated with newly collected data for model training. 2): *Data re-organization*, there exist datasets for different tasks in similar areas. They may not be the same, but can be re-organized to extract the common distributions. Proper identification and extraction can be explored to save effort on data collection. 3): *Feature evolution* is also an important trait of streaming data as feature may appear and disappear over time. If a feature appears and is found to be relevant, it is necessary to incorporate that for the learning process. In this case, disappearance of a feature can be considered as a drift and the unavailability is treated as missing values. Ignoring this feature may lead to inaccurate prediction. Taking the relevancy of feature evolution for different problem domains. Other challenges that related to the unbalanced data have been discussed in §4.2.1 as well.

6 CONCLUSION

Growing numbers of internet-connected things (IoT) produce vast amounts of data, build applications and provide various services in domains such as smart cities, energy, mobility, and smart transportation. ML is becoming a preliminary technique for analyzing IoT data. It produces high-level abstraction and insight that is fed to the IoT systems for fine-tuning and improvement of the services. In this survey, we reviewed the characteristics of the IoT development lifecycle and the role of ML for individual steps. Specifically, we divided the development lifecycles into different modules and presented a novel taxonomy to characterize and analyze various techniques used to build an ML-based IoT application. In summary, this survey seeks to provide systematic and insightful information for researchers. It assists the development of future orchestration solutions by providing a holistic view on the current status of ML-based IoT application development, deriving key open research issues that were identified based on our critical review.

REFERENCES

[1] M. Abadi, P. Barham, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on OSDI.* 265–283.
[2] S. Achleitner, T. La Porta, et al. 2017. Adversarial network forensics in software defined networking. In *Proceedings of the Symposium on SDN Research.* ACM, 8–20.
[3] A. F. Aji and K. Heafield. 2017. Sparse Communication for Distributed Gradient Descent. In *EMNLP (2017).* 440–445.
[4] F. Alam, R. Mehmood, et al. 2017. Data fusion and IoT for smart ubiquitous environments: A survey. *IEEE Access* 5 (2017), 9533–9554.
[5] S. A. Alasadi and W. S. Bhaya. 2017. Review of data preprocessing techniques in data mining. *Journal of Engineering and Applied Sciences* 12, 16 (2017), 4102–4107.
[6] D. Alistarh, D. Grubic, et al. 2017. QSGD: Communication-efficient SGD via gradient quantization and encoding. In *NIPS (2017).* 1709–1720.
[7] F. A. Aoudia, M. Gautier, et al. 2018. RLMan: an energy manager based on reinforcement learning for energy harvesting wireless sensor networks. *IEEE-TGCN* (2018).
[8] J. Appleyard, T. Kocisky, et al. 2016. Optimizing performance of recurrent neural networks on gpus. *arXiv preprint arXiv:1604.01946* (2016).
[9] K. Arulkumaran, M. P. Deisenroth, et al. 2017. A brief survey of deep reinforcement learning. *arXiv preprint arXiv:1708.05866* (2017).
[10] G. Ateniese, G. Felici, et al. 2013. Hacking smart machines with smarter ones: How to extract meaningful data from machine learning classifiers. *arXiv* (2013).
[11] S. Ayache and G. Quénot. 2007. Evaluation of active learning strategies for video indexing. *Signal Processing: Image Communication* 22, 7-8 (2007), 692–704.
[12] J. Ba, R. Grosse, et al. 2016. Distributed second-order optimization using Kronecker-factored approximations. (2016).
[13] B. Baker, O. Gupta, et al. 2017. Accelerating neural architecture search using performance prediction. *arXiv preprint arXiv:1705.10823* (2017).
[14] A. K. Balan, V. Rathod, et al. 2015. Bayesian dark knowledge. In *NIPS (2015).* 3438–3446.

[15] P. Baldi. 2012. Autoencoders, unsupervised learning, and deep architectures. In *Proceedings of ICML workshop on unsupervised and transfer learning*. 37–49.

[16] R. Battiti. 1994. Using mutual information for selecting features in supervised neural net learning. *IEEE Transactions on neural networks* 5, 4 (1994), 537–550.

[17] A. Beck, A. Nedić, et al. 2014. An $o(1/k)$ gradient method for network resource allocation problems. *IEEE Transactions on Control of Network Systems* 1, 1 (2014), 64–73.

[18] T. Ben-Nun, E. Levy, et al. 2015. Memory access patterns: the missing piece of the multi-GPU puzzle. In *SC'15*. IEEE, 1–12.

[19] G. M. Benedek and A. Itai. 1991. Learnability with respect to fixed distributions. *Theoretical Computer Science* 86, 2 (1991), 377–389.

[20] Y. Bengio. 2012. Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade*. Springer, 437–478.

[21] J. L. Berral, Í. Goiri, et al. 2010. Towards energy-aware scheduling in data centers using machine learning. In *E-Energy (2010)*. ACM, 215–224.

[22] B. Bigdeli and P. Reinartz. 2015. Fusion of hyperspectral and LIDAR data using decision template-based fuzzy multiple classifier system. *INT J APPL EARTH OBS* (2015).

[23] B. Biggio, B. Nelson, et al. 2012. Poisoning attacks against support vector machines. *arXiv preprint arXiv:1206.6389* (2012).

[24] B. Biggio, K. Rieck, et al. 2014. Poisoning behavioral malware clustering. In *Proceedings of the 2014 workshop on artificial intelligent and security workshop*. ACM, 27–36.

[25] F. H. Bijarbooneh, W. Du, et al. 2016. Cloud-assisted data fusion and sensor selection for internet of things. *IEEE Internet of Things Journal* 3, 3 (2016), 257–268.

[26] T. A. Biresaw, A. Cavallaro, et al. 2015. Tracker-level fusion for robust Bayesian visual tracking. *IEEE Transactions on Circuits and Systems for Video Technology* 25, 5 (2015).

[27] A. Blázquez-García, A. Conde, et al. 2020. A review on outlier/anomaly detection in time series data. *arXiv preprint arXiv:2002.04236* (2020).

[28] M. Boehm, M. W. Dusenberry, et al. 2016. Systemml: Declarative machine learning on spark. *Proceedings of the VLDB Endowment* 9, 13 (2016), 1425–1436.

[29] K. Bonawitz, H. Eichner, et al. 2019. Towards federated learning at scale: System design. *arXiv preprint arXiv:1902.01046* (2019).

[30] K. Bonawitz, F. Salehi, et al. 2019. Federated learning with autotuned communication-efficient secure aggregation. *arXiv preprint arXiv:1912.00131* (2019).

[31] L. Bottou. 2010. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*. Springer, 177–186.

[32] L. Bottou. 2012. Stochastic gradient descent tricks. In *Neural networks: Tricks of the trade*. Springer, 421–436.

[33] L. Bottou, F. E. Curtis, et al. 2018. Optimization methods for large-scale machine learning. *Siam Review* 60, 2 (2018), 223–311.

[34] O. Bousquet, S. Boucheron, et al. 2003. Introduction to statistical learning theory. In *Summer School on Machine Learning*. Springer, 169–207.

[35] J. K. Bradley, A. Kyrola, et al. 2011. Parallel coordinate descent for l1-regularized loss minimization. *arXiv preprint arXiv:1105.5379* (2011).

[36] L. Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.

[37] L. Breiman. 2017. *Classification and regression trees*. Routledge.

[38] U. Breitenbucher, K. Kepes, et al. 2017. Declarative vs. Imperative: How to Model the Automated Deployment of IoT Applications? *SummerSOC'17* (2017), 18–27.

[39] A. Brock, T. Lim, et al. 2017. SMASH: one-shot model architecture search through hypernetworks. *arXiv preprint arXiv:1708.05344* (2017).

[40] R. H. Byrd, S. L. Hansen, et al. 2016. A stochastic quasi-Newton method for large-scale optimization. *SIAM Journal on Optimization* 26, 2 (2016), 1008–1031.

[41] B. Cabé, E. I. W. Group, et al. 2018. IoT Developer Survey 2018. *SlideShare, April* 13 (2018).

[42] M. Castro, B. Liskov, et al. 1999. Practical Byzantine fault tolerance. In *OSDI*, Vol. 99. 173–186.

[43] Z. B. Celik, L. Babun, et al. 2018. Sensitive information tracking in commodity IoT. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 1687–1704.

[44] N. V. Chawla, N. Japkowicz, et al. 2004. Special issue on learning from imbalanced data sets. *ACM Sigkdd Explorations Newsletter* 6, 1 (2004), 1–6.

[45] K. Chellapilla, S. Puri, et al. 2006. High performance convolutional neural networks for document processing. In *IWFHR 10*. Suvisoft.

[46] C.-C. Chen, C.-L. Yang, et al. 2018. Efficient and robust parallel dnn training through model parallelism on multi-gpu platform. *arXiv preprint arXiv:1809.02839* (2018).

[47] J. Chen, X. Pan, et al. 2016. Revisiting distributed synchronous SGD. *arXiv preprint arXiv:1604.00981* (2016).

[48] L. Chen, J. Lu, et al. 2018. Part-activated deep reinforcement learning for action prediction. In *Proceedings of the European Conference on Computer Vision (ECCV)*. 421–436.

[49] T. Chen, T. Moreau, et al. 2018. {TVM}: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on OSDI*. 578–594.

[50] X. Chen, C. Liu, et al. 2017. Targeted backdoor attacks on deep learning systems using data poisoning. *arXiv preprint arXiv:1712.05526* (2017).

[51] Y. Cheng, D. Wang, et al. 2017. A survey of model compression and acceleration for deep neural networks. *arXiv preprint arXiv:1710.09282* (2017).

[52] S. Chetlur, C. Woolley, et al. 2014. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* (2014).

[53] T. Chilimbi, Y. Suzue, et al. 2014. Project adam: Building an efficient and scalable deep learning training system. In *11th USENIX Symposium on OSDI*. 571–582.

[54] C.-T. Chin, K. Mehrotra, et al. 1994. Training techniques to obtain fault-tolerant neural networks. In *FTCS*. IEEE, 360–369.

[55] C.-T. Chiu, K. Mehrotra, et al. 1993. Robustness of feedforward neural networks. In *Neural Networks, 1993., IEEE International Conference on*. IEEE, 783–788.

[56] F. Chollet. 2017. Xception: Deep learning with depthwise separable convolutions. In *CVPR (2017)*. 1251–1258.

[57] P. Chopra and S. K. Yadav. 2015. Fault detection and classification by unsupervised feature extraction and dimensionality reduction. *Complex & Intelligent Systems* (2015).

[58] C.-T. Chu, S. K. Kim, et al. 2007. Map-reduce for machine learning on multicore. In *NIPS (2007)*. 281–288.

[59] L.-C. Chu and B. W. Wah. 1990. Fault tolerant neural networks with hybrid redundancy. In *Neural Networks, 1990., 1990 IJCNN International Joint Conference on*. IEEE.

[60] I. Colin, L. Dos Santos, et al. 2019. Theoretical Limits of Pipeline Parallel Optimization and Application to Distributed Deep Learning. In *NIPS (2019)*. 12350–12359.

[61] G. Cormode, M. Garofalakis, et al. 2011. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends® in Databases* 4, 1–3 (2011), 1–294.

[62] N. Corporation. 2015. NVIDIA Collective Communications Library (NCCL). Retrieved March 2, 2020 from https://developer.nvidia.com/nccl

[63] C. Cortes and V. Vapnik. 1995. Support-vector networks. *Machine learning* 20, 3 (1995), 273–297.

[64] M. Courbariaux, I. Hubara, et al. 2016. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv* (2016).

[65] D. Crankshaw, G.-E. Sela, et al. 2018. InferLine: ML Inference Pipeline Composition Framework. *arXiv preprint arXiv:1812.01776* (2018).

[66] D. Crankshaw, X. Wang, et al. 2017. Clipper: A low-latency online prediction serving system. In *14th USENIX Symposium on NSDI*. 613–627.

[67] H. Cui, H. Zhang, et al. 2016. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *EuroSys (2016)*. 1–16.

[68] A. C. F. da Silva, U. Breitenbücher, et al. 2017. Internet of Things Out of the Box: Using TOSCA for Automating the Deployment of IoT Environments.. In *CLOSER*. 330–339.

[69] M. Dash and H. Liu. 2003. Consistency-based search in feature selection. *Artificial intelligence* 151, 1-2 (2003), 155–176.

[70] R. S. M. de Barros and S. G. T. de Carvalho Santos. 2019. An overview and comprehensive comparison of ensembles for concept drift. *Information Fusion* 52 (2019), 213–244.

[71] C. M. De Sa, C. Zhang, et al. 2015. Taming the wild: A unified analysis of hogwild-style algorithms. In *NIPS (2015)*. 2674–2682.

[72] J. Dean, G. Corrado, et al. 2012. Large scale distributed deep networks. In *NIPS (2012)*. 1223–1231.

[73] O. Dekel, R. Gilad-Bachrach, et al. 2012. Optimal distributed online prediction using mini-batches. *JMLR* 13, Jan (2012), 165–202.

[74] J. Deng, W. Dong, et al. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 248–255.

[75] S. Devi and T. Neetha. 2017. Machine Learning based traffic congestion prediction in a IoT based Smart City. (2017).

[76] G. Diamos, S. Sengupta, et al. 2016. Persistent rnns: Stashing recurrent weights on-chip. In *International Conference on Machine Learning*. 2024–2033.

[77] J.-D. Dong, A.-C. Cheng, et al. 2018. Dpp-net: Device-aware progressive search for pareto-optimal neural architectures. In *Proceedings of the ECCV*. 517–531.

