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 article 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.

CCS Concepts: • **General and reference** → **Surveys and overviews**; • **Distributed artificial intelligence** → *Cooperation and coordination*; • **Computer systems organization** → internet of things; • **Software and its engineering** → Development frameworks and environments;

Additional Key Words and Phrases: IoT, machine learning, deep learning, orchestration

ACM Reference format:
Bin Qian, Jie Su, Zhenyu Wen, Devki Nandan Jha, Yinhao Li, Yu Guan, Deepak Puthal, Philip James, Renyu Yang, Albert Y. Zomaya, Omer Rana, Lizhe Wang, Maciej Koutny, and Rajiv Ranjan. 2020. Orchestrating the Development Lifecycle of Machine Learning-based IoT Applications: A Taxonomy and Survey. *ACM Comput. Surv.* 53, 4, Article 82 (August 2020), 47 pages.
https://doi.org/10.1145/3398020

Authors' addresses: B. Qian, J. Su, Z. Wen (corresponding author), D. N. Jha, Y. Li, Yu Guan, D. Puthal, P. James, M. Koutny, and R. Ranjan are with School of Computing, Newcastle University, UK; emails: {b.qian3, j.su4, Zhenyu.Wen, D.N.Jha2, y.li119, yu.guan, deepak.puthal, philip.james, maciej.koutny, raj.ranjan}@newcastle.ac.uk; R. Yang is with School of Computing, University of Leeds, UK; email: r.yang1@leeds.ac.uk; A. Y. Zomaya is with School of Computer Science, The University of Sydney, Australia; email: albert.zomaya@sydney.edu.au; O. Rana is with School of Computer Science and Informatics, Cardiff University; email: ranaof@cardiff.ac.uk; L. Wang, School of Computer Science, China University of Geoscience, P. R China; email: lzwang@cug.edu.cn.

0360-0300/2020/08-ART82 $15.00
https://doi.org/10.1145/3398020

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 that 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 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, Reference [260] discussed encountered challenges of developing the next generation of AI systems. References [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 the 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. Figure 1 represents a smart city that consists of various IoT applications with many of them using Machine Learning (ML) techniques. For example, a *smart traffic 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]. However, a *smart car navigation system* [136] allows the passengers to set and change destinations via built-in car audio

[1]https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/.

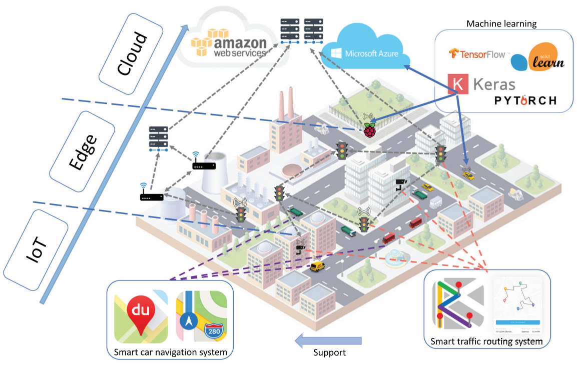

Fig. 1. Smart city.

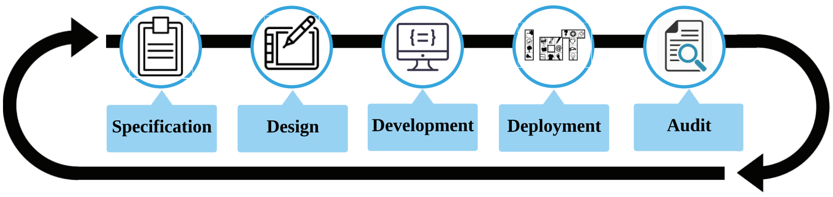

Fig. 2. The development lifecycle of an ML-based IoT application.

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. Figure 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.

Taxonomy. Figure 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 Reference [286] has reviewed cloud resource orchestration techniques. It outlines the

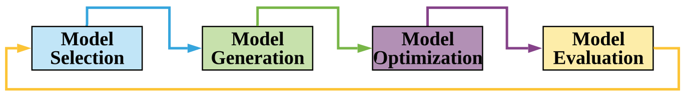

Fig. 3. A general pipeline of model development.

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 article 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 Section 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 Section 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 Section 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 Section 5).

2 MODEL DEVELOPMENT

One of the core components in this article 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 Figure 3), including *model selection*, *model generation*, *model optimization*, and *model evaluation*, and we explain with adaptive video streaming [183] as an example.

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 Figure 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 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** (Section 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 (Section 2.1.4). Next, we will choose a suitable development framework to implement the

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

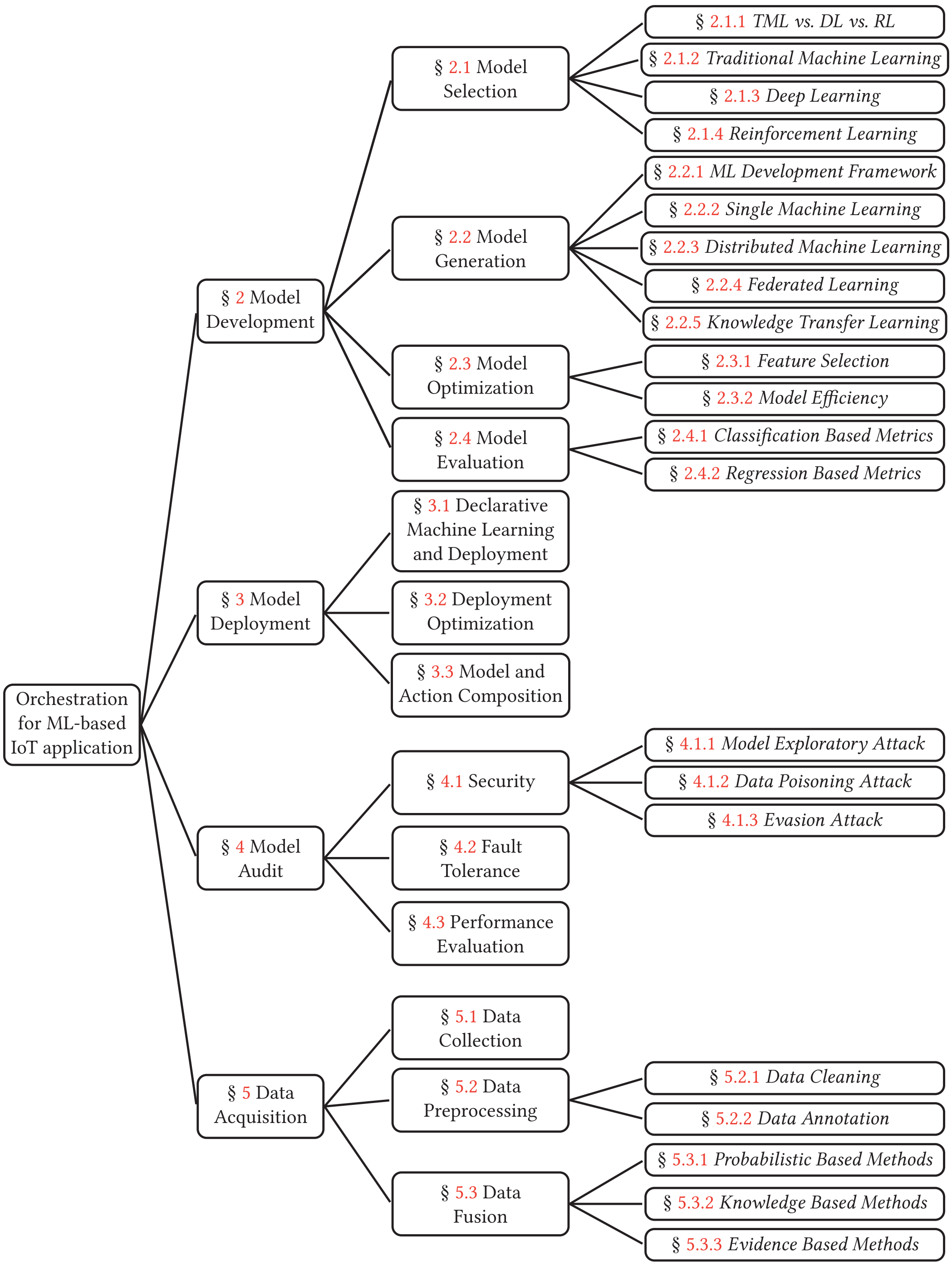
Orchestration for ML-based IoT application
§ 2 Model Development
§ 2.1 Model Selection
§ 2.1.1 TML vs. DL vs. RL
§ 2.1.2 Traditional Machine Learning
§ 2.1.3 Deep Learning
§ 2.1.4 Reinforcement Learning
§ 2.2 Model Generation
§ 2.2.1 ML Development Framework
§ 2.2.2 Single Machine Learning
§ 2.2.3 Distributed 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.