[78] W. Dong, C. Chen, et al. 2013. D2: Anomaly detection and diagnosis in networked embedded systems by program profiling and symptom mining. In *RTSS*. IEEE, 202–211.
[79] N. Dryden, N. Maruyama, et al. 2019. Improving strong-scaling of CNN training by exploiting finer-grained parallelism. In *IPDPS (2019)*. IEEE, 210–220.
[80] N. Dryden, N. Maruyama, et al. 2019. Channel and filter parallelism for large-scale CNN training. In *SC'19*. 1–20.
[81] N. Dryden, T. Moon, et al. 2016. Communication quantization for data-parallel training of deep neural networks. In *2016 2nd Workshop on MLHPC*. IEEE, 1–8.
[82] J. Duchi, E. Hazan, et al. 2011. Adaptive subgradient methods for online learning and stochastic optimization. *JMLR* 12, Jul (2011), 2121–2159.
[83] S. Egea, A. R. Mañez, et al. 2017. Intelligent IoT traffic classification using novel search strategy for fast-based-correlation feature selection in industrial environments. *IEEE Internet of Things Journal* 5, 3 (2017), 1616–1624.
[84] A. Elk. 2019. Distributed Machine Learning Toolkit: BIG DATA, BIG MODEL, FLEXIBILITY, EFFICIENCY. Retrieved March 7, 2019 from http://www.dmtk.io
[85] T. Elsken, J. H. Metzen, et al. 2018. Multi-objective architecture search for cnns. *arXiv preprint arXiv:1804.09081* 2 (2018).
[86] M. D. Emmerson et al. 1993. Determining and improving the fault tolerance of multilayer perceptrons in a pattern-recognition application. *IEEE Trans. Neural Netw.* (1993).
[87] T. Erpek, Y. E. Sagduyu, et al. 2018. Deep learning for launching and mitigating wireless jamming attacks. *TCCN* (2018).
[88] S. Eswaran, A. Misra, et al. 2012. Utility-based bandwidth adaptation in mission-oriented wireless sensor networks. *TOSN* (2012).
[89] M. Everingham, L. Van Gool, et al. 2010. The Pascal Visual Object Classes (VOC) Challenge. *International Journal of Computer Vision* 88, 2 (June 2010), 303–338.
[90] H. I. Fawaz, G. Forestier, et al. 2019. Deep learning for time series classification: a review. *Data Mining and Knowledge Discovery* 33, 4 (2019), 917–963.
[91] G. Frewat, C. Baroud, et al. 2016. Android voice recognition application with multi speaker feature. In *2016 18th MELECON*. IEEE, 1–5.
[92] C. E. Fuss, A. A. Berg, et al. 2016. DEM Fusion using a modified k-means clustering algorithm. *International journal of digital earth* 9, 12 (2016), 1242–1255.
[93] J. a. Gama, I. Žliobaitundefined, et al. 2014. A Survey on Concept Drift Adaptation. *ACM Comput. Surv.* 46, 4, Article Article 44 (March 2014), 37 pages.
[94] P. Garraghan, R. Yang, et al. 2018. Emergent failures: Rethinking cloud reliability at scale. *IEEE Cloud Computing* 5, 5 (2018), 12–21.
[95] A. L. Gaunt, M. A. Johnson, et al. 2017. AMPNet: Asynchronous model-parallel training for dynamic neural networks. *arXiv preprint arXiv:1705.09786* (2017).
[96] A. Gholami, A. Azad, et al. 2018. Integrated model, batch, and domain parallelism in training neural networks. In *SPAA'18*. 77–86.
[97] R. Gilyazev and D. Y. Turdakov. 2018. Active Learning and Crowdsourcing: A Survey of Optimization Methods for Data Labeling. *PROGRAM COMPUT SOFT+* (2018).
[98] D. E. Goldberg. 1989. *Genetic Algorithms in Search, Optimization and Machine Learning* (1st ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
[99] H. M. Gomes, A. Bifet, et al. 2017. Adaptive random forests for evolving data stream classification. *Machine Learning* 106, 9-10 (2017), 1469–1495.
[100] I. Goodfellow, J. Pouget-Abadie, et al. 2014. Generative adversarial nets. In *NIPS (2014)*. 2672–2680.
[101] I. J. Goodfellow, J. Shlens, et al. 2014. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572* (2014).
[102] P. Goyal, P. Dollár, et al. 2017. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677* (2017).
[103] J. Granjal, E. Monteiro, et al. 2010. A secure interconnection model for IPv6 enabled wireless sensor networks. In *2010 IFIP Wireless Days*. IEEE, 1–6.
[104] A. Graves, A. rahman Mohamed, et al. 2013. Speech recognition with deep recurrent neural networks. *ICASSP* (2013), 6645–6649.
[105] M. Grzenda, H. M. Gomes, et al. 2019. Delayed labelling evaluation for data streams. *Data Mining and Knowledge Discovery* (2019), 1–30.
[106] T. Gu, B. Dolan-Gavitt, et al. 2017. Badnets: Identifying vulnerabilities in the machine learning model supply chain. *arXiv preprint arXiv:1708.06733* (2017).
[107] L. Guan, W. Yin, et al. 2019. XPipe: Efficient Pipeline Model Parallelism for Multi-GPU DNN Training. *arXiv preprint arXiv:1911.04610* (2019).
[108] Y. Guan and T. Plötz. 2017. Ensembles of deep lstm learners for activity recognition using wearables. *IMWUT* 1, 2 (2017), 11.
[109] S. Gupta, A. Agrawal, et al. 2015. Deep learning with limited numerical precision. In *ICML*. 1737–1746.
[110] R. Gutierrez-Osuna. 2002. Pattern analysis for machine olfaction: a review. *IEEE Sensors journal* (2002).
[111] H. Haddad, A. Dehghan, et al. 2018. A deep Recurrent Neural Network based approach for Internet of Things malware threat hunting. *FUTURE GENER COMP SY* (2018).
[112] M. A. Hall. 2000. Correlation-based feature selection of discrete and numeric class machine learning. (2000).
[113] Y. Han, D. Hubczenko, et al. 2019. Adversarial Reinforcement Learning under Partial Observability in Software-Defined Networking. *arXiv:1902.09062* (2019).
[114] P. Hansen, N. Mladenović, et al. 2010. Variable neighbourhood search: methods and applications. *Annals of Operations Research* 175, 1 (2010), 367–407.
[115] A. Harlap, D. Narayanan, et al. 2018. Pipedream: Fast and efficient pipeline parallel dnn training. *arXiv preprint arXiv:1806.03377* (2018).
[116] J. A. Hartigan and M. A. Wong. 1979. Algorithm AS 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 28, 1 (1979).
[117] K. He, G. Gkioxari, et al. 2017. Mask R-CNN. *2017 IEEE International Conference on Computer Vision (ICCV)* (2017), 2980–2988.
[118] X. He, D. Mudigere, et al. 2017. Distributed hessian-free optimization for deep neural network. In *Workshops at the Thirty-First AAAI*.
[119] G. Hinton. 2019. RMSprop. Retrieved June 7, 2019 from http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf
[120] G. Hinton, O. Vinyals, et al. 2015. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531* (2015).
[121] Q. Ho, J. Cipar, et al. 2013. More effective distributed ml via a stale synchronous parallel parameter server. In *NIPS (2013)*. 1223–1231.
[122] S. Hochreiter and J. Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.
[123] S. C. Hoi, R. Jin, et al. 2006. Batch mode active learning and its application to medical image classification. In *Proceedings of the 23rd ICML*. ACM, 417–424.
[124] S. C. Hoi, D. Sahoo, et al. 2018. Online learning: A comprehensive survey. *arXiv preprint arXiv:1802.02871* (2018).
[125] L. Hou, R. Zhang, et al. 2019. Analysis of Quantized Models. In *International Conference on Learning Representations*. https://openreview.net/forum?id=ryM_IoAqYX
[126] A. G. Howard, M. Zhu, et al. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).
[127] K. Hsieh, A. Harlap, et al. 2017. Gaia: Geo-distributed machine learning approaching {LAN} speeds. In *14th Symposium on NSDI*. 629–647.
[128] C.-H. Hsu, S.-H. Chang, et al. 2018. Monas: Multi-objective neural architecture search using reinforcement learning. *arXiv preprint arXiv:1806.10332* (2018).
[129] H. Hu, M. Kantardzic, et al. 2019. No Free Lunch Theorem for concept drift detection in streaming data classification: A review. *WIRES DATA MIN KNOWL* (2019), e1327.
[130] T. Hu and Y. Fei. 2010. QELAR: A machine-learning-based adaptive routing protocol for energy-efficient and lifetime-extended underwater sensor networks. *TMC* (2010).
[131] G. Huang, S. Liu, et al. 2018. Condensenet: An efficient densenet using learned group convolutions. In *CVPR (2018)*. 2752–2761.
[132] Y. Huang, Y. Cheng, et al. 2019. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *NIPS (2019)*. 103–112.
[133] I. Hubara, M. Courbariaux, et al. 2017. Quantized neural networks: Training neural networks with low precision weights and activations. *JMLR* 18, 1 (2017), 6869–6898.
[134] Z. Huo, B. Gu, et al. 2018. Decoupled parallel backpropagation with convergence guarantee. *arXiv preprint arXiv:1804.10574* (2018).
[135] S. Ioffe and C. Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167* (2015).
[136] J. Ivanecky and S. Mehlhase. 2012. An in-car speech recognition system for disabled drivers. In *International Conference on Text, Speech and Dialogue*. Springer, 505–512.
[137] H. Jamshidi, T. Lukaszewicz, et al. 2011. Fusion of digital map traffic signs and camera-detected signs. In *ICSPCS*. IEEE, 1–7.
[138] S. Jeaugey. 2017. NCCL 2.0. Retrieved March 2, 2020 from http://on-demand.gputechconf.com/gtc/2017/presentation/s7155-jeaugey-nccl.pdf
[139] D. N. Jha, P. Michalak, et al. 2019. Multi-objective Deployment of Data Analysis Operations in Heterogeneous IoT Infrastructure. *IEEE T IND INFORM* (2019).

[140] Z. Jia, M. Zaharia, et al. 2018. Beyond data and model parallelism for deep neural networks. *arXiv preprint arXiv:1807.05358* (2018).

[141] P. Jin, B. Ginsburg, et al. 2018. Spatially parallel convolutions. (2018).

[142] A. J. Joshi, F. Porikli, et al. 2009. Multi-class active learning for image classification. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 2372–2379.

[143] D. Jouan-Rimbaud, D.-L. Massart, et al. 1995. Genetic algorithms as a tool for wavelength selection in multivariate calibration. *Analytical Chemistry* 67, 23 (1995), 4295–4301.

[144] P. Kairouz, H. B. McMahan, et al. 2019. Advances and open problems in federated learning. *arXiv preprint arXiv:1912.04977* (2019).

[145] R. Kemker, M. McClure, et al. 2018. Measuring catastrophic forgetting in neural networks. In *AAAI*.

[146] H. Kim, M. Kim, et al. 2018. Nsml: Meet the mlaas platform with a real-world case study. *arXiv preprint arXiv:1810.09957* (2018).

[147] J. K. Kim, Q. Ho, et al. 2016. STRADS: a distributed framework for scheduled model parallel machine learning. In *EuroSys (2016)*. 1–16.

[148] D. P. Kingma and J. Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[149] C. Kolias, V. Kolias, et al. 2017. TermID: A distributed swarm intelligence-based approach for wireless intrusion detection. *ISECURE* (2017).

[150] V. R. Konda and J. N. Tsitsiklis. 2000. Actor-critic algorithms. In *NIPS (2000)*. 1008–1014.

[151] J. Konečnỳ, H. B. McMahan, et al. 2016. Federated optimization: Distributed machine learning for on-device intelligence. *arXiv preprint arXiv:1610.02527* (2016).

[152] W. Kong, Z. Y. Dong, et al. 2017. Short-term residential load forecasting based on LSTM recurrent neural network. *IEEE Trans Smart Grid* 10, 1 (2017), 841–851.

[153] N. Kothari, T. Millstein, et al. 2008. Deriving state machines from TinyOS programs using symbolic execution. In *IPSN*. IEEE Computer Society.

[154] A. Krizhevsky. 2014. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997* (2014).

[155] V. Krunic, E. Trumpler, et al. 2007. NodeMD: Diagnosing node-level faults in remote wireless sensor systems. In *MobiSys*. ACM, 43–56.

[156] P. Kumar, H. Gauba, et al. 2017. Coupled HMM-based multi-sensor data fusion for sign language recognition. *Pattern Recognition Letters* 86 (2017), 1–8.

[157] A. Kurakin, I. Goodfellow, et al. 2016. Adversarial machine learning at scale. *arXiv preprint arXiv:1611.01236* (2016).

[158] B. P. L. Lau, S. H. Marakkalage, et al. 2019. A survey of data fusion in smart city applications. *Information Fusion* 52 (2019), 357–374.

[159] V. Lebedev, Y. Ganin, et al. 2014. Speeding-up convolutional neural networks using fine-tuned cp-decomposition. *arXiv preprint arXiv:1412.6553* (2014).

[160] S. Lee, J. K. Kim, et al. 2014. On model parallelization and scheduling strategies for distributed machine learning. In *NIPS (2014)*. 2834–2842.

[161] H. Li, W. Ouyang, et al. 2016. Multi-bias non-linear activation in deep neural networks. In *International conference on machine learning*. 221–229.

[162] J. Li, H. Gao, et al. 2018. Deep reinforcement learning based computation offloading and resource allocation for MEC. In *WCNC*. IEEE, 1–6.

[163] J. Li, W. Monroe, et al. 2016. Deep reinforcement learning for dialogue generation. *arXiv preprint arXiv:1606.01541* (2016).

[164] L. Li, Y. Lv, et al. 2016. Traffic signal timing via deep reinforcement learning. *IEEE/CAA Journal of Automatica Sinica* 3, 3 (2016), 247–254.

[165] L. Li, W. Xu, et al. 2019. Rsa: Byzantine-robust stochastic aggregation methods for distributed learning from heterogeneous datasets. In *AAAI*, Vol. 33. 1544–1551.

[166] M. Li, D. G. Andersen, et al. 2014. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on OSDI*. 583–598.

[167] M. Li, D. G. Andersen, et al. 2014. Communication efficient distributed machine learning with the parameter server. In *NIPS (2014)*.

[168] R. Li, C. Zhang, et al. 2019. Learning Driven Mobility Control of Airborne Base Stations in Emergency Networks. *ACM SIGMETRICS Performance Evaluation Review* (2019).

[169] J. Lin, W. Yu, et al. 2017. A survey on internet of things: Architecture, enabling technologies, security and privacy, and applications. *IEEE Internet Things J.* (2017).

[170] T. Lin, S. U. Stich, et al. 2018. Don't Use Large Mini-Batches, Use Local SGD. *arXiv preprint arXiv:1808.07217* (2018).

[171] Y. Lin, S. Han, et al. 2017. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887* (2017).

[172] C. Liu, B. Zoph, et al. 2018. Progressive neural architecture search. In *Proceedings of the European Conference on Computer Vision (ECCV)*. 19–34.

[173] H. Liu, K. Simonyan, et al. 2017. Hierarchical representations for efficient architecture search. *arXiv preprint arXiv:1711.00436* (2017).

[174] L. Liu, M. Zhang, et al. 2014. A survey on workflow management and scheduling in cloud computing. In *CCGrid 2014*. IEEE, 837–846.

[175] M. Lopez-Martin, B. Carro, et al. 2017. Conditional variational autoencoder for prediction and feature recovery applied to intrusion detection in iot. *Sensors* (2017).

[176] Q. Lu, T. Peng, et al. 2010. Utility-based resource allocation in uplink of OFDMA-based cognitive radio networks. *International Journal of Communication Systems* (2010).

[177] M.-T. Luong, H. Pham, et al. 2015. Effective approaches to attention-based neural machine translation. *EMNLP* (2015).

[178] N. Ma, X. Zhang, et al. 2018. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *Proceedings of the European Conference on Computer Vision (ECCV)*.

[179] D. J. MacKay and D. J. Mac Kay. 2003. *Information theory, inference and learning algorithms*. Cambridge university press.

[180] R. Madan, J. Borran, et al. 2010. Cell association and interference coordination in heterogeneous LTE-A cellular networks. *J-SAC* (2010).

[181] R. Madan, S. P. Boyd, et al. 2010. Fast algorithms for resource allocation in wireless cellular networks. *IEEE/ACM Transactions on Networking (TON)* 18, 3 (2010), 973–984.

[182] H. Mao, M. Alizadeh, et al. 2016. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*. ACM.

[183] H. Mao, R. Netravali, et al. 2017. Neural adaptive video streaming with pensieve. In *SIGCOMM*. ACM.

[184] A. Marcano-Cedeno et al. 2010. Feature selection using sequential forward selection and classification applying artificial metaplasticity neural network. In *IECON*. IEEE.

[185] M. Mathieu, M. Henaff, et al. 2013. Fast training of convolutional networks through ffts. *arXiv preprint arXiv:1312.5851* (2013).

[186] W. McColl. 1995. Bulk synchronous parallel computing. *Abstract Machine Models for Highly Parallel Computers, Oxford University Press, Oxford* (1995).

[187] R. McDonald, K. Hall, et al. 2010. Distributed training strategies for the structured perceptron. In *NAACL-HLT*. Association for Computational Linguistics, 456–464.

[188] H. B. McMahan, E. Moore, et al. 2016. Communication-efficient learning of deep networks from decentralized data. *arXiv preprint arXiv:1602.05629* (2016).

[189] H. B. McMahan, D. Ramage, et al. 2017. Learning differentially private recurrent language models. *ICLR 2018* (2017).

[190] X. Meng, J. Bradley, et al. 2016. Mllib: Machine learning in apache spark. *JMLR* 17, 1 (2016), 1235–1241.

[191] A. Merentitis and C. Debes. 2015. Automatic fusion and classification using random forests and features extracted with deep learning. In *IGARSS*. IEEE, 2943–2946.

[192] E. M. E. Mhamdi and R. Guerraoui. 2017. When Neurons Fail. In *IPDPS (2017)*. 1028–1037. https://doi.org/10.1109/IPDPS.2017.66

[193] A. Mirhoseini, A. Goldie, et al. 2018. A hierarchical model for device placement. (2018).

[194] A. Mirhoseini, H. Pham, et al. 2017. Device placement optimization with reinforcement learning. In *ICML (2017)*. JMLR. org, 2430–2439.

[195] T. M. Mitchell. 1999. Machine learning and data mining. *Commun. ACM* 42, 11 (1999), 30–36.

[196] V. Mnih, A. P. Badia, et al. 2016. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*. 1928–1937.

[197] M. Mohammadi, A. Al-Fuqaha, et al. 2018. Deep learning for IoT big data and streaming analytics: A survey. *IEEE Communications Surveys & Tutorials* 20, 4 (2018).

[198] P. Moritz, R. Nishihara, et al. 2016. A linearly-convergent stochastic L-BFGS algorithm. In *Artificial Intelligence and Statistics*. 249–258.

[199] P. Moritz, R. Nishihara, et al. 2018. Ray: A distributed framework for emerging {AI} applications. In *13th USENIX Symposium on OSDI*. 561–577.

[200] M. S. Munir, S. F. Abedin, et al. 2017. RNN based Energy Demand Prediction for Smart-Home in Smart-Grid Framework.

[201] A. Murad, F. A. Kraemer, et al. 2019. Autonomous Management of Energy-Harvesting IoT Nodes Using Deep Reinforcement Learning. *arXiv:1905.04181* (2019).

[202] P. Nakkiran, G. Kaplun, et al. 2019. Deep double descent: Where bigger models and more data hurt. *arXiv preprint arXiv:1912.02292* (2019).

[203] P. Naraei, A. Abhari, et al. 2016. Application of multilayer perceptron neural networks and support vector machines in classification of healthcare data. In *FTC*. 848–852.

[204] B. Nelson, B. I. Rubinstein, et al. 2012. Query strategies for evading convex-inducing classifiers. *JMLR* 13, May (2012), 1293–1332.

[205] D. Oh and I. Yun. 2018. Residual error based anomaly detection using auto-encoder in smd machine sound. *Sensors* 18, 5 (2018), 1308.

[206] K. Osawa, Y. Tsuji, et al. 2018. Second-order optimization method for large mini-batch: Training resnet-50 on imagenet in 35 epochs. *arXiv preprint:1811.12019* (2018).

[207] X. Pan, M. Lam, et al. 2016. Cyclades: Conflict-free asynchronous machine learning. In *NIPS (2016)*. 2568–2576.

[208] N. Papernot, P. McDaniel, et al. 2017. Practical black-box attacks against machine learning. In *ASIACCS 2017*. ACM, 506–519.

[209] N. Papernot, P. McDaniel, et al. 2016. The limitations of deep learning in adversarial settings. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE.

[210] G. I. Parisi, R. Kemker, et al. 2019. Continual lifelong learning with neural networks: A review. *Neural Networks* (2019).

[211] R. Parr and S. J. Russell. 1998. Reinforcement learning with hierarchies of machines. In *NIPS*.

[212] Z. Peng, X. Chen, et al. 2018. AXNet: ApproXimate computing using an end-to-end trainable neural network. In *ICCAD*. ACM, 11.

[213] A. Petrowski, G. Dreyfus, et al. 1993. Performance analysis of a pipelined backpropagation parallel algorithm. *IEEE Transactions on Neural Networks* 4, 6 (1993), 970–981.

[214] H. Pham, M. Y. Guan, et al. 2018. Efficient neural architecture search via parameter sharing. *arXiv preprint arXiv:1802.03268* (2018).

[215] J. Plasse and N. Adams. 2016. Handling delayed labels in temporally evolving data streams. In *2016 IEEE International Conference on Big Data (Big Data)*. IEEE, 2416–2424.

[216] D. Povey, X. zhang, et al. 2017. Parallel training of DNNs with Natural Gradient and Parameter Averaging. *ICLR 2015* - (Aug 2017).

[217] P. W. Protzel, D. L. Palumbo, et al. 1993. Performance and fault-tolerance of neural networks for optimization. *IEEE transactions on Neural Networks* 4, 4 (1993), 600–614.

[218] N. Qian. 1999. On the momentum term in gradient descent learning algorithms. *Neural networks* 12, 1 (1999), 145–151.

[219] A. Qiao, B. Aragam, et al. 2018. Fault Tolerance in Iterative-Convergent Machine Learning. *arXiv preprint arXiv:1810.07354* (2018).

[220] J. Quackenbush. 2002. Microarray data normalization and transformation. *Nature genetics* 32, 4s (2002), 496.

[221] D. Quillen, E. Jang, et al. 2018. Deep reinforcement learning for vision-based robotic grasping: A simulated comparative evaluation of off-policy methods. In *ICRA (2018)*.

[222] J. R. Quinlan. 1986. Induction of decision trees. *Machine learning* 1, 1 (1986), 81–106.

[223] J. R. Quinlan. 2014. *C4. 5: programs for machine learning*. Elsevier.

[224] R. Raina, A. Madhavan, et al. 2009. Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th annual ICML*. ACM.

[225] N. Ramakrishnan and T. Soni. 2018. Network Traffic Prediction Using Recurrent Neural Networks. In *2018 17th IEEE ICMLA*. IEEE.

[226] B. Randell. 1975. System structure for software fault tolerance. *Ieee transactions on software engineering* 2 (1975), 220–232.

[227] M. Rastegari, V. Ordonez, et al. 2016. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*. Springer.

[228] E. Real, A. Aggarwal, et al. 2019. Regularized evolution for image classifier architecture search. In *AAAI*.

[229] B. Recht, C. Re, et al. 2011. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS (2011)*. 693–701.

[230] L. Ren, X. Yuan, et al. 2018. Deep reinforcement learning with iterative shift for visual tracking. In *ECCV (2018)*. 684–700.

[231] K. Rieck, P. Trinius, et al. 2011. Automatic analysis of malware behavior using machine learning. *Journal of Computer Security* 19, 4 (2011), 639–668.

[232] H. Röger and R. Mayer. 2019. A Comprehensive Survey on Parallelization and Elasticity in Stream Processing. *ACM Computing Surveys (CSUR)* 52, 2 (2019), 36.

[233] R. Roman, J. Zhou, et al. 2013. On the features and challenges of security and privacy in distributed internet of things. *Computer Networks* 57, 10 (2013), 2266–2279.

[234] A. Romero, N. Ballas, et al. 2014. Fitnets: Hints for thin deep nets. *arXiv preprint arXiv:1412.6550* (2014).

[235] F. Ruelens, B. J. Claessens, et al. 2016. Residential demand response of thermostatically controlled loads using batch reinforcement learning. *IEEE Trans Smart Grid* (2016).

[236] V. Saeidi et al. 2014. Fusion of airborne lidar with multispectral spot 5 image for enhancement of feature extraction using dempster–shafer theory. *TGRS* (2014).

[237] M. Sandler, A. Howard, et al. 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In *CVPR (2018)*.

[238] J. Schulman, F. Wolski, et al. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).

[239] F. Seide, H. Fu, et al. 2014. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In *INTERSPEECH (2014)*.

[240] A. Sergeev and M. Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799* (2018).

[241] B. Settles. 2012. Active learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning* 6, 1 (2012), 1–114.

[242] A. Shafahi, W. R. Huang, et al. 2018. Poison frogs! targeted clean-label poisoning attacks on neural networks. In *NIPS (2018)*.

[243] W. Shang, K. Sohn, et al. 2016. Understanding and improving convolutional neural networks via concatenated rectified linear units. In *ICML*. 2217–2225.

[244] N. Shazeer, Y. Cheng, et al. 2018. Mesh-TensorFlow: Deep Learning for Supercomputers. In *Neural Information Processing Systems*.

[245] W. Shi, J. Cao, et al. 2016. Edge computing: Vision and challenges. *IEEE internet of things journal* 3, 5 (2016), 637–646.

[246] J. Shlens. 2014. A tutorial on principal component analysis. *arXiv preprint arXiv:1404.1100* (2014).

[247] R. Shokri, M. Stronati, et al. 2017. Membership inference attacks against machine learning models. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 3–18.

[248] A. Shustanov and P. Yakimov. 2017. CNN design for real-time traffic sign recognition. *Procedia engineering* 201 (2017), 718–725.

[249] S. Sicari, A. Rizzardi, et al. 2015. Security, privacy and trust in Internet of Things: The road ahead. *Computer networks* 76 (2015), 146–164.

[250] L. Sifre and S. Mallat. 2014. Rigid-motion scattering for image classification. *Ph. D. dissertation* (2014).

[251] A. K. Sikder, H. Aksu, et al. 2017. 6thsense: A context-aware sensor-based attack detector for smart devices. In *26th {USENIX} Security Symposium*. 397–414.

[252] K. J. Singh and D. S. Kapoor. 2017. Create Your Own Internet of Things: A survey of IoT platforms. *IEEE Consumer Electronics Magazine* 6, 2 (2017), 57–68.