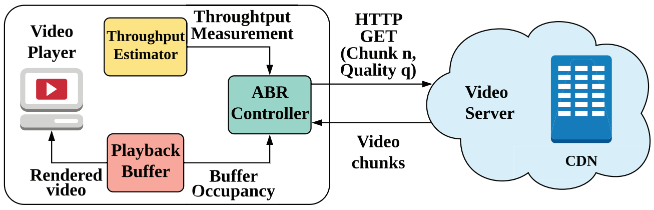

Fig. 5. Adaptive video streaming.

model and utilize different acceleration techniques to reduce the latency of **model generation** (Section 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** (Section 2.3). In **model evaluation** (Section 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 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.

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

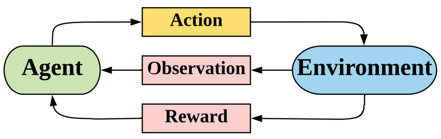

Fig. 6. Reinforcement learning paradigm.

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], and so on. In RL, there is usually one or more agent(s) interacting with the outside environment, where optimal control policies are learnt through experience. Figure 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.

Discussion. In IoT environments, a variety of problems can be modelled by using the aforementioned three approaches. The applications range from system and networking [182, 183], smart city [164, 320], to smart grid [235, 290], and so on. 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 that need interaction with environment or system such as smart temperature control systems, or recommendation systems (with cold start problem). However, 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, and so on, 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]. However, 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., Figure 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 $Y :\leftarrow f(X)$. The most representative algorithms include *Logistic Regression (LR)*, *Artificial Neural Networks (ANN)*, *Decision Tree (DT)* [222], *Random Forest (RF)* [36], and *Support Vector Machine (SVM)* [63]. The TML algorithms are based on mathematics and statistics modelling that give more interpretability for the model itself.

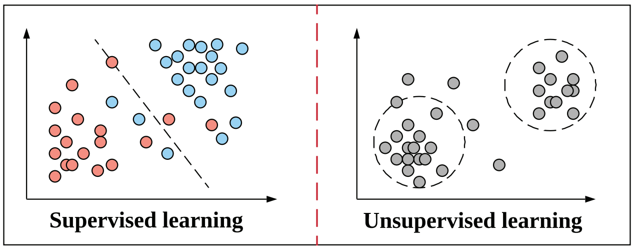

Fig. 7. Examples of supervised learning (linear regression) and unsupervised learning (clustering).

Unsupervised Learning. The unsupervised learning algorithm (see Figure 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, and so on. 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), and so on. 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, and so on. 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], and so on. 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], and so on.

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*, however, 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

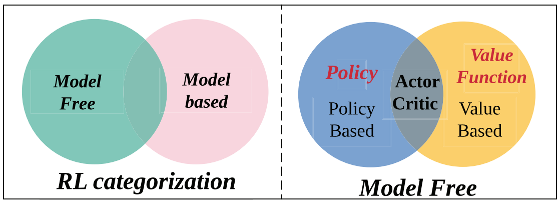

Fig. 8. Reinforcement learning categorization.

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 Section 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, and so on. 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 Reference [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, and so on.

As shown in Figure 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.

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 Reference [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 and DDQN. The continuous action space, however, 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 Reference [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, and so on. 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.

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.*

Table 1. Comparison of Deep Learning Frameworks

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

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), however, 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 cannot 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 cannot 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

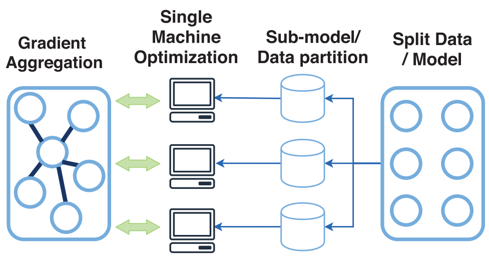

Fig. 9. Distributed machine learning pipeline.

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 Reference [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 that distribute and accelerate the machine learning workload.

Figure 9 shows the schematic diagram of a distributed ML pipeline. It has multiple components that 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 Section 2.2.2), which 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 cannot 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 cannot 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 Reference [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. Reference [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, Reference [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 Reference [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 cannot 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], and 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. Reference [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 Section 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 (Section 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 Reference [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, Reference [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, Reference [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 Reference [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], and 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].

Table 2. Confusion Matrix for Classification

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

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. However, *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.

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 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—then 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), and so on. 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 cannot 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 (Section 3.1). Next, the optimization techniques can be applied to generate the deployment plans that select the suitable ML models for the deployment, optimizes the resource utilization of the model deployment and improve the reusability of the deployed models (Section 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 (Section 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. Reference [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 Section 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 cannot handle complex deployments such as specifying the details of ML-based application. Moreover, the IoT applications consist of installing devices and sensors that 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. Reference [308] investigates the resource allocation problems solved by evolutionary approaches in cloud computing.