[253] S. Singh and I. Chana. 2016. A survey on resource scheduling in cloud computing: Issues and challenges. *Journal of grid computing* 14, 2 (2016), 217–264.

[254] S. P. Singh. 1992. Reinforcement learning with a hierarchy of abstract models. In *Proceedings of the National Conference on Artificial Intelligence*.

[255] S. L. Smith, P.-J. Kindermans, et al. 2017. Don't decay the learning rate, increase the batch size. *arXiv preprint arXiv:1711.00489* (2017).

[256] E. R. Sparks, A. Talwalkar, et al. 2015. Automating model search for large scale machine learning. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*. ACM.

[257] N. Srivastava, G. Hinton, et al. 2014. Dropout: a simple way to prevent neural networks from overfitting. *JMLR* 15, 1 (2014), 1929–1958.

[258] J. Steinhardt, P. W. W. Koh, et al. 2017. Certified defenses for data poisoning attacks. In *NIPS (2017)*. 3517–3529.

[259] S. U. Stich. 2019. Local SGD Converges Fast and Communicates Little. In *ICLR 2019 ICLR 2019 International Conference on Learning Representations*.

[260] I. Stoica, D. Song, et al. 2017. A berkeley view of systems challenges for ai. *arXiv preprint arXiv:1712.05855* (2017).

[261] N. Strom. 2015. Scalable distributed DNN training using commodity GPU cloud computing. In *INTERSPEECH (2015)*.

[262] O. Suciu, R. Marginean, et al. 2018. When does machine learning FAIL? generalized transferability for evasion and poisoning attacks. In *USENIX Security 18)*. 1299–1316.

[263] S. Sultana, D. Midi, et al. 2014. Kinesis: a security incident response and prevention system for wireless sensor networks. In *SenSys*. ACM.

[264] J. Supancic III and D. Ramanan. 2017. Tracking as online decision-making: Learning a policy from streaming videos with reinforcement learning. In *ICCV (2017)*. 322–331.

[265] R. S. Sutton and A. G. Barto. 2018. *Reinforcement learning: An introduction*. MIT press.

[266] C. Tan, F. Sun, et al. 2018. A survey on deep transfer learning. In *International conference on artificial neural networks*. Springer, 270–279.
[267] M. Tan, B. Chen, et al. 2019. Mnasnet: Platform-aware neural architecture search for mobile. In *CVPR (2019)*.
[268] H. Tang, W. Wang, et al. 2019. Expression conditional gan for facial expression-to-expression translation. In *ICIP (2019)*. IEEE, 4449–4453.
[269] TensorFlow. 2019. High performance inference with TensorRT Integration. https://medium.com/tensorflow/high-performance-inference-with-tensorrt-integration-c4d78795fbfe
[270] S. Thrun and A. Schwartz. 1995. Finding structure in reinforcement learning. In *NIPS*.
[271] J. T. Townsend. 1971. Theoretical analysis of an alphabetic confusion matrix. *Perception & Psychophysics* 9, 1 (1971), 40–50.
[272] F. Tramer, F. Zhang, et al. 2016. Stealing machine learning models via prediction apis. In *25th USENIX Security Symposium (USENIX Security 16)*. 601–618.
[273] A. Ullah, J. Ahmad, et al. 2018. Action Recognition in Video Sequences using Deep Bi-Directional LSTM With CNN Features. *IEEE Access* 6 (2018), 1155–1166.
[274] M. Vacher, B. Lecouteux, et al. 2015. Speech and speaker recognition for home automation: Preliminary results. In *SpeD*. IEEE.
[275] L. G. Valiant. 1984. A theory of the learnable. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*. ACM, 436–445.
[276] V. Vanhoucke, A. Senior, et al. 2011. Improving the speed of neural networks on CPUs. (2011).
[277] J. Vanschoren. 2018. Meta-learning: A survey. *arXiv preprint arXiv:1810.03548* (2018).
[278] S. Venugopalan, H. Xu, et al. 2014. Translating videos to natural language using deep recurrent neural networks. *arXiv preprint arXiv:1412.4729* (2014).
[279] R. Volpi, H. Namkoong, et al. 2018. Generalizing to unseen domains via adversarial data augmentation. In *NIPS (2018)*. 5334–5344.
[280] G. Wang, J. Hao, et al. 2010. A new approach to intrusion detection using Artificial Neural Networks and fuzzy clustering. *Expert systems with applications* (2010).
[281] J. Wang and L. Perez. 2017. The effectiveness of data augmentation in image classification using deep learning. *Convolutional Neural Networks Vis. Recognit* (2017).
[282] M. Wang and X.-S. Hua. 2011. Active learning in multimedia annotation and retrieval: A survey. *ACM Transactions on Intelligent Systems and Technology (TIST)* (2011).
[283] M. Wang, X. S. Hua, et al. 2007. Interactive video annotation by multi concept multi modality active learning. *ICSC* (2007).
[284] M. Wang, C.-c. Huang, et al. 2019. Supporting very large models using automatic dataflow graph partitioning. In *EuroSys (2019)*. 1–17.
[285] S. Wang, T. Tuor, et al. 2018. When edge meets learning: Adaptive control for resource-constrained distributed machine learning. In *INFOCOM (2018)*. IEEE, 63–71.
[286] D. Weerasiri, M. C. Barukh, et al. 2017. A taxonomy and survey of cloud resource orchestration techniques. *ACM Computing Surveys (CSUR)* 50, 2 (2017), 26.
[287] J. Wei, W. Dai, et al. 2015. Managed communication and consistency for fast data-parallel iterative analytics. In *SoCC'15*. ACM, 381–394.
[288] W. Wen, C. Xu, et al. 2017. Terngrad: Ternary gradients to reduce communication in distributed deep learning. In *NIPS (2017)*. 1509–1519.
[289] Z. Wen, P. Bhatotia, et al. 2018. Approxiot: Approximate analytics for edge computing. In *ICDCS*. IEEE.
[290] Z. Wen, D. OâĂŹNeill, et al. 2015. Optimal demand response using device-based reinforcement learning. *IEEE Trans Smart Grid* 6, 5 (2015), 2312–2324.
[291] Z. Wen, R. Yang, et al. 2017. Fog orchestration for internet of things services. *IEEE Internet Computing* 21, 2 (2017), 16–24.
[292] S. Wu, G. Li, et al. 2018. Training and Inference with Integers in Deep Neural Networks. In *International Conference on Learning Representations*.
[293] Y. Wu, M. Schuster, et al. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint:1609.08144* (2016).
[294] H. Xiao, H. Xiao, et al. 2012. Adversarial Label Flips Attack on Support Vector Machines.. In *ECAI*. 870–875.
[295] E. P. Xing, Q. Ho, et al. 2015. Petuum: A new platform for distributed machine learning on big data. *IEEE Transactions on Big Data* 1, 2 (2015), 49–67.
[296] C. Xiong, V. Zhong, et al. 2017. Dcn+: Mixed objective and deep residual coattention for question answering. *arXiv preprint arXiv:1711.00106* (2017).
[297] X. Xu, J. Zheng, et al. 2017. Data classification using evidence reasoning rule. *Knowledge-Based Systems* 116 (2017), 144–151.
[298] Z. Xu, Y. Wang, et al. 2017. A deep reinforcement learning based framework for power-efficient resource allocation in cloud RANs. In *ICC*. IEEE.
[299] N. J. Yadwadkar, B. Hariharan, et al. 2016. Multi-task learning for straggler avoiding predictive job scheduling. *JMLR* 17, 1 (2016), 3692–3728.
[300] C. Yang, L. Feng, et al. 2018. A novel data fusion algorithm to combat false data injection attacks in networked radar systems. *TSIPN* (2018).
[301] E. Yigitoglu, M. Mohamed, et al. 2017. Foggy: a framework for continuous automated IoT application deployment in fog computing. In *AIMS*. IEEE, 38–45.
[302] Y. You, I. Gitman, et al. 2017. Large batch training of convolutional networks. *arXiv preprint arXiv:1708.03888* (2017).
[303] H. Yu, S. Yang, et al. 2018. Parallel restarted SGD for non-convex optimization with faster convergence and less communication. *arXiv preprint:1807.06629* 2, 4 (2018), 7.
[304] H. Yu, S. Yang, et al. 2019. Parallel restarted SGD with faster convergence and less communication: Demystifying why model averaging works for deep learning. In *AAAI*.
[305] S. Yun, J. Choi, et al. 2017. Action-decision networks for visual tracking with deep reinforcement learning. In *CVPR (2017)*. 2711–2720.
[306] S. Zagoruyko et al. 2016. Paying more attention to attention: Improving performance of convolutional neural networks via attention transfer. *arXiv:1612.03928* (2016).
[307] S. Zhai, Y. Cheng, et al. 2016. Doubly convolutional neural networks. In *NIPS (2016)*. 1082–1090.
[308] Z.-H. Zhan, X.-F. Liu, et al. 2015. Cloud computing resource scheduling and a survey of its evolutionary approaches. *ACM Computing Surveys (CSUR)* 47, 4 (2015), 63.
[309] C. Zhang, A. Kumar, et al. 2016. Materialization optimizations for feature selection workloads. *ACM Transactions on Database Systems (TODS)* 41, 1 (2016), 2.
[310] C. Zhang, P. Patras, et al. 2019. Deep learning in mobile and wireless networking: A survey. *IEEE Communications Surveys & Tutorials* (2019).
[311] H. Zhang, J. Li, et al. 2017. Zipml: Training linear models with end-to-end low precision, and a little bit of deep learning. In *ICML (2017)*. JMLR. org, 4035–4043.
[312] J. Zhang, C. De Sa, et al. 2016. Parallel SGD: When does averaging help? *arXiv preprint arXiv:1606.07365* (2016).
[313] Q.-s. Zhang and S.-C. Zhu. 2018. Visual interpretability for deep learning: a survey. *Frontiers of Information Technology & Electronic Engineering* 19, 1 (2018), 27–39.
[314] S. Zhang, A. E. Choromanska, et al. 2015. Deep learning with elastic averaging SGD. In *NIPS (2015)*. 685–693.
[315] W. Zhang, W. Guo, et al. 2018. LSTM-based analysis of industrial IoT equipment. *IEEE Access* 6 (2018), 23551–23560.
[316] W.-A. Zhang, B. Chen, et al. 2016. Hierarchical fusion estimation for clustered asynchronous sensor networks. *IEEE Trans. Automat. Control* 61, 10 (2016), 3064–3069.
[317] X. Zhang, X. Zhou, et al. 2018. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *CVPR (2018)*. 6848–6856.
[318] C. Zhao, C. Chen, et al. 2018. Application of Auxiliary Classifier Wasserstein Generative Adversarial Networks in Wireless Signal Classification of Illegal Unmanned Aerial Vehicles. *Applied Sciences* 8, 12 (2018), 2664.
[319] C. Zhao, M. Shi, et al. 2018. Research on the open-categorical classification of the Internet-of-things based on generative adversarial networks. *Applied Sciences* (2018).
[320] D. Zhao, Y. Chen, et al. 2016. Deep reinforcement learning with visual attention for vehicle classification. *IEEE Transactions on Cognitive and Developmental Systems* (2016).
[321] B. Zhou, J. Cao, et al. 2010. Adaptive traffic light control in wireless sensor network-based intelligent transportation system. In *VTC*. IEEE, 1–5.
[322] S. Zhou, Q. Chen, et al. 2013. Active deep learning method for semi-supervised sentiment classification. *Neurocomputing* 120 (2013), 536–546.
[323] S. Zhou, Y. Wu, et al. 2016. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160* (2016).
[324] Y. Zhou, S. Ebrahimi, et al. 2018. Resource-efficient neural architect. *arXiv preprint arXiv:1806.07912* (2018).
[325] M. Zinkevich, M. Weimer, et al. 2010. Parallelized stochastic gradient descent. In *NIPS (2010)*. 2595–2603.

A APPENDIX A

Software Specification	**Cloud**	**Edge**	**IoT devices**
Main OS	- Ubuntu - CentOS - Debian - RHEL - Windows Server - Amazon Linux	- Raspbian - NOOBS - Amazon FreeRTOS - RIOT - Google Fuchsia OS - Windows 10 IoT	- Amazon FreeRTOS - Contiki - TinyOS - RIOT - Ubuntu Core - Mbed OS
Programming Language	- Java - ASP.NET - Python - PHP - Ruby	- Java - Python - C - C++ - JavaScript	- C - C++ - Java - JavaScript - Python
Platforms	- AWS - Azure - Google Cloud Platform - IBM Cloud - Oracle Cloud	- Amazon Greengrass - EdgeX - Cisco IOx - Akraino Edge Stack - Eclipse ioFog	- AWS IoT - Azure IoT - GCP IoT - IBM Watson - Cisco IoT cloud connect

Table 1. List of OS, programming language and platform in IoT layers

B APPENDIX B

B.1 Traditional Machine Learning (TML) methods

In this subsection, we give the details of several TML algorithms, as well as their IoT applications.

Logistic Regression (LR). Logistic regression is a linear classifier capable of performing binary or multi-class classification. It is among the simplest classification algorithms. In a binary classification setting, prediction target y is usually formulated as $y \in \{0, 1\}$ and the prediction probability of the positive class given a d-dimensional input $\mathbf{x} = [x_1, x_2, ..., x_d] \in \mathbb{R}^d$ can be calculated via:

$$p(y = 1|\mathbf{x}; \boldsymbol{\theta}) = \sigma(\mathbf{w}^T\mathbf{x} + b) = \sigma(\sum_{i=1}^{d} w_i x_i + b), \quad (1)$$

In the equation $\boldsymbol{\theta} = \{\mathbf{w} = [w_1, w_2, ..., w_d] \in \mathbb{R}^d, b \in \mathbb{R}\}$ denotes the model parameters, and $\sigma(\cdot)$ is an activation function used to squash the linear output within the range $[0, 1]$ for probabilistic interpretation. The training of the classifier aims to learn suitable values for the parameters $\boldsymbol{\theta} = \{\mathbf{w}, b\}$, starting from some random initialization, through minimizing of a loss or cost function $J(\boldsymbol{\theta})$. For LR, log (Cross-Entropy) loss is used, i.e., $J(\boldsymbol{\theta}) = -\ln p(y|\mathbf{x}; \boldsymbol{\theta})$ for data point $\{\mathbf{x}, y\}$ to facilitate the calculation of the model gradient and to minimize the model loss. With the trained model, for any query data the classification decision can be made via thresholding the predicted probability.

Extend the binary LR to support c-class prediction scenarios, the target $y \in \{1, 2, ..., c\}$. Softmax function is applied to the output layer to normalize the c outputs into probabilities. Different from binary LR, there are c sets of parameters $\boldsymbol{\theta} = \{\mathbf{W} = [\mathbf{w}_1, ..., \mathbf{w}_c]^T, \mathbf{b} = [b_1, ..., b_c]^T\}$ to be estimated by minimizing the cross-entropy loss (i.e., log loss). With the trained model parameters, at the inference stage the class label will be assigned to the one with the largest classification probability.

Decision Trees (DT). Decision Tree is a tree-like model for classification or regression tasks. A decision tree is made up of nodes and edges where a node can be seen as a feature, and the edge represents a condition for classification. The learning process of DT is to select the optimal feature that can best split the training examples in a recursive manner. Based on certain criteria such as information gain [46] or Gini impurity [13], the root node can be selected from the features, which will divide the whole training population into two or more homogeneous sets. For each sub-population, sub-nodes will be selected in a similar manner and this process will repeat until all the subsets are pure (i.e., with the same class label for each subset). These nodes and edges constitute the trained model and can be used for inference.

Different from most of the other classifiers (such as LR, SVM, ANN, DL) which require the input to be normalised to numerical values, the unique tree-structure of DT make it possible to take both numerical and nominal values, making it a highly interpretable tool for various classification tasks (e.g., medical records diagnosis). However, DT suffers from the "curse of dimensionality", and it faces an overfitting problem when the input dimensionality is too high. With unstructured high-dimensional data, feature engineering/extraction is one of the necessary steps to take (for dimensionality reduction), before DT is applied.

DT can be used as a main classifier (e.g., for low-dimensional structured data) or collaborative classifier with other machine learning algorithms on various IoT applications, such as Intrusion detection system [45] in fog environment.

Random Forest (RF). As its name implies, a RF is an ensemble model with many DTs as base classifiers. The individual DTs are constructed by random sampling the features and the training examples for diversity, boosting the performance of multiple classifier systems. The random sampling process makes the individual DTs less correlated —with different prediction errors— and the aggregating function can smooth the large prediction variance, making RF a robust classifier with high generalization capabilities. There are several key hyper-parameters for RF, and two main ones are: feature number for individual DTs, and number of DT classifiers. For individual DTs, there is a trade-off between generalization and discrimination capabilities with respect to feature dimension, and one popular heuristic value is to use the square root of original dimension number (e.g., the default setting in scikit-learn). For the RF though, the performance of the model usually correlates wit the number of DTs. Yet the performance gain tends to become less significant since the diversity (among the DT classifiers, which are correlated to some extent) will decrease accordingly. Since both the efficiency and storage are proportional (in a linear manner) to the classifier number, to find a number configuration that can result in optimal effectiveness, storage and efficiency performance is of vital importance.

RF is one of the most popular classifiers due to its great generalization capability. For example, it has been used for intrusion detection [6, 36] and anomaly detection. In a previous study [36], the authors collected and manually labeled data from 17 distinct IoT devices and used RF algorithms to recognize IoT device categories from the white list.

Support Vector Machine (SVM). Support Vector Machine (SVM), also referred to as large margin classifier, aims to find a decision boundary (separating hyperplane) that can best separate (i.e., with the largest margin) positive/negative classes. Fig. 1 shows some basic concepts in SVM including support vectors, margin and slack variable ξ. ξ is a non-negative variable that is used to measure the misclassified instances and those within the margin (i.e., margin violation, as shown in Fig. 1). The objective function of (soft-margin) SVM can be constructed to maximize the margin while penalizing these instances:

$$\underset{\mathbf{w},b,\xi_i}{\operatorname{argmin}} \frac{||\mathbf{w}||^2}{2} + C\sum_{i}^{N}\xi_i, \quad \text{subject to} \quad y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1 - \xi_i, \text{where} \quad \xi_i \geq 0, \quad i = 1, 2, ..., N. \tag{2}$$

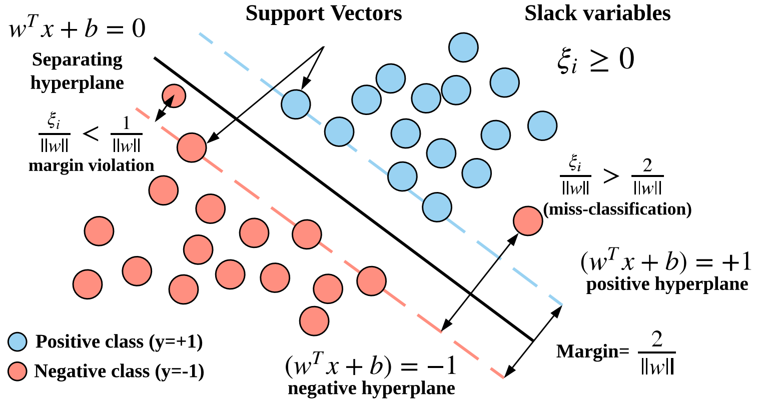

Fig. 1. Basic concepts in SVM

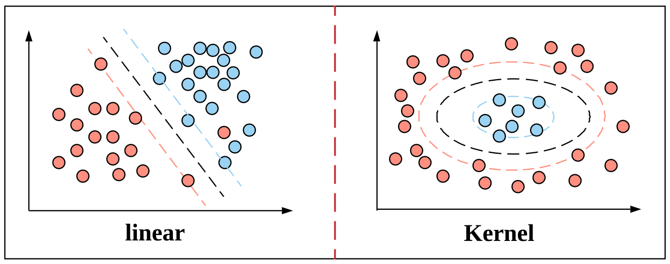

Fig. 2. Margins for linear (left) and kernel (right) SVM

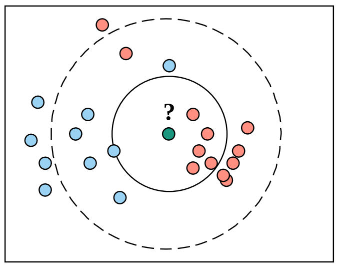

Fig. 3. Majority voting process in KNN

In Eq (2), $\{\mathbf{x}_i, y_i\}_{i=1}^N$ are the N training sets with $y_i \in \{-1, 1\}$, and C is a regularization hyper-parameter that trades-off between the margin and errors (i.e., measured by $\sum_i^N \xi_i$). It is obvious to see that if we set $C = 0$, we can get a classifier with the large margin at a cost of potential high training errors. On the other hand, if we set C to a very large number (or ∞ in theory), errors are less likely to be tolerant, and we may end up with a classifier with narrow or hard margin. It is worth noting that Eq.(2) can be further simplified into $\text{argmin}_{\mathbf{w},b} \frac{||\mathbf{w}||^2}{2} + C \sum_i^N \max(0, 1 - y_i(\mathbf{w}^T \mathbf{x}_i + b))$, where the term $\max(0, 1 - y_i(\mathbf{w}^T \mathbf{x}_i + b))$ is also referred to as hinge loss.

On the other hand, for highly non-linearly separable data, instead of employing feature engineering/extraction processes, an elegant alternative—kernel SVM can be applied, and Radial Basis Function (RBF) is one of the most popular kernel functions. Kernel SVM also has the aforementioned characteristics such as soft margin, and it tends to have great performance on small non-linearly separable data. Fig. 2 shows the margins on linearly separable (by linear SVM) and non-linearly separable (by kernel SVM) data, respectively.

SVM is notable for its generalization capability and is suitable for those small datasets with high-dimensional features [67], and there are many IoT applications such as Android malware detection system [51], smart weather prediction [47], etc.

K-Nearest Neighbour (KNN). KNN is a nonparametric, instance-based, non-linear classifier. Under the classification setting, given any query data, KNN essentially performs majority voting among the K most similar training samples, as shown in Fig. 3. The similarity can be measured by some distance metrics such as Minkowski distance $D(\mathbf{x}, \mathbf{y}) = (\sum_{j=1}^{d} |x_j - y_j|^p)^{\frac{1}{p}}$ (for d-dimensional vectors $\mathbf{x}, \mathbf{y}$). It is worth noting that when $p = 1$ and $p = 2$, the Minkowski distance can be seen as Manhattan Distance and Euclidean distance respectively, yet for different applications the optimal distance metric may vary from case to case.

One of the key properties is that KNN does not require any training process (i.e., lazy learning), and for any query data, the distance calculation has to be performed for each sample in the whole training set, which makes KNN a less-scalable approach for large datasets. Another issue is the selection of hyper-parameter K. A small K may make KNN sensitive to outliers in the training set while a large one may make KNN less discriminate. Nevertheless, KNN is a powerful non-linear classifier with low-dimensional small datasets, and there are many IoT applications such as network intrusion detection [31], anomaly detection [61] and Urban noise identification [1], etc.

Naive Bayes (NB). NB is a probabilistic classifier which takes the class prior distribution into account, and assumes the features are conditionally independent. Based on the Bayesian theory and the posterior probability, it can be presented as $p(y|\mathbf{x}) = \frac{p(\mathbf{x}|y)p(y)}{p(\mathbf{x})}$, where $p(\mathbf{x}|y)$ is the likelihood; $p(y)$ is the prior; $p(\mathbf{x})$ is the evidence. The classification process is to assign the label with the largest posterior probability, and in this case the term evidence $p(\mathbf{x})$ remains a constant which can be cancelled out, i.e., $p(y|\mathbf{x}) \propto p(\mathbf{x}|y)p(y)$, which can be further written into $p(y|\mathbf{x}) \propto \prod_{j=1}^{d} p(x_j|y)p(y)$ for d-dimensional input $\mathbf{x}$ due to the feature independence assumption. In practice, logarithm is often used to avoid the problem of floating point underflow, and the class label $\hat{y}$ can be assigned via:

$$\hat{y} = \underset{y \in \{1,2,\ldots,c\}}{\text{argmax}} \ \ln p(y) + \sum_{j=1}^{d} \ln p(x_j|y), \tag{3}$$

NB is good at modelling both continuous and discrete data. For example the likelihood of a discrete feature can be calculated by frequency while the likelihood of a continuous feature can be calculated by density estimation (e.g., Gaussian). It takes the prior of class distribution into consideration, which is helpful in data imbalanced problems. However, it also suffers from the "curse of dimensionality" like DT, and normally can not be used directly on unstructured data before feature extraction/engineering approaches have been applied. When the feature independence assumption is not significantly violated, it is normally served as an efficient and effective classifier. There are many IoT applications, such as network traffic analysis for DoS attach detection [23], animal health monitoring [56], etc.

K-Means. Different from the supervised classification models above, K-means is an unsupervised clustering algorithm without using class label information for training. Given a number of data points, K-Means aims to find K centroids (i.e., means), and the corresponding nearest samples to form the clusters. Various distance metrics can be used for K-means algorithm, and the most common one is based on Euclidean distance, whose objective function is:

$$\text{argmin} \sum_{i=1}^{N} \sum_{k=1}^{K} \gamma_{ik} ||\mathbf{x}_i - \mu_k||^2, \tag{4}$$

where μ_k is the centroid of the kth cluster and $\gamma_{ik} \in \{0, 1\}$ denotes whether sample $\mathbf{x}_i$ belongs to the kth cluster (1) or not (0). K-means clustering is an heuristic process—starting from random values (of the centroids $\{\mu_k\}_{k=1}^{K}$), it will 1) assign each example to the nearest cluster and 2) update the K centroids (by re-calculating the means of the

Table 2. Summary of Traditional Machine Learning Models (Note: In time complexity, m represents the number of training sample, n represents the feature dimension, k represents the selected K value, c represents the class number and t indicates the tree number)

Method	Learning model	Category	Typical input data	Time Complexity	Characteristics	IoT Application
DT		Discriminative	Various	$O(m \cdot n^2)$ [60]	• Dividing training samples to branches and leaves • High interpretability method • Require large memory space due to the construction nature.	• Intrusion detection [44] • Suspicious detection on traffic sources [14] • Future Heart Attack Quantity prediction [26]
SVM		Discriminative	Various	$O(m^2 \cdot n)$[8]	• Good Generalization capability and suitability for small dataset with large feature • Difficult on selection optimal kernel • High computation complexity on large dataset with complex kernel	• Malware detection [51] • Attack detection in smart grids [62] • Smart weather prediction [47]
NB	Supvervised	Discriminative	Various	$O(m \cdot n + n \cdot c)$ [8]	• Ease to implement • Generalizes well to multi-class problem • Low dependence on large dataset and robustness • Hard to capture relation information	• Detection of network intrusion [39] • Animal health monitoring [56]
KNN		Discriminative	Various	$O(m \cdot n \cdot k)$ [77]	• Nonparametric, instance-based • No training process, no model construction • Sensitive to the outlier • Difficult on selection optimal K	• Anomalies detection [19] • Urban noise identification [1]
RF		Discriminative	Various	$O(t \cdot n^2 \cdot logn)$ [34]	• Robust to over-fitting • Bypasses feature selection • Impractical in specific real-time application	• DDoS attack detection [29] • Unauthorized IoT devices detection [36]
K-Means	Unsupervised	Clustering	Various	$O(m \cdot n \cdot k)$[8]	• Ease to use on Unlabelled data • Produces tighter cluster than hierarchical clustering • Less effective than supervised learning method • Difficult on selection optimal K	• Sensor fault detection [78] • Sybil detection in industrial WSNs [72]
PCA		Dimension Reduction	Various	$O(m \cdot n^2 + n^3)$ [8]	• Used for dimensional reduction • Consequently reduce the complexity of the model • Should be used with other ML methods	• Real-time detection systems in IoT environments [15] • Traffic anomaly detection [12]

corresponding samples for each cluster). It is an iterative process and the updating will stop until Eq.(4) is minimized (e.g., lower than a pre-defined threshold).