Machine learning-based method. The 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 Section 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, and so on. 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. Reference [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, and so on.
- 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 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 Reference [103] based on their proposed model. Reference [249] categorized the security issues of IoT into *eight* domains including authentication, access control, confidentiality, privacy, trust, secure middleware, mobile security, and policy enforcement. Reference [233] studied the main challenges and solutions of designing and deploying security mechanisms in centralized and distributed IoT architectures. Reference [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 **Appendix 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]. Reference [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, and so on. 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, and so on [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, then it also reveals that the patient has this disease. The experiments in Reference [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 References [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. Reference [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

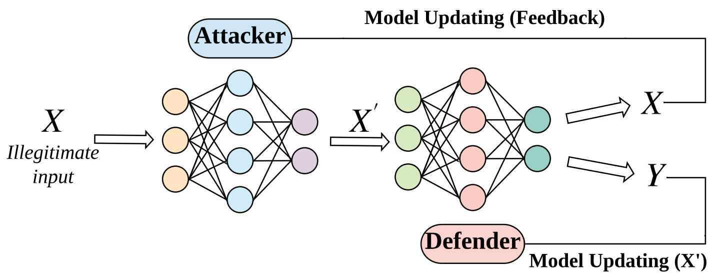

Fig. 10. Arms race game between the attacker and the defender. The attacker takes an illegitimate example (X) as the input of his/her neural network and generates an adversarial example (X'). This example attempts to fool the defender, which is a classification neural network. If the classifier recognizes the adversarial example as Y, which belongs to the legitimate input, then 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.

successful rate, time consumption and required resources (the number of modified samples) can be reduced significantly [50, 106, 258]. References [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. Figure 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 Figure 10, the defender aims to stop using the illegitimate input X to train the itself. 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 Reference [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. Of the adversarial examples generated by the local substitute model, 84% 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, and so on. To build a secure IoT application, we must consider the security issues from 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 Figure 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), and so on.

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., Reference [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, and so on.

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 been proposed 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 Reference [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 $\mathcal{N}_{fault}$. To guarantee the robustness of the neural model, the designer needs to ensure that the error (left-hand side in Equation (1)) is below a predefined threshold (right-hand side 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 Reference [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 Reference [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 Section 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] 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 Section 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 (Section 5.1). With proper preprocessing techniques (Section 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 (Section 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, and so on. However, *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, and so on. 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. References [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 that 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 Section 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 Section 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 Reference [283], *uncertainty*, *diversity*, *density*, and *relevance* are combined for multi-modality video annotation. Similarly, work in Reference [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 Reference [4]. First, it cannot scale with the size and modality of the data. Second, uncertain and noisy data cannot 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, References [22, 191] (supervised learning method) and References [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. References [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, then 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 Section 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] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. 265–283.

[2] Stefan Achleitner, Thomas La Porta, Trent Jaeger, and Patrick McDaniel. 2017. Adversarial network forensics in software defined networking. In *Proceedings of the Symposium on SDN Research*. ACM, 8–20.

[3] Alham Fikri Aji and Kenneth Heafield. 2017. Sparse communication for distributed gradient descent. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP'17)*. 440–445.

[4] Furqan Alam, Rashid Mehmood, Iyad Katib, Nasser N. Albogami, and Aiiad Albeshri. 2017. Data fusion and IoT for smart ubiquitous environments: A survey. *IEEE Access* 5 (2017), 9533–9554.

[5] Suad A. Alasadi and Wesam S. Bhaya. 2017. Review of data preprocessing techniques in data mining. *J. Eng. Appl. Sci.* 12, 16 (2017), 4102–4107.

[6] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. 2017. QSGD: Communication-efficient SGD via gradient quantization and encoding. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS'17)*. 1709–1720.

[7] Fayçal Ait Aoudia, Matthieu Gautier, and Olivier Berder. 2018. RLMan: An energy manager based on reinforcement learning for energy harvesting wireless sensor networks. *IEEE Trans. Green Commun. Netw.* 2, 2 (2018), 408–417.

[8] Jeremy Appleyard, Tomas Kocisky, and Phil Blunsom. 2016. Optimizing performance of recurrent neural networks on GPUs. *arXiv preprint arXiv:1604.01946* (2016).

[9] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. 2017. A brief survey of deep reinforcement learning. *arXiv preprint arXiv:1708.05866* (2017).

[10] Giuseppe Ateniese, Giovanni Felici, Luigi V. Mancini, Angelo Spognardi, Antonio Villani, and Domenico Vitali. 2013. Hacking smart machines with smarter ones: How to extract meaningful data from machine learning classifiers. *arXiv* (2013).

[11] Stéphane Ayache and Georges Quénot. 2007. Evaluation of active learning strategies for video indexing. *Signal Process.: Image Commun.* 22, 7–8 (2007), 692–704.

[12] Jimmy Ba, Roger Grosse, and James Martens. 2016. Distributed second-order optimization using Kronecker-factored approximations. In *5th International Conference on Learning Representations Toulon, France, April 24-26, 2017, Conference Track Proceedings*. https://openreview.net/forum?id=SkkTMpjex.

[13] Bowen Baker, Otkrist Gupta, Ramesh Raskar, and Nikhil Naik. 2017. Accelerating neural architecture search using performance prediction. *arXiv preprint arXiv:1705.10823* (2017).

[14] Anoop Korattikara Balan, Vivek Rathod, Kevin P. Murphy, and Max Welling. 2015. Bayesian dark knowledge. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS'15)*. 3438–3446.

[15] Pierre Baldi. 2012. Autoencoders, unsupervised learning, and deep architectures. In *Proceedings of ICML Workshop on Unsupervised and Transfer Learning*. 37–49.

[16] Roberto Battiti. 1994. Using mutual information for selecting features in supervised neural net learning. *IEEE Trans. Neural Netw.* 5, 4 (1994), 537–550.

[17] Amir Beck, Angelia Nedić, Asuman Ozdaglar, and Marc Teboulle. 2014. An $o(1/k)$ gradient method for network resource allocation problems. *IEEE Trans. Control Netw. Syst.* 1, 1 (2014), 64–73.

[18] Tal Ben-Nun, Ely Levy, Amnon Barak, and Eri Rubin. 2015. Memory access patterns: The missing piece of the multi-GPU puzzle. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'15)*. IEEE, 1–12.

[19] Gyora M. Benedek and Alon Itai. 1991. Learnability with respect to fixed distributions. *Theoret. Comput. Sci.* 86, 2 (1991), 377–389.

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

[21] Josep Ll Berral, Íñigo Goiri, Ramón Nou, Ferran Julià, Jordi Guitart, Ricard Gavaldà, and Jordi Torres. 2010. Towards energy-aware scheduling in data centers using machine learning. In *Proceedings of the Conference on E-Energy*. ACM, 215–224.

[22] Behnaz Bigdeli and Peter Reinartz. 2015. Fusion of hyperspectral and LIDAR data using decision template-based fuzzy multiple classifier system. *Int. J. Appl. Earth Obs.* 38 (2015), 309–320.

[23] Battista Biggio, Blaine Nelson, and Pavel Laskov. 2012. Poisoning attacks against support vector machines. *arXiv preprint arXiv:1206.6389* (2012).

[24] Battista Biggio, Konrad Rieck, Davide Ariu, Christian Wressnegger, Igino Corona, Giorgio Giacinto, and Fabio Roli. 2014. Poisoning behavioral malware clustering. In *Proceedings of the Workshop on Artificial Intelligence and Security Workshop*. ACM, 27–36.

[25] Farshid Hassani Bijarbooneh, Wei Du, Edith C.-H. Ngai, Xiaoming Fu, and Jiangchuan Liu. 2016. Cloud-assisted data fusion and sensor selection for internet of things. *IEEE Internet Things J.* 3, 3 (2016), 257–268.

[26] Tewodros A. Biresaw, Andrea Cavallaro, and Carlo S. Regazzoni. 2015. Tracker-level fusion for robust Bayesian visual tracking. *IEEE Trans. Circ. Syst. Video Technol.* 25, 5 (2015).

[27] Ane Blázquez-García, Angel Conde, Usue Mori, and Jose A. Lozano. 2020. A review on outlier/anomaly detection in time series data. *arXiv preprint arXiv:2002.04236* (2020).