K-means is among the most popular clustering algorithms due to its simplicity. However, using a Euclidean distance based method is limited to spherical datasets, and finding the most suitable distance metrics (for different applications/-datasets) is one of the key issues for improving the performance of K-means algorithm. K-means is widely used for IoT applications such as sensor fault detection [78], Sybil detection [72], etc.

Principal component analysis (PCA). PCA is another unsupervised learning approach and it is normally used for dimensionality reduction or feature decorrelation. Covariance matrix $\mathbf{S}$ which reflects the correlation of the features can be calculated via $\mathbf{S} = \frac{1}{N}\sum_{i=1}^{N}(\mathbf{x}_i - \mu)(\mathbf{x}_i - \mu)^T$ for N d-dimensional training data points $\{\mathbf{x}_i\}_{i=1}^{N}$ with $\mu = \frac{1}{N}\sum_{i=1}^{N}\mathbf{x}_i$. Eigenvalue decomposition can be performed on $\mathbf{S}$ such that the leading $s(< d)$ eigenvectors can be used as a transformation matrix for feature decorrelation or dimensionality reduction. PCA is one of the most popular feature extraction tools due to its simplicity, and there are many IoT applications such as traffic anomaly detection [4].

Table 2 summarizes the aforementioned TML, including their advantages, disadvantages and applications in IoT systems.

Artificial Neural Networks (ANN). ANN is a general extension of the aforementioned linear classifiers. Compared with *Perceptron* or *LR* which linearly project input data to the output, *ANN* has an additional "hidden layer" (with a non-linear activation function), which enables *ANN* to model non-linearity. However, in contrast to linear classifiers, this additional hidden layer makes it more difficult to see the relationship between the input and output data (i.e., low interpretability). Although in theory, with one hidden layer *ANN* can model any complex non-linear functions, in practice it has limited generalization capabilities when facing unseen data. *ANN* with more layers, also referred to as deep neural networks, tend to have better modelling capability, which will be introduced in the next subsection.

B.2 Deep Learning (DL) methods

In this section, we detail several deep learning algorithms: Deep Neural Networks (DNN), Convolutional Neural Network (CNN), Recurrent Neural Network (RNN), etc.

Deep Neural Networks (DNN). As previously mentioned, LR is one of the most popular linear classifiers, which means the decision boundary is a line or hyperplane in the feature space. However, most of the data are more complex and are not linearly separable in real-world scenarios. In this case, directly applying linear classifiers such as LR on the raw data would yield unsatisfied classification performance. One could empirically use hand-crafted features followed by LR, yet this trial-and-error process can be time-consuming, and it may be challenging to define the high-order discriminant features where the data can be linearly separable. One alternative is to use data-driven methods, and it is straightforward to extend LR to DNN (i.e., Multilayer Perceptron MLP).

DNN is built upon LR with a stack of hidden layers. By applying activation functions like sigmoid, tanh, or ReLU on the hidden layers, raw features can be transformed in a non-linear manner, yielding discriminant (i.e., linearly separable) features before the output layer. Similar to LR, the training process is to minimise the log loss. However, the parameters of DNN are the layer-wise connections, which can be learned using the back-propagation approach. DNN performs end-to-end learning, which means it learns the feature extractors and classifiers simultaneously.

DNN can be deemed as the simplest form of deep learning model which comprises one input layer, one output layer as well as multiple hidden layers for more complex feature extraction. It is worth noting that Artificial Neural Network (ANN) is a special case of DNN with only one hidden layer. The layers are organised in a hierarchical manner, with each layer being a function of the layer that preceded it. For example, the second layer $\mathbf{h}^{(2)}$ (i.e., also the first hidden layer) can be expressed as a function of the first (i.e., input) layer (e.g., with input vector $\mathbf{x} \in \mathbb{R}^d$):

$$\mathbf{h}^{(2)} = g^{(2)}(\mathbf{W}^{(2)T}\mathbf{x} + \mathbf{b}^{(2)}), \tag{5}$$

where $\{\mathbf{W}^{(2)}, \mathbf{b}^{(2)}\}$ are the parameters and $g^{(2)}$ is the activation function (e.g., ReLU) for the second layer. We also call these layers fully connected or dense layers. Similarly, the $l^{th}(l > 2)$ hidden layer can be written as

$$\mathbf{h}^{(l)} = g^{(l)}(\mathbf{W}^{(l)T}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}). \tag{6}$$

Assuming there are a total number of L layers, then an output layer linearly transforms the previous hidden units $\mathbf{h}^{(L-1)}$, followed by a softmax function for probability scaling:

$$p(\mathbf{y}|\mathbf{h}^{(L-1)}) = \text{softmax}(\mathbf{W}^{(L)T}\mathbf{h}^{(L-1)} + \mathbf{b}^{(L)}), \tag{7}$$

Note the model parameters include the layer-wise weight matrices and bias vectors, i.e., $\boldsymbol{\theta} = \{\mathbf{W}^{(l)}, \mathbf{b}^{(l)}\}_{l=2}^{L}$, which needed to be estimated by minimising the loss function (e.g., log loss). DNN has been widely used for various IoT applications, including wearable-based activity recognition [65], traffic congestion prediction[11], healthcare[42], etc.

Convolutional Neural Networks (CNNs). Similar to the DNN, CNN also comprises multiple hidden layers. It learns to map from 2D data (e.g., images) at the input, output the class probabilities at the output. DNN usually contains three types of hidden layers: convolution layer and pooling layer at each stage and FC/dense layer right before the output layer. In a convolution layer, local patterns can be extracted by learning several convolutional kernels/filters (with a

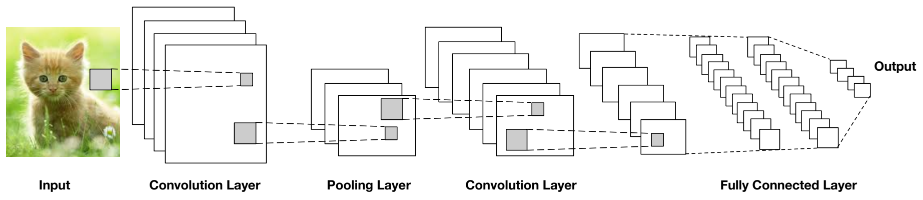

Fig. 4. An architecture of CNN

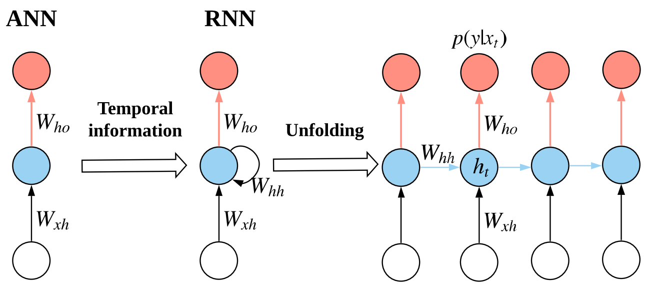

Fig. 5. From ANN to RNN

predefined size, such as 3×3), each of which has the same shared weights to be estimated (via back-propagation). This weight-sharing scheme makes CNN effective in dealing with high-dimensional data (such as high-resolution images). After convolution operation with a number of kernels/filters, the corresponding feature maps can be formed, and then a pooling layer can be employed for down-sampling these feature maps for more compact representation. In CNN, normally there are multiple convolution/pooling layers, before FC/dense layers can be applied; Fig. 4 shows an example architecture of CNN. Since cameras are one of the major parts of the IoT environment, CNN can be a very useful tool and has been widely used such as in traffic sign detection on autonomous driving [58].

Recurrent Neural Networks (RNNs). RNN is a type of deep model and it is designed to model and recognise sequential data (e.g., time-series data). Compared with DNN, a hidden unit of RNN can feed-forward on itself in the next timestamp and thus can memorise the temporal information for sequential inference. Fig. 5 shows how to extend an ANN (DNN with one hidden layer) to a three-layer vanilla RNN by modelling the temporal information, we use an arrow to represent FC/dense layer for better visual effects. Compared with ANN, RNN has an additional set of parameters, i.e., the hidden-to-hidden transformation matrix $\mathbf{W}_{hh}$ to be estimated, and the full parameters can be expressed as: $\Theta = \{\mathbf{W}_{xh}, \mathbf{W}_{hh}, \mathbf{W}_{ho}, \mathbf{b}_o, \mathbf{b}_h\}$. For $\mathbf{x}_t$, the input vector at the t^{th} timestamp, the feed-forward pass can be written as:

$$p(\mathbf{y}|\mathbf{x}_t) = \text{softmax}(\mathbf{W}_{ho}^T \mathbf{h}_t + \mathbf{b}_o)$$
$$\text{where} \qquad \mathbf{h}_t = \tanh(\begin{bmatrix} \mathbf{W}_{xh} \\ \mathbf{W}_{hh} \end{bmatrix}^T [\mathbf{x}_t, \mathbf{h}_{t-1}] + \mathbf{b}_h), \tag{8}$$

It is clear that the current hidden state $\mathbf{h}_t$ is calculated based on the previous hidden state $\mathbf{h}_{t-1}$ and the current signal $\mathbf{x}_t$, while the output layer remains the same as DNN (see Eq.(7)). We can see RNN is an end-to-end method where both the sample-wise features $\mathbf{h}_t$ and predictions $p(\mathbf{y}|\mathbf{x}_t)$ can be learned simultaneously.

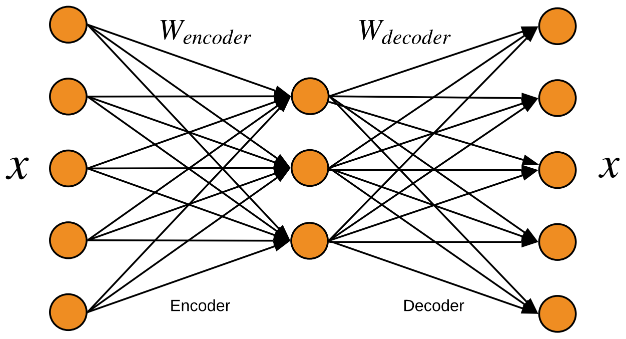

Fig. 6. The encoder-decoder structure of a three-layer Auto-encoder

For complex time-series data analysis, we need a deeper RNN (i.e., larger window, multiple layers, etc.) to capture the high-level temporal/contextual information. Yet vanilla RNN has a gradient explosion/vanishing problem owing to its numerical properties. In the 1990s, a complex hidden unit named long short term memory (LSTM) [24] was proposed for large-scale recurrent neural network construction, which can preserve the error/gradient that can be back-propagated effectively through time and layers. LSTM includes four different gates organised in a special internal structure, in contrast to tanh function for activation in vanilla RNN. Accordingly, it also has four sets of specific gate parameters to be estimated.

RNNs achieve great performance in time-series applications, for example machine translation and speech recognition. In IoT applications, it has shown its great performance in sensor-based power station condition prediction [74], wearable-based activity recognition [20], etc.

Auto-encoder (AE). An Auto-encoder(AE) is a type of unsupervised neural network, and it can transform the data into latent code/representation/feature (e.g., in lower-dimensionality), from which the original data can be reconstructed. AE is widely used for data compression and feature extraction, and a typical AE includes two parts: encoder, and decoder. Fig. 6 shows a three-layer encoder-decoder structure for AE, and the d-dimensional input vector $\mathbf{x}$ can be transformed into the latent representation $\mathbf{h}$ via a linear transformation followed by an activation function $g(.)$, i.e., $\mathbf{h} = g(\mathbf{W}_{\text{encoder}}^T\mathbf{x} + \mathbf{b}_0)$. Similarly, decoder can be used to reconstruct the data such that $\mathbf{x}' = g(\mathbf{W}_{\text{decoder}}^T\mathbf{h} + \mathbf{b}_1)$. Note $\mathbf{\Theta} = \{\mathbf{W}_{\text{encoder}}, \mathbf{b}_0, \mathbf{W}_{\text{decoder}}, \mathbf{b}_1\}$ are the model parameters that can be estimated via minimising a reconstruction loss

$$\underset{\mathbf{\Theta}}{\operatorname{argmin}} \, L(\mathbf{x} - \mathbf{x}'). \tag{9}$$

Note in Eq. (9), $L(.)$ can be very flexible, and some popular ones include mean squared error, log loss (for binary input), etc. It is worth noting that with small dataset, a popular constraint is to set $\mathbf{W}_{\text{encoder}} = \mathbf{W}_{\text{decoder}}^T$, which can limit the degree of freedom of this model with better generalisation capabilities. AE can be used as a powerful tool for dimensionality reduction. Compared with PCA, the non-linear transformation may extract more discriminant information for classification. It can also be used for anomaly detection, and the intuition is that the trained AE (based on normal class) cannot reconstruct the abnormal class well, yielding large reconstruction errors. There are many IoT applications for AE, such as fault diagnosis in hardware devices [7], and anomaly detection in the performance of assembly lines [43].

Generative Adversarial Network (GAN). Generative Adversarial Networks (GAN) is a generative model with adversarial architecture. Fig. 7 shows the basic network architecture of a GAN, which contains a generator and a discriminator.

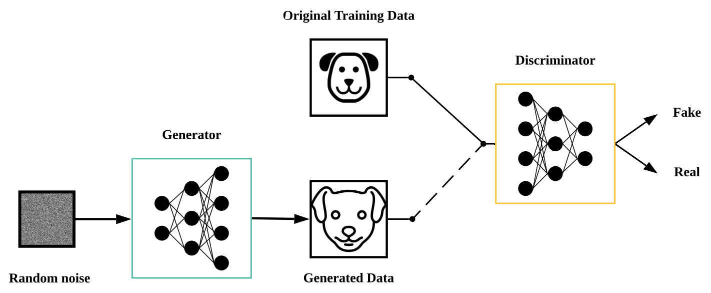

Fig. 7. Generative Adversarial Network Architecture

The generator aims at generating indistinguishable samples compared to the real data while the discriminator works adversarially to distinguish the generated fake samples from the real data. It is an iterative competition process that will eventually lead to a state where the generated samples are indistinguishable from the real data. The generative model aims to learn the data distribution and generate data samples for those domains that lack data samples.

The decent idea of applying adversarial structure on deep neural networks brought impressive results related to content generated, and realistic visual content. Moreover, it can be applied to many applications such as image restoration, style transfer, sample generation. However, there are three main drawbacks on GAN frameworks. The first one is that the network is hard to train and the training process is destabilized (i.e., "Non-convergence"). The second drawback is that the generator will tend to produce limited varieties of samples at the end of a training period (i.e., "Mode collapse"). The last drawback of GAN is the gradient diminishing problem that occurs when the balance between generator and discriminator were broken.

GANs have been recently implemented in the IoT environment, mainly on IoT security. For example, a previous study [76] proposed a GAN based framework for improving open-categorical classification on individual identity authentication application. Moreover, GAN has been used as a tool to generate large datasets that do not need manual annotation. For example, a study[75] explored the feasibility of using GAN to generate illegal Unmanned Aerial Vehicles (UAVs) dataset and obtain a better classification model with better accuracy. As the data generated from sensors may be unlabelled, GANs may have more potential applications in the IoT environment.

B.3 Reinforcement Learning (RL) methods

The goal of a reinforcement learning agent is to find an optimal policy to maximize the expected sum of future rewards $J(\theta)$ parameterized by θ. At each time step t, reward $r_t = r(a_t, s_t)$ is given when an agent takes an action $a_t \in A$ at state $s_t \in S$.

$$\underset{\theta}{\arg\max} \quad J(\theta) = \mathbb{E}_{t \sim p_\theta(t)}[\sum_t r(s_t, a_t)] \tag{10}$$

In the equation, $p_\theta(t)$ represents the interactions between the RL agent and outer environments. The trajectory depends on two factors: the agent policy and the environment dynamic. Agent takes actions based on its policy $\pi_{\theta(a_t|s_t)}$. The

Table 3. Summary of Deep Machine Learning Models

Method	Learning Type	Category	Input data Type	Characteristics	IoT Application
CNNs	Supvervised	Discriminative	2-D (image, sound, etc.)	• Mainly used on image processing • Less connection compared to DNNs. • Require large training samples.	• Traffic sign detection • Plant disease detection • Bridge crack detection
RNNs		Discriminative	Sequential data	• Mainly used to analyze sequential data • Useful in IoT applications with time-dependent data	• Identify movement pattern • Behavior detection • Human activity recognition • Mobility prediction
AEs	Unsupervised	Generative	Various	• Mainly used for feature extraction, and dimensionality reduction • Optimized by reconstructs input data • Can be used on unlabeled data	• Emotion recognition • Machinery fault diagnosis • Intrusion detection • Failure detection
GANs		Generative	Various	• Learn data discribution • Can be used as a data generation tool • Two part networks: a generator and a discriminator	• Localization and way finding • Image to text

dynamic transition for the environment $s_t \times a_t \longrightarrow s_{t+1}$ can be expressed as $p(s_{t+1}|s_t, a_t)$ and is usually unknown. Overall, the whole trajectory $p_\theta(t)$ can be represented as:

$$P_\theta(s_1, a_1 ... s_T, a_T) = p(s_1) \prod_{t=1}^{T} \pi_{\theta(a_t|s_t)} p(s_{t+1}|s_t, a_t) \tag{11}$$

It is clear from the above equation that, if we are smart on the agent policy $\pi_{\theta(a_t|s_t)}$ or the transition dynamic $p(s_{t+1}|s_t, a_t)$, we can find the best trajectory $p_\theta(t)$ which maximizes reward at each time step. The RL community has formalized these two approaches as *Model-Free* and *Model-Based* approaches based on the fact that the later one directly models the system dynamic transitions.

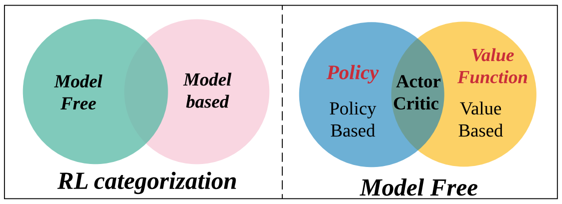

Fig. 8. Reinforcement Learning Categorization

Fig. 8 gives a broad view of the reinforcement learning world. The reinforcement learning method can be divided into *Model-free* methods and *Model-based* methods with correlations combining the features of both methods. The model-free approach can be subdivided into *Policy based*, *Value based* and *Actor-Critic* approaches according to the ways that the best policy is generated. In this section, we will detail both approaches with simple mathematical expressions and list the most famous algorithms. We will also cover some works that take advantage of both methods.

Model-Free Methods. They learn a policy and decide the best action to take given a certain state. It can be categorized as *policy-gradient*, *value-based* and *actor-critic* methods, based on how the policy is generated.

Value-Based methods learn a value function to estimate the "goodness" $V(s_t)$ for reaching a certain state s_t, or the "goodness" $Q(s_t, a_t)$ for taking certain action a_t given the state s_t. Hereby the "goodness" function estimates the sum of future rewards from current state s_t till the end s_T (given a finite trajectory). At each step, the agent chooses the action with the highest score based on the estimated value function. *Value-based* approach is deterministic, and may not be sufficient to solve complex problems. A list of the most prevalent algorithms includes Q-Learning [70], DQN [38], Prioritised Experience Replay [52], Dueling DQN [69], Double DQN [66] and Retrace [40].

Policy-Gradient methods provide an attractive paradigm by directly maximising $J(\theta)$ (Equa 10) with respect to the parameters θ of the policy $\pi_\theta(a|s)$. The gradient with respect to the parameters θ can be derived as

$$\nabla_\theta J = \mathbb{E}_\theta[\sum_t \nabla_\theta log \pi_\theta(a_t|s_t)(R_t - b_t)] \tag{12}$$

$R_t = \sum_{t'=t} \gamma^{t'-t} r(s_{t'}, a_{t'})$ with a discounting factor γ emphasizing the agent more on recent rewards. To reduce the policy variance given different datasets, a baseline b_t that does not depend on future states or actions is subtracted. In practice, the expected future return is sampled and aggregated within a trajectory. Several works have been proposed based on this paradigm to either reduce the policy variance [53], increase scalability [22] or reduce sample complexity [54].

Actor-Critic algorithms [28] are similar to the policy-gradient approach in updating the policy, while an estimated value function $V(s_t)$ is leveraged in place of the constant baseline b in the original equation 12. The term $(R_t - b_t)$ is thus an estimate of the *Advantage* defined as $A(a_t, s_t) = Q(a_t, s_t) - V(s_t)$ where a_t is the action and s_t denotes the current state, with R_t estimating $Q(a_t, s_t)$. Actor-critic methods experience much lower variance with the policy π and value function V seen as actors and critics respectively. Works utilising this architecture benefit from both the policy-gradient and value-based methods, with DDPG [32] combining deep Q-learning for continuous action space, A3C [37] for concurrent training. Other prevalent works include TD3 [17], Soft-Critic SAC [21].

Model-based Methods. The model-based algorithms differ from the model-free methods in that the latter cares less about the environment's inner working and the rewards are estimated through sampling. On the contrary, a model-based approach focuses on the model to predict the next state at each time step. Model-based RL achieves good sample efficiency by learning the transition dynamics of the environments directly. During learning, sample trajectories are collected and trained with supervised learning. Several different approaches have been applied to study the dynamics, Gaussian Process approaches [10][27][2], Time-varying linear models approaches [30][33] or Deep networks [41][18][9].

C APPENDIX C

Metrics	Formula	Type	Evaluation Focus
Accuracy(acc)	$\frac{tp+tn}{tp+fp+tn+fn}$	Binary	This metric measures the correct percentage of the total samples
	$\frac{\sum_{i=1}^{l}\frac{tp_i+tn_i}{tp_i+tn_i+fp_i+tn_i}}{l}$	Multi class	Average accuracy for all classes
Error Rate(err)	$\frac{fp+fn}{tp+fp+tn+fn}$	Binary	This metric measures the miss-classification percentage over evaluated samples
	$\frac{\sum_{i=1}^{l}\frac{fp_i+fn_i}{tp_i+tn_i+fp_i+tn_i}}{l}$	Multi class	Average Error Rate of all classes
Precision(p)	$\frac{tp}{tp+fp}$	Binary	Precision measures correct classified positive samples in a positive class
	$\frac{\sum_{i=1}^{l}\frac{tp_i}{tp_i+fp_i}}{l}$	Multi class	Average of precision on each class
Recall(r)	$\frac{tp}{tp+fn}$	Binary	Recall measures the fraction of correct classified positive samples
	$\frac{\sum_{i=1}^{l}\frac{tp_i}{tp_i+tn_i}}{l}$	Multi class	The average of recall for each class
F1-Score(FS)	$\frac{2*p*r}{p+r}$	Binary	This metric measures the harmonic mean of recall and precision
	$\frac{2*p_M*r_M}{p_M+r_M}$	Multi class	The average F1-Score
Geometric Mean(GM)	$\sqrt{tp*tn}$	Binary	This metric is similar to the F1-Score but aims to maximize the tp rate and tn rate

Table 4. Evaluation Metrics for Classification Problem.
Note: i indicates the class C_i, and M donates the macro-averaging.

Metric	Formula	Evaluation Focus
Mean Squared Error	$\frac{1}{N}\sum_{i=1}^{N}(y_i-\hat{y})^2$	MSE calculates the square residual for every data point, contribute more to the outlier
Mean Absolute Error	$\frac{1}{N}\sum_{i=1}^{N}\lvert y_i-\hat{y}\rvert$	MAE calculates the absolute residual for every data point, so that negative and positive residuals do not cancel out
Mean Absolute Percentage Error	$\frac{1}{N}\sum_{i=1}^{N}\lvert\frac{y_i-\hat{y}}{y}\rvert$	Percentage equivalent of MAE
Mean Percentage Error	$\frac{1}{N}\sum_{i=1}^{N}(\frac{y_i-\hat{y}}{y})$	MPE can help to check whether the model underestimates or overestimates

Table 5. Evaluation Metrics for Classification Problem. Note: y_i indicates the predicted value for ith sample, and $\hat{y}$ denotes the ground truth value. N indicates the total number of samples.

D APPENDIX D

In this section, we review the defending methods for the *three* types of the attacks discussed in the manuscript.

Defending model exploratory attack. The most straightforward defense of this type of attack is to constrain the API, not returning confidences and not responding to probing queries. The minimization can be achieved in three approaches: *Rounding confidence*, *Differential privacy* and *Ensemble methods*

Rounding confidence is a type of defense that rounds confidence sources of an application to some fixed precisions [16]. Notably, some online ML service providers are already working on it. For instance, BigML and Amazon provide five decimal places and 16 significant digits for their confidence scores respectively when answering queries. Limiting the precision can decrease the success rate of attacks, this is because the outputs of the model are approximated. In equation-solving attacks, for example, if the output of an equation-system is rounded, it will increase the difficulty for attackers to guess the target function.

Differential privacy is a class of mechanisms to protect, especially the privacy of training data [68]. A set of *Differential privacy* methods has been applied to protect regressions [5, 73], SVMs [48], decision trees [25] and neural networks [57]. The main idea of *Differential privacy* is to avoid a query that allows an adversary to distinguish closely neighboring model parameters.

Ensemble methods returns an aggregated output predicted by a set of models. This prevention method was mentioned in [64] and may have more resilience against the model exploratory attacks, compared with other methods. Bonawitz [3] designed a new protocol to compute the sum of a large subset of models supporting a secure federated learning setting.