[28] Matthias Boehm, Michael W. Dusenberry, Deron Eriksson, Alexandre V. Evfimievski, Faraz Makari Manshadi, Niketan Pansare, Berthold Reinwald, Frederick R. Reiss, Prithviraj Sen, Arvind C. Surve, et al. 2016. Systemml: Declarative machine learning on spark. *Proc. VLDB Endow.* 9, 13 (2016), 1425–1436.

[29] Keith Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloe Kiddon, Jakub Konecny, Stefano Mazzocchi, H. Brendan McMahan, et al. 2019. Towards federated learning at scale: System design. *arXiv preprint arXiv:1902.01046* (2019).

[30] Keith Bonawitz, Fariborz Salehi, Jakub Konečný, Brendan McMahan, and Marco Gruteser. 2019. Federated learning with autotuned communication-efficient secure aggregation. *arXiv preprint arXiv:1912.00131* (2019).

[31] Leon Bottou. 2010. Large-scale machine learning with stochastic gradient descent. In *Proceedings of the International Conference on Computational Statistics (COMPSTAT'10).* Springer, 177–186.

[32] Léon Bottou. 2012. Stochastic gradient descent tricks. In *Neural Networks: Tricks of the Trade.* Springer, 421–436.

[33] Léon Bottou, Frank E. Curtis, and Jorge Nocedal. 2018. Optimization methods for large-scale machine learning. *SIAM Rev.* 60, 2 (2018), 223–311.

[34] Olivier Bousquet, Stephane Boucheron, and Gabor Lugosi. 2003. Introduction to statistical learning theory. In *Proceedings of the Summer School on Machine Learning*. Springer, 169–207.

[35] Joseph K Bradley, Aapo Kyrola, Danny Bickson, and Carlos Guestrin. 2011. Parallel coordinate descent for l1-regularized loss minimization. *arXiv preprint arXiv:1105.5379* (2011).

[36] Leo Breiman. 2001. Random forests. *Mach. Learn.* 45, 1 (2001), 5–32.

[37] Leo Breiman. 2017. *Classification and Regression Trees.* Routledge.

[38] Uwe Breitenbucher, Kalman Kepes, Frank Leymann, and Michael Wurster. 2017. Declarative vs. imperative: How to model the automated deployment of IoT applications? In*Proceedings of the Summer School on Service Oriented Computing (SummerSOC'17)*, 18–27.

[39] Andrew Brock, Theodore Lim, James M. Ritchie, and Nick Weston. 2017. SMASH: One-shot model architecture search through hypernetworks. *arXiv preprint arXiv:1708.05344* (2017).

[40] Richard H. Byrd, Samantha L. Hansen, Jorge Nocedal, and Yoram Singer. 2016. A stochastic quasi-Newton method for large-scale optimization. *SIAM J. Optimiz.* 26, 2 (2016), 1008–1031.

[41] Benjamin Cabé, Eclipse IoT Working Group, et al. 2018. IoT developer survey 2018. *SlideShare, April* 13 (2018).

[42] Miguel Castro, Barbara Liskov, et al. 1999. Practical Byzantine fault tolerance. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'99)*, Vol. 99. 173–186.

[43] Z. Berkay Celik, Leonardo Babun, Amit Kumar Sikder, Hidayet Aksu, Gang Tan, Patrick McDaniel, and A. Selcuk Uluagac. 2018. Sensitive information tracking in commodity IoT. In *Proceedings of the 27th USENIX Security Symposium (USENIXSecurity'18).* 1687–1704.

[44] Nitesh V. Chawla, Nathalie Japkowicz, and Aleksander Kotcz. 2004. Special issue on learning from imbalanced data sets. *ACM SIGKDD Explor. Newslett.* 6, 1 (2004), 1–6.

[45] Kumar Chellapilla, Sidd Puri, and Patrice Simard. 2006. High performance convolutional neural networks for document processing. In *Proceedings of the International Conference on Frontiers in Handwriting Recognition (IWFHR'06).* Suvisoft.

[46] Chi-Chung Chen, Chia-Lin Yang, and Hsiang-Yun Cheng. 2018. Efficient and robust parallel DNN training through model parallelism on multi-GPU platform. *arXiv preprint arXiv:1809.02839* (2018).

[47] Jianmin Chen, Xinghao Pan, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. 2016. Revisiting distributed synchronous SGD. *arXiv preprint arXiv:1604.00981* (2016).

[48] Lei Chen, Jiwen Lu, Zhanjie Song, and Jie Zhou. 2018. Part-activated deep reinforcement learning for action prediction. In *Proceedings of the European Conference on Computer Vision (ECCV'18).* 421–436.

[49] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18).* 578–594.

[50] Xinyun Chen, Chang Liu, Bo Li, Kimberly Lu, and Dawn Song. 2017. Targeted backdoor attacks on deep learning systems using data poisoning. *arXiv preprint arXiv:1712.05526* (2017).

[51] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. 2017. A survey of model compression and acceleration for deep neural networks. *arXiv preprint arXiv:1710.09282* (2017).

[52] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* (2014).

[53] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. 2014. Project adam: Building an efficient and scalable deep learning training system. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14).* 571–582.

[54] Ching-Tai Chin, Kishan Mehrotra, Chilukuri K. Mohan, S. Rankat, et al. 1994. Training techniques to obtain fault-tolerant neural networks. In *Proceedings of the International Symposium on Fault-Tolerant Computing (FTCS'94).* IEEE, 360–369.

[55] C.-T. Chiu, Kishan Mehrotra, Chilukuri K. Mohan, and Sanjay Ranka. 1993. Robustness of feedforward neural networks. In *Proceedings of the IEEE International Conference on Neural Networks*. IEEE, 783–788.

[56] François Chollet. 2017. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'17)*. 1251–1258.

[57] Praveen Chopra and Sandeep Kumar Yadav. 2015. Fault detection and classification by unsupervised feature extraction and dimensionality reduction. *Complex Intell. Systems* 1, 1–4 (2015), 25–33.

[58] Cheng-Tao Chu, Sang K. Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Kunle Olukotun, and Andrew Y. Ng. 2007. Map-reduce for machine learning on multicore. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS' (2007)*. 281–288.

[59] L.-C. Chu and Benjamin W. Wah. 1990. Fault tolerant neural networks with hybrid redundancy. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN'90)*. IEEE.

[60] Igor Colin, Ludovic Dos Santos, and Kevin Scaman. 2019. Theoretical limits of pipeline parallel optimization and application to distributed deep learning. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS'19)*. 12350–12359.

[61] Graham Cormode, Minos Garofalakis, Peter J. Haas, Chris Jermaine, et al. 2011. Synopses for massive data: Samples, histograms, wavelets, sketches. *Found. Trends Databases* 4, 1–3 (2011), 1–294.

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

[63] Corinna Cortes and Vladimir Vapnik. 1995. Support-vector networks. *Mach. Learn.* 20, 3 (1995), 273–297.

[64] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv* (2016).

[65] Daniel Crankshaw, Gur-Eyal Sela, Corey Zumar, Xiangxi Mo, Joseph E. Gonzalez, Ion Stoica, and Alexey Tumanov. 2018. InferLine: ML inference pipeline composition framework. *arXiv preprint arXiv:1812.01776* (2018).