Defending data poisoning attack. Compared with other systems, defending data poisoning attacks is more critical in machine learning systems, because training data coming from the outside world is very easy to poison. Steinhardt et al. [59] developed an outlier detector for linear classifiers to generate approximate upper bounds of the training dataset. The data (poisoned) that exceeds the bounds will be removed from the training dataset, not changing the distribution of the clean data. The paper [49] aimed to develop a strategy to defend against attacks that attempts to poison the PCA based anomaly detection models. Since the PCA methods are less sensitive to outliers, an ideal approach against the variance injection caused by the perturbed data is to filter the poisoned data with predefined threshold. The paper also proposed two approaches [49] for selecting the threshold: the first uses covariance matrix, and the second is to find the maximal scale estimate of the data projection.

Defending evasion attack. Crafting the adversarial samples is a complex optimization process as it is very hard to build a general tool for defending. Thus, *adversarial training* which adds the adversarial examples to the training set is a straightforward way to increase the model robustness [35, 55, 63]. Moreover, Defense-GAN [50] leverages a generative model to generate more samples similar to the training data, reducing the adversarial perturbation significantly. Apart from these general defending methods, some defenses use model hardening techniques. [71] proposed a feature squeezing method to reduce the complexity of representing the data. Less important adversarial perturbations are filtered out afterwards. In the following section, we survey the attacks and the defenses in real IoT applications and then discuss the challenges of building a secure IoT application.

REFERENCES

[1] Y. Alsouda, S. Pllana, et al. 2019. IoT-based Urban Noise Identification Using Machine Learning: Performance of SVM, KNN, Bagging, and Random Forest. In *Proceedings of the International Conference on Omni-Layer Intelligent Systems*. ACM, 62–67.

[2] J. Boedecker, J. T. Springenberg, et al. 2014. Approximate real-time optimal control based on sparse gaussian process models. In *2014 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*. IEEE, 1–8.

[3] K. Bonawitz, V. Ivanov, et al. 2017. Practical secure aggregation for privacy-preserving machine learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1175–1191.

[4] D. Brauckhoff, K. Salamatian, et al. 2009. Applying PCA for traffic anomaly detection: Problems and solutions. In *IEEE INFOCOM 2009*. IEEE, 2866–2870.

[5] K. Chaudhuri and C. Monteleoni. 2009. Privacy-preserving logistic regression. In *NIPS (2009)*. 289–296.

[6] L. Chen, Y. Zhang, et al. 2018. Detection of DNS DDoS Attacks with Random Forest Algorithm on Spark. *Procedia computer science* 134 (2018), 310–315.

[7] P. Chopra and S. K. Yadav. 2015. Fault detection and classification by unsupervised feature extraction and dimensionality reduction. *Complex & Intelligent Systems* (2015).

[8] C.-T. Chu, S. K. Kim, et al. 2007. Map-reduce for machine learning on multicore. In *NIPS (2007)*. 281–288.

[9] K. Chua, R. Calandra, et al. 2018. Deep reinforcement learning in a handful of trials using probabilistic dynamics models. In *NIPS (2018)*. 4754–4765.

[10] M. Deisenroth and C. E. Rasmussen. 2011. PILCO: A model-based and data-efficient approach to policy search. In *Proceedings of the 28th International Conference on machine learning (ICML-11)*. 465–472.

[11] S. Devi and T. Neetha. 2017. Machine Learning based traffic congestion prediction in a IoT based Smart City. (2017).

[12] M. Ding and H. Tian. 2016. PCA-based network traffic anomaly detection. *Tsinghua Science and Technology* 21, 5 (2016), 500–509.

[13] W. Du and Z. Zhan. 2002. Building decision tree classifier on private data. In *Proceedings of the IEEE international conference on Privacy, security and data mining-Volume 14*. Australian Computer Society, Inc., 1–8.

[14] Y. Elovici, A. Shabtai, et al. 2007. Applying machine learning techniques for detection of malicious code in network traffic. In *Annual Conference on Artificial Intelligence*. Springer, 44–50.

[15] M. F. Elrawy, A. I. Awad, et al. 2018. Intrusion detection systems for IoT-based smart environments: a survey. *Journal of Cloud Computing* 7, 1 (2018), 21.

[16] M. Fredrikson, S. Jha, et al. 2015. Model inversion attacks that exploit confidence information and basic countermeasures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1322–1333.

[17] S. Fujimoto, H. van Hoof, et al. 2018. Addressing function approximation error in actor-critic methods. *arXiv preprint arXiv:1802.09477* (2018).

[18] Y. Gal, R. McAllister, et al. 2016. Improving PILCO with Bayesian neural network dynamics models. In *Data-Efficient Machine Learning workshop, ICML*, Vol. 4.

[19] M. Goldstein and S. Uchida. 2016. A comparative evaluation of unsupervised anomaly detection algorithms for multivariate data. *PloS one* 11, 4 (2016), e0152173.

[20] Y. Guan and T. Plötz. 2017. Ensembles of deep lstm learners for activity recognition using wearables. *IMWUT* 1, 2 (2017), 11.

[21] T. Haarnoja, A. Zhou, et al. 2018. Soft actor-critic algorithms and applications. *arXiv preprint arXiv:1812.05905* (2018).

[22] N. Heess, S. Sriram, et al. 2017. Emergence of locomotion behaviours in rich environments. *arXiv preprint arXiv:1707.02286* (2017).

[23] V. Hema. 2015. DoS Attack Detection Based on Naive Bayes Classifier. (2015).

[24] S. Hochreiter and J. Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[25] G. Jagannathan, K. Pillaipakkamnatt, et al. 2009. A practical differentially private random decision tree classifier. In *2009 IEEE International Conference on Data Mining Workshops*. IEEE, 114–121.

[26] M. M. Kirmani and S. I. Ansarullah. [n. d.]. Prediction of Heart Disease using Decision Tree a Data Mining Technique. ([n. d.]).

[27] J. Ko and D. Fox. 2009. GP-BayesFilters: Bayesian filtering using Gaussian process prediction and observation models. *Autonomous Robots* 27, 1 (2009), 75–90.

[28] V. R. Konda and J. N. Tsitsiklis. 2000. Actor-critic algorithms. In *NIPS (2000)*. 1008–1014.

[29] S. Lakshminarasimman, S. Ruswin, et al. 2017. Detecting DDoS attacks using decision tree algorithm. In *2017 Fourth International Conference on Signal Processing, Communication and Networking (ICSCN)*. IEEE, 1–6.

[30] S. Levine and P. Abbeel. 2014. Learning neural network policies with guided policy search under unknown dynamics. In *NIPS (2014)*. 1071–1079.

[31] W. Li, P. Yi, et al. 2014. A new intrusion detection system based on KNN classification algorithm in wireless sensor network. *Journal of Electrical and Computer Engineering* 2014 (2014).

[32] T. P. Lillicrap, J. J. Hunt, et al. 2015. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971* (2015).

[33] R. Lioutikov, A. Paraschos, et al. 2014. Sample-based informationl-theoretic stochastic optimal control. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 3896–3902.

[34] G. Louppe. 2014. Understanding random forests: From theory to practice. *arXiv preprint arXiv:1407.7502* (2014).

[35] C. Lyu, K. Huang, et al. 2015. A unified gradient regularization family for adversarial examples. In *2015 IEEE International Conference on Data Mining*. IEEE, 301–309.

[36] Y. Meidan, M. Bohadana, et al. 2017. Detection of unauthorized iot devices using machine learning techniques. *arXiv preprint arXiv:1709.04647* (2017).

[37] V. Mnih, A. P. Badia, et al. 2016. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*. 1928–1937.

[38] V. Mnih, K. Kavukcuoglu, et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (2015), 529.

[39] S. Mukherjee and N. Sharma. 2012. Intrusion detection using naive Bayes classifier with feature reduction. *Procedia Technology* 4 (2012), 119–128.

[40] R. Munos, T. Stepleton, et al. 2016. Safe and efficient off-policy reinforcement learning. In *NIPS (2016)*. 1054–1062.

[41] A. Nagabandi, G. Kahn, et al. 2018. Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 7559–7566.

[42] P. Naraei, A. Abhari, et al. 2016. Application of multilayer perceptron neural networks and support vector machines in classification of healthcare data. In *FTC*. 848–852.

[43] D. Oh and I. Yun. 2018. Residual error based anomaly detection using auto-encoder in smd machine sound. *Sensors* 18, 5 (2018), 1308.

[44] S. Peddabachigari, A. Abraham, et al. [n. d.]. Intrusion detection systems using decision trees and support vector machines. ([n. d.]).

[45] K. Peng, V. Leung, et al. 2018. Intrusion detection system based on decision tree over big data in fog environment. *Wireless Communications and Mobile Computing* 2018 (2018).

[46] J. R. Quinlan. 1986. Induction of decision trees. *Machine learning* 1, 1 (1986), 81–106.

[47] T. Rao, N. Rajasekhar, et al. [n. d.]. An efficient approach for weather forecasting using support vector machines.

[48] B. I. Rubinstein, P. L. Bartlett, et al. 2009. Learning in a large function space: Privacy-preserving mechanisms for SVM learning. *arXiv preprint arXiv:0911.5708* (2009).

[49] B. I. Rubinstein, B. Nelson, et al. 2009. Antidote: understanding and defending against poisoning of anomaly detectors. In *SIGCOMM on Internet measurement*. ACM.

[50] P. Samangouei, M. Kabkab, et al. 2018. Defense-gan: Protecting classifiers against adversarial attacks using generative models. *arXiv preprint arXiv:1805.06605* (2018).

[51] B. Sanjaa and E. Chuluun. 2013. Malware detection using linear SVM. In *Ifost*, Vol. 2. IEEE, 136–138.

[52] T. Schaul, J. Quan, et al. 2015. Prioritized experience replay. *arXiv preprint arXiv:1511.05952* (2015).

[53] J. Schulman, S. Levine, et al. 2015. Trust region policy optimization. In *International Conference on Machine Learning*. 1889–1897.

[54] J. Schulman, F. Wolski, et al. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).

[55] U. Shaham, Y. Yamada, et al. 2018. Understanding adversarial training: Increasing local stability of supervised models through robust optimization. *Neurocomputing* (2018).

[56] T. A. Shinde and J. R. Prasad. 2017. IoT based animal health monitoring with naive Bayes classification. *IJETT* 1, 2 (2017).

[57] R. Shokri and V. Shmatikov. 2015. Privacy-preserving deep learning. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*. ACM, 1310–1321.

[58] A. Shustanov and P. Yakimov. 2017. CNN design for real-time traffic sign recognition. *Procedia engineering* 201 (2017), 718–725.

[59] J. Steinhardt, P. W. W. Koh, et al. 2017. Certified defenses for data poisoning attacks. In *NIPS (2017)*. 3517–3529.

[60] J. Su and H. Zhang. 2006. A fast decision tree learning algorithm.

[61] M.-Y. Su. 2011. Real-time anomaly detection systems for Denial-of-Service attacks by weighted k-nearest-neighbor classifiers. *Expert Systems with Applications* 38, 4 (2011), 3492–3498.

[62] R. B. S. Swetha and K. G. Meena. [n. d.]. Smart Grid-A Network based Intrusion Detection System. *International Journal of Computer Applications* 975 ([n. d.]), 8887.

[63] F. Tramer, A. Kurakin, et al. 2017. Ensemble adversarial training: Attacks and defenses. *arXiv preprint arXiv:1705.07204* (2017).

[64] F. Tramer, F. Zhang, et al. 2016. Stealing machine learning models via prediction apis. In *25th USENIX Security Symposium (USENIX Security 16)*. 601–618.

[65] M. Vacher, B. Lecouteux, et al. 2015. Speech and speaker recognition for home automation: Preliminary results. In *SpeD*. IEEE.

[66] H. Van Hasselt, A. Guez, et al. 2016. Deep reinforcement learning with double q-learning. In *AAAI*.

[67] V. Vapnik. 2013. *The nature of statistical learning theory*. Springer science & business media.

[68] S. A. Vinterbo. 2012. Differentially private projected histograms: Construction and use for prediction. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 19–34.

[69] Z. Wang, T. Schaul, et al. 2015. Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581* (2015).

[70] C. J. Watkins and P. Dayan. 1992. Q-learning. *Machine learning* 8, 3-4 (1992), 279–292.

[71] W. Xu, D. Evans, et al. 2017. Feature squeezing: Detecting adversarial examples in deep neural networks. *arXiv preprint arXiv:1704.01155* (2017).

[72] J. Yang, Y. Chen, et al. 2008. Detecting sybil attacks inwireless and sensor networks using cluster analysis. In *2008 5th IEEE International Conference on Mobile Ad Hoc and Sensor Systems*. IEEE, 834–839.

[73] J. Zhang, Z. Zhang, et al. 2012. Functional mechanism: regression analysis under differential privacy. *Proceedings of the VLDB Endowment* 5, 11 (2012), 1364–1375.

[74] W. Zhang, W. Guo, et al. 2018. LSTM-based analysis of industrial IoT equipment. *IEEE Access* 6 (2018), 23551–23560.

[75] C. Zhao, C. Chen, et al. 2018. Application of Auxiliary Classifier Wasserstein Generative Adversarial Networks in Wireless Signal Classification of Illegal Unmanned Aerial Vehicles. *Applied Sciences* 8, 12 (2018), 2664.

[76] C. Zhao, M. Shi, et al. 2018. Research on the open-categorical classification of the Internet-of-things based on generative adversarial networks. *Applied Sciences* (2018).

[77] W. Zhao, S. Tang, et al. [n. d.]. An improved kNN algorithm based on essential vector. *Elektronika ir Elektrotechnika* 123, 7 ([n. d.]), 119–122.

[78] X. Zhao, Z. Gao, et al. 2011. A fault detection algorithm based on cluster analysis in wireless sensor networks. In *2011 Seventh International Conference on Mobile Ad-hoc and Sensor Networks*. IEEE, 354–355.