[66] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. 2017. Clipper: A low-latency online prediction serving system. In *Proceedings of the 14th USENIX Symposium on NSDI*. 613–627.

[67] Henggang Cui, Hao Zhang, Gregory R. Ganger, Phillip B. Gibbons, and Eric P. Xing. 2016. Geeps: Scalable deep learning on distributed GPUs with a GPU-specialized parameter server. In *Proceedings of the European Conference on Computer Systems (EuroSys'16)*. 1–16.

[68] Ana Cristina Franco da Silva, Uwe Breitenbücher, Pascal Hirmer, Kálmán Képes, Oliver Kopp, Frank Leymann, Bernhard Mitschang, and Ronald Steinke. 2017. Internet of Things out of the box: Using TOSCA for automating the deployment of IoT environments. In *Proceedings of the International Conference on Cloud Computing and Services Science (CLOSER'17)*. 330–339.

[69] Manoranjan Dash and Huan Liu. 2003. Consistency-based search in feature selection. *Artific. Intell.* 151, 1–2 (2003), 155–176.

[70] Roberto Souto Maior de Barros and Silas Garrido T. de Carvalho Santos. 2019. An overview and comprehensive comparison of ensembles for concept drift. *Info. Fusion* 52 (2019), 213–244.

[71] Christopher M De Sa, Ce Zhang, Kunle Olukotun, and Christopher Ré. 2015. Taming the wild: A unified analysis of hogwild-style algorithms. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS'15)*. 2674–2682.

[72] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V. Le, et al. 2012. Large scale distributed deep networks. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS'12)*. 1223–1231.

[73] Ofer Dekel, Ran Gilad-Bachrach, Ohad Shamir, and Lin Xiao. 2012. Optimal distributed online prediction using mini-batches. *J. Mach. Learn. Res.* 13 (Jan. 2012), 165–202.

[74] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 248–255.

[75] Suguna Devi and T. Neetha. 2017. Machine learning-based traffic congestion prediction in a IoT-based smart city. *Int. Res. J. Eng. Technol.* 4 (2017), 3442–3445.

[76] Greg Diamos, Shubho Sengupta, Bryan Catanzaro, Mike Chrzanowski, Adam Coates, Erich Elsen, Jesse Engel, Awni Hannun, and Sanjeev Satheesh. 2016. Persistent rnns: Stashing recurrent weights on-chip. In *Proceedings of the International Conference on Machine Learning*. 2024–2033.

[77] Jin-Dong Dong, An-Chieh Cheng, Da-Cheng Juan, Wei Wei, and Min Sun. 2018. Dpp-net: Device-aware progressive search for pareto-optimal neural architectures. In *Proceedings of the European Conference on Computer Vision (ECCV'18)*. 517–531.

[78] Wei Dong, Chun Chen, Jiajun Bu, Xue Liu, and Yunhao Liu. 2013. D2: Anomaly detection and diagnosis in networked embedded systems by program profiling and symptom mining. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS'13)*. IEEE, 202–211.

[79] Nikoli Dryden, Naoya Maruyama, Tom Benson, Tim Moon, Marc Snir, and Brian Van Essen. 2019. Improving strong-scaling of CNN training by exploiting finer-grained parallelism. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'19)*. IEEE, 210–220.

[80] Nikoli Dryden, Naoya Maruyama, Tim Moon, Tom Benson, Marc Snir, and Brian Van Essen. 2019. Channel and filter parallelism for large-scale CNN training. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'19)*. 1–20.

[81] Nikoli Dryden, Tim Moon, Sam Ade Jacobs, and Brian Van Essen. 2016. Communication quantization for data-parallel training of deep neural networks. In *Proceedings of the 2nd Workshop on Machine Learning in HPC Environments (MLHPC'16)*. IEEE, 1–8.

[82] John Duchi, Elad Hazan, and Yoram Singer. 2011. Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.* 12 (July 2011), 2121–2159.

[83] Santiago Egea, Albert Rego Mañez, Belen Carro, Antonio Sánchez-Esguevillas, and Jaime Lloret. 2017. Intelligent IoT traffic classification using novel search strategy for fast-based-correlation feature selection in industrial environments. *IEEE Internet Things J.* 5, 3 (2017), 1616–1624.

[84] Anne Elk. 2019. Distributed Machine Learning Toolkit: Big Data, Big Model, Flexibility, Efficiency. Retrieved from http://www.dmtk.io.

[85] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. 2018. Multi-objective architecture search for cnns. *arXiv preprint arXiv:1804.09081* 2 (2018).

[86] Martin D. Emmerson et al. 1993. Determining and improving the fault tolerance of multilayer perceptrons in a pattern-recognition application. *IEEE Trans. Neural Netw.* 4, 5 (1993), 788–793.

[87] Tugba Erpek, Yalin E. Sagduyu, and Yi Shi. 2018. Deep learning for launching and mitigating wireless jamming attacks. *TCCN* (2018).

[88] Sharanya Eswaran, Archan Misra, Flavio Bergamaschi, and Thomas La Porta. 2012. Utility-based bandwidth adaptation in mission-oriented wireless sensor networks. *TOSN* 8, 2 (2012), 1–26.

[89] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. 2010. The pascal visual object classes (VOC) challenge. *Int. J. Comput. Vision* 88, 2 (June 2010), 303–338.

[90] Hassan Ismail Fawaz, Germain Forestier, Jonathan Weber, Lhassane Idoumghar, and Pierre-Alain Muller. 2019. Deep learning for time series classification: A review. *Data Min. Knowl. Discov.* 33, 4 (2019), 917–963.

[91] George Frewat, Charbel Baroud, Roy Sammour, Abdallah Kassem, and Mustapha Hamad. 2016. Android voice recognition application with multi speaker feature. In *Proceedings of the 18th IEEE Mediterranean Electrotechnical Conference (MELECON'16)*. IEEE, 1–5.

[92] Colleen E. Fuss, Aaron A. Berg, and John B. Lindsay. 2016. DEM Fusion using a modified k-means clustering algorithm. *Int. J. Dig. Earth* 9, 12 (2016), 1242–1255.

[93] João Gama, Indrundefined Žliobaitundefined, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. 2014. A survey on concept drift adaptation. *ACM Comput. Surv.* 46, 4 (Mar. 2014).

[94] Peter Garraghan, Renyu Yang, Zhenyu Wen, Alexander Romanovsky, Jie Xu, Rajkumar Buyya, and Rajiv Ranjan. 2018. Emergent failures: Rethinking cloud reliability at scale. *IEEE Cloud Comput.* 5, 5 (2018), 12–21.

[95] Alexander L. Gaunt, Matthew A. Johnson, Maik Riechert, Daniel Tarlow, Ryota Tomioka, Dimitrios Vytiniotis, and Sam Webster. 2017. AMPNet: Asynchronous model-parallel training for dynamic neural networks. *arXiv preprint arXiv:1705.09786* (2017).

[96] Amir Gholami, Ariful Azad, Peter Jin, Kurt Keutzer, and Aydin Buluc. 2018. Integrated model, batch, and domain parallelism in training neural networks. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures (SPAA'18)*. 77–86.

[97] R. A. Gilyazev and D. Yu Turdakov. 2018. Active learning and crowdsourcing: A survey of optimization methods for data labeling. *Prog. Comput. Soft.* 44, 6 (2018), 476–491.

[98] David E. Goldberg. 1989. *Genetic Algorithms in Search, Optimization and Machine Learning* (1st ed.). Addison-Wesley Longman Publishing, Boston, MA.

[99] Heitor M. Gomes, Albert Bifet, Jesse Read, Jean Paul Barddal, Fabrício Enembreck, Bernhard Pfharinger, Geoff Holmes, and Talel Abdessalem. 2017. Adaptive random forests for evolving data stream classification. *Mach. Learn.* 106, 9-10 (2017), 1469–1495.

[100] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative adversarial nets. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS'14)*. 2672–2680.

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

[102] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677* (2017).

[103] Jorge Granjal, Edmundo Monteiro, and Jorge Sá Silva. 2010. A secure interconnection model for IPv6 enabled wireless sensor networks. In *Proceedings of the International Federation for Information Processing Wireless Days (IFIP'10)*. IEEE, 1–6.

[104] Alex Graves, Abdel Rahman Mohamed, and Geoffrey E. Hinton. 2013. Speech recognition with deep recurrent neural networks. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP'13)*. 6645–6649.

[105] Maciej Grzenda, Heitor Murilo Gomes, and Albert Bifet. 2019. Delayed labelling evaluation for data streams. *Data Min. Knowl. Discov.* (2019), 1–30.

[106] Tianyu Gu, Brendan Dolan-Gavitt, and Siddharth Garg. 2017. Badnets: Identifying vulnerabilities in the machine learning model supply chain. *arXiv preprint arXiv:1708.06733* (2017).

[107] Lei Guan, Wotao Yin, Dongsheng Li, and Xicheng Lu. 2019. XPipe: Efficient pipeline model parallelism for multi-GPU DNN training. *arXiv preprint arXiv:1911.04610* (2019).

[108] Yu Guan and Thomas Plötz. 2017. Ensembles of deep lstm learners for activity recognition using wearables. *Interact. Mobile Wear. Ubiq. Technol.* 1, 2 (2017), 11.

[109] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep learning with limited numerical precision. In *Proceedings of the International Conference on Machine Learning (ICML'15)*. 1737–1746.

[110] Ricardo Gutierrez-Osuna. 2002. Pattern analysis for machine olfaction: A review. *IEEE Sensors journal* (2002).

[111] Hamed Haddad, Ali Dehghan, Raouf Khayami, and Kim-Kwang Raymond Choo. 2018. A deep Recurrent Neural Network-based approach for Internet of Things malware threat hunting. *Future Gen. Comput. Syst.* 85 (2018), 88–96.

[112] Mark A. Hall. 2000. Correlation-based feature selection of discrete and numeric class machine learning. In *Proceedings of the Seventeenth International Conference on Machine Learning (ICML'00)*. Morgan Kaufmann Publishers Inc., 359–366.

[113] Yi Han, David Hubczenko, Paul Montague, Olivier De Vel, Tamas Abraham, Benjamin IP Rubinstein, Christopher Leckie, Tansu Alpcan, and Sarah Erfani. 2019. Adversarial reinforcement learning under partial observability in software-defined networking. *arXiv:1902.09062* (2019).

[114] Pierre Hansen, Nenad Mladenović, and José A. Moreno Pérez. 2010. Variable neighbourhood search: Methods and applications. *Ann. Operat. Res.* 175, 1 (2010), 367–407.

[115] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. 2018. Pipedream: Fast and efficient pipeline parallel dnn training. *arXiv preprint arXiv:1806.03377* (2018).

[116] John A. Hartigan and Manchek A. Wong. 1979. Algorithm AS 136: A k-means clustering algorithm. *J. Roy. Stat. Soc. Ser. C (Appl. Stat.)* 28, 1 (1979).

[117] Kaiming He, Georgia Gkioxari, Piotr Dollar, and Ross B. Girshick. 2017. Mask R-CNN. *Proceedings of the IEEE International Conference on Computer Vision (ICCV'17)*. 2980–2988.

[118] Xi He, Dheevatsa Mudigere, Mikhail Smelyanskiy, and Martin Takác. 2017. Distributed hessian-free optimization for deep neural network. In *Proceedings of the Workshops at the 31st AAAI Conference on Artificial Intelligence*.

[119] Geoffrey Hinton. 2019. RMSprop. Retrieved from http://www.cs.toronto.edu/ tijmen/csc321/slides/lecture_slides_lec6.pdf.

[120] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531* (2015).

[121] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B. Gibbons, Garth A. Gibson, Greg Ganger, and Eric P Xing. 2013. More effective distributed ml via a stale synchronous parallel parameter server. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS'13)*. 1223–1231.

[122] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Comput.* 9, 8 (1997), 1735–1780.

[123] Steven C. H. Hoi, Rong Jin, Jianke Zhu, and Michael R. Lyu. 2006. Batch mode active learning and its application to medical image classification. In *Proceedings of the 23rd International Conference on Machine Learning (ICML'06)*. ACM, 417–424.

[124] Steven C. H. Hoi, Doyen Sahoo, Jing Lu, and Peilin Zhao. 2018. Online learning: A comprehensive survey. *arXiv preprint arXiv:1802.02871* (2018).

[125] Lu Hou, Ruiliang Zhang, and James T. Kwok. 2019. Analysis of quantized models. In *Proceedings of the International Conference on Learning Representations*. Retrieved from https://openreview.net/forum?id=ryM_IoAqYX.

[126] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[127] Kevin Hsieh, Aaron Harlap, Nandita Vijaykumar, Dimitris Konomis, Gregory R. Ganger, Phillip B. Gibbons, and Onur Mutlu. 2017. Gaia: Geo-distributed machine learning approaching LAN speeds. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*. 629–647.

[128] Chi-Hung Hsu, Shu-Huan Chang, Jhao-Hong Liang, Hsin-Ping Chou, Chun-Hao Liu, Shih-Chieh Chang, Jia-Yu Pan, Yu-Ting Chen, Wei Wei, and Da-Cheng Juan. 2018. Monas: Multi-objective neural architecture search using reinforcement learning. *arXiv preprint arXiv:1806.10332* (2018).

[129] Hanqing Hu, Mehmed Kantardzic, and Tegjyot S. Sethi. 2019. No free lunch theorem for concept drift detection in streaming data classification: A review. *Wires data Min. Knowl.* 10, 2 (2019), e1327.

[130] Tiansi Hu and Yunsi Fei. 2010. QELAR: A machine-learning-based adaptive routing protocol for energy-efficient and lifetime-extended underwater sensor networks. *IEEE Transactions on Mobile Computing* 9, 6 (2010), 796–809.

[131] Gao Huang, Shichen Liu, Laurens Van der Maaten, and Kilian Q. Weinberger. 2018. Condensenet: An efficient densenet using learned group convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'18)*. 2752–2761.

[132] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. 2019. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS'19)*. 103–112.

[133] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2017. Quantized neural networks: Training neural networks with low precision weights and activations. *J. Mach. Learn. Res.* 18, 1 (2017), 6869–6898.

[134] Zhouyuan Huo, Bin Gu, Qian Yang, and Heng Huang. 2018. Decoupled parallel backpropagation with convergence guarantee. *arXiv preprint arXiv:1804.10574* (2018).

[135] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167* (2015).

[136] Jozef Ivanecky and Stephan Mehlhase. 2012. An in-car speech recognition system for disabled drivers. In *Proceedings of the International Conference on Text, Speech and Dialogue*. Springer, 505–512.

[137] Homayoun Jamshidi, Thomas Lukaszewicz, Amin Kashi, Ansel Berghuvud, Hans-Jurgen Zepernick, and Siamak Khatibi. 2011. Fusion of digital map traffic signs and camera-detected signs. In *Proceedings of the International Conference on Signal Processing and Communication Systems (ICSPCS'11)*. IEEE, 1–7.

[138] Sylvain Jeaugey. 2017. NCCL 2.0. Retrieved from http://on-demand.gputechconf.com/gtc/2017/presentation/s7155-jeaugey-nccl.pdf.

[139] Devki Nandan Jha, Peter Michalak, Zhenyu Wen, Paul Watson, and Rajiv Ranjan. 2019. Multi-objective deployment of data analysis operations in heterogeneous IoT infrastructure. *IEEE Trans. Industr. Inform.* (2019), 1–1.

[140] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2018. Beyond data and model parallelism for deep neural networks. *arXiv preprint arXiv:1807.05358* (2018).

[141] Peter Jin, Boris Ginsburg, and Kurt Keutzer. 2018. Spatially parallel convolutions. In *6th International Conference on Learning Representations (ICLR'17)*. https://openreview.net/forum?id=S1Yt0d1vG.

[142] Ajay J. Joshi, Fatih Porikli, and Nikolaos Papanikolopoulos. 2009. Multi-class active learning for image classification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 2372–2379.

[143] Delphine Jouan-Rimbaud, Desire-Luc Massart, Riccardo Leardi, and Onno E. De Noord. 1995. Genetic algorithms as a tool for wavelength selection in multivariate calibration. *Analyt. Chem.* 67, 23 (1995), 4295–4301.

[144] Peter Kairouz, H. Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Keith Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings et al. 2019. Advances and open problems in federated learning. *arXiv preprint arXiv:1912.04977* (2019).

[145] Ronald Kemker, Marc McClure, Angelina Abitino, Tyler L. Hayes, and Christopher Kanan. 2018. Measuring catastrophic forgetting in neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence.*

[146] Hanjoo Kim, Minkyu Kim, Dongjoo Seo, Jinwoong Kim, Heungseok Park, Soeun Park, Hyunwoo Jo, KyungHyun Kim, Youngil Yang, Youngkwan Kim, et al. 2018. Nsml: Meet the mlaas platform with a real-world case study. *arXiv preprint arXiv:1810.09957* (2018).

[147] Jin Kyu Kim, Qirong Ho, Seunghak Lee, Xun Zheng, Wei Dai, Garth A. Gibson, and Eric P. Xing. 2016. STRADS: A distributed framework for scheduled model parallel machine learning. In *Proceedings of the European Conference on Computer Systems (EuroSys'16)*. 1–16.

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

[149] Constantinos Kolias, Vasilis Kolias, and Georgios Kambourakis. 2017. TermID: A distributed swarm intelligence-based approach for wireless intrusion detection. *International Journal of Information Security* 16, 4 (2017), 401–416.

[150] Vijay R. Konda and John N. Tsitsiklis. 2000. Actor-critic algorithms. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS'00)*. 1008–1014.

[151] Jakub Konečnỳ, H. Brendan McMahan, Daniel Ramage, and Peter Richtárik. 2016. Federated optimization: Distributed machine learning for on-device intelligence. *arXiv preprint arXiv:1610.02527* (2016).

[152] Weicong Kong, Zhao Yang Dong, Youwei Jia, David J. Hill, Yan Xu, and Yuan Zhang. 2017. Short-term residential load forecasting based on LSTM recurrent neural network. *IEEE Trans. Smart Grid* 10, 1 (2017), 841–851.

[153] Nupur Kothari, Todd Millstein, and Ramesh Govindan. 2008. Deriving state machines from TinyOS programs using symbolic execution. In *Proceedings of the International Conference on Information Processing in Sensor Networks (IPSN'08)*. IEEE Computer Society.

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

[155] Veljko Krunic, Eric Trumpler, and Richard Han. 2007. NodeMD: Diagnosing node-level faults in remote wireless sensor systems. In *Proceedings of the ACM International Conference on Mobile Systems, Applications, and Services (MobiSys'07)*. ACM, 43–56.

[156] Pradeep Kumar, Himaanshu Gauba, Partha Pratim Roy, and Debi Prosad Dogra. 2017. Coupled HMM-based multi-sensor data fusion for sign language recognition. *Pattern Recogn. Lett.* 86 (2017), 1–8.

[157] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. 2016. Adversarial machine learning at scale. *arXiv preprint arXiv:1611.01236* (2016).

[158] Billy Pik Lik Lau, Sumudu Hasala Marakkalage, Yuren Zhou, Naveed Ul Hassan, Chau Yuen, Meng Zhang, and U-Xuan Tan. 2019. A survey of data fusion in smart city applications. *Info. Fusion* 52 (2019), 357–374.

[159] Vadim Lebedev, Yaroslav Ganin, Maksim Rakhuba, Ivan Oseledets, and Victor Lempitsky. 2014. Speeding-up convolutional neural networks using fine-tuned cp-decomposition. *arXiv preprint arXiv:1412.6553* (2014).

[160] Seunghak Lee, Jin Kyu Kim, Xun Zheng, Qirong Ho, Garth A. Gibson, and Eric P. Xing. 2014. On model parallelization and scheduling strategies for distributed machine learning. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS'14)*. 2834–2842.

[161] Hongyang Li, Wanli Ouyang, and Xiaogang Wang. 2016. Multi-bias non-linear activation in deep neural networks. In *Proceedings of the International Conference on Machine Learning*. 221–229.

[162] Ji Li, Hui Gao, Tiejun Lv, and Yueming Lu. 2018. Deep reinforcement learning-based computation offloading and resource allocation for MEC. In *Proceedings of the Wireless Communications and Networking Conference (WCNC'18)*. IEEE, 1–6.

[163] Jiwei Li, Will Monroe, Alan Ritter, Michel Galley, Jianfeng Gao, and Dan Jurafsky. 2016. Deep reinforcement learning for dialogue generation. *arXiv preprint arXiv:1606.01541* (2016).

[164] Li Li, Yisheng Lv, and Fei-Yue Wang. 2016. Traffic signal timing via deep reinforcement learning. *IEEE/CAA J. Automat. Sinica* 3, 3 (2016), 247–254.

[165] Liping Li, Wei Xu, Tianyi Chen, Georgios B. Giannakis, and Qing Ling. 2019. Rsa: Byzantine-robust stochastic aggregation methods for distributed learning from heterogeneous datasets. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 1544–1551.

[166] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. 2014. Scaling distributed machine learning with the parameter server. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*. 583–598.

[167] Mu Li, David G. Andersen, Alexander J. Smola, and Kai Yu. 2014. Communication efficient distributed machine learning with the parameter server. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS' (2014)*.

[168] Rui Li, Chaoyun Zhang, Paul Patras, Razvan Stanica, and Fabrice Valois. 2019. Learning driven mobility control of airborne base stations in emergency networks. *ACM SIGMETRICS Perform. Eval. Rev.* 46, 3 (2019), 163–166.

[169] Jie Lin, Wei Yu, Nan Zhang, Xinyu Yang, Hanlin Zhang, and Wei Zhao. 2017. A survey on internet of things: Architecture, enabling technologies, security and privacy, and applications. *IEEE Internet Things J.* 4, 5 (2017), 1125–1142.

[170] Tao Lin, Sebastian U. Stich, Kumar Kshitij Patel, and Martin Jaggi. 2018. Don't use large mini-batches, use local SGD. *arXiv preprint arXiv:1808.07217* (2018).

[171] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J. Dally. 2017. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887* (2017).

[172] Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. 2018. Progressive neural architecture search. In *Proceedings of the European Conference on Computer Vision (ECCV'18)*. 19–34.

[173] Hanxiao Liu, Karen Simonyan, Oriol Vinyals, Chrisantha Fernando, and Koray Kavukcuoglu. 2017. Hierarchical representations for efficient architecture search. *arXiv preprint arXiv:1711.00436* (2017).

[174] Li Liu, Miao Zhang, Yuqing Lin, and Liangjuan Qin. 2014. A survey on workflow management and scheduling in cloud computing. In *Proceedings of the IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'14)*. IEEE, 837–846.

[175] Manuel Lopez-Martin, Belen Carro, Antonio Sanchez-Esguevillas, and Jaime Lloret. 1967. Conditional variational autoencoder for prediction and feature recovery applied to intrusion detection in iot. *Sensors* 17, 9 (1967).

[176] Qianxi Lu, Tao Peng, Wei Wang, Wenbo Wang, and Chao Hu. 2010. Utility-based resource allocation in uplink of OFDMA-based cognitive radio networks. *Int. J. Commun. Syst.* 23, 2 (2010), 252–274.

[177] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective approaches to attention-based neural machine translation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP'15)* (2015).

[178] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. 2018. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *Proceedings of the European Conference on Computer Vision (ECCV'18).*

[179] David J. C. MacKay and David J. C. MacKay. 2003. *Information Theory, Inference and Learning Algorithms.* Cambridge University Press.

[180] Ritesh Madan, Jaber Borran, Ashwin Sampath, Naga Bhushan, Aamod Khandekar, and Tingfang Ji. 2010. Cell association and interference coordination in heterogeneous LTE-A cellular networks. *IEEE Journal on Selected Areas in Communications* 28, 9 (2010), 1479–1489.

[181] Ritesh Madan, Stephen P. Boyd, and Sanjay Lall. 2010. Fast algorithms for resource allocation in wireless cellular networks. *IEEE/ACM Trans. Netw.* 18, 3 (2010), 973–984.

[182] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. 2016. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks.* ACM.

[183] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. 2017. Neural adaptive video streaming with pensieve. In *Proceedings of the ACM Special Interest Group on Data Communications (SIGCOMM'17).* ACM.

[184] A Marcano-Cedeno et al. 2010. Feature selection using sequential forward selection and classification applying artificial metaplasticity neural network. In *Proceedings of the Annual Conference of the IEEE Industrial Electronics Society (IECON'10).* IEEE.

[185] Michael Mathieu, Mikael Henaff, and Yann LeCun. 2013. Fast training of convolutional networks through ffts. *arXiv preprint arXiv:1312.5851* (2013).

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

[187] Ryan McDonald, Keith Hall, and Gideon Mann. 2010. Distributed training strategies for the structured perceptron. In *Proceedings of the Annual Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT'10).* Association for Computational Linguistics, 456–464.

[188] H. Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, et al. 2016. Communication-efficient learning of deep networks from decentralized data. *arXiv preprint arXiv:1602.05629* (2016).

[189] H. Brendan McMahan, Daniel Ramage, Kunal Talwar, and Li Zhang. 2017. Learning differentially private recurrent language models. *Proceedings of the International Conference on Learning Representations (ICLR'18).*

[190] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, D. B. Tsai, Manish Amde, Sean Owen, et al. 2016. Mllib: Machine learning in apache spark. *J. Mach. Learn. Res.* 17, 1 (2016), 1235–1241.

[191] Andreas Merentitis and Christian Debes. 2015. Automatic fusion and classification using random forests and features extracted with deep learning. In *Proceedings of the IEEE Geoscience and Remote Sensing Society (IGARSS'15).* IEEE, 2943–2946.

[192] E. M. El Mhamdi and R. Guerraoui. 2017. When neurons fail. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'17).* 1028–1037. DOI: https://doi.org/10.1109/IPDPS.2017.66

[193] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V. Le, and Jeff Dean. 2018. A hierarchical model for device placement. In *International Conference on Learning Representations.* https://openreview.net/forum?id=Hkc-TeZ0W.

[194] Azalia Mirhoseini, Hieu Pham, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. 2017. Device placement optimization with reinforcement learning. In *Proceedings of the International Conference of Machine Learning (ICML'17).* JMLR.org, 2430–2439.

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

[196] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous methods for deep reinforcement learning. In *Proceedings of the International Conference on Machine Learning (ICML'16).* 1928–1937.

[197] Mehdi Mohammadi, Ala Al-Fuqaha, Sameh Sorour, and Mohsen Guizani. 2018. Deep learning for IoT big data and streaming analytics: A survey. *IEEE Commun. Surveys Tutor.* 20, 4 (2018).

[198] Philipp Moritz, Robert Nishihara, and Michael Jordan. 2016. A linearly convergent stochastic L-BFGS algorithm. In *Proceedings of the Conference on Artificial Intelligence and Statistics.* 249–258.

[199] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, et al. 2018. Ray: A distributed framework for emerging AI applications. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18).* 561–577.

[200] Md Shirajum Munir, Sarder Fakhrul Abedin, Md Golam Rabiul Alam, Do Hyeon Kim, and Choong Seon Hong. 2017. RNN-based energy demand prediction for smart-home in smart-grid framework. *Journal of Korean Information Science Society (2017)*, 437–439.

[201] Abdulmajid Murad, Frank Alexander Kraemer, Kerstin Bach, and Gavin Taylor. 2019. Autonomous management of energy-harvesting IoT nodes using deep reinforcement learning. *arXiv:1905.04181* (2019).

[202] Preetum Nakkiran, Gal Kaplun, Yamini Bansal, Tristan Yang, Boaz Barak, and Ilya Sutskever. 2019. Deep double descent: Where bigger models and more data hurt. *arXiv preprint arXiv:1912.02292* (2019).

[203] P. Naraei, A. Abhari, and A. Sadeghian. 2016. Application of multilayer perceptron neural networks and support vector machines in classification of healthcare data. In *Proceedings of the Future Technologies Conference (FTC'16).* 848–852.

[204] Blaine Nelson, Benjamin I. P. Rubinstein, Ling Huang, Anthony D. Joseph, Steven J. Lee, Satish Rao, and J. D. Tygar. 2012. Query strategies for evading convex-inducing classifiers. *J. Mach. Learn. Res.* 13, May (2012), 1293–1332.

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

[206] Kazuki Osawa, Yohei Tsuji, Yuichiro Ueno, Akira Naruse, Rio Yokota, and Satoshi Matsuoka. 2018. Second-order optimization method for large mini-batch: Training resnet-50 on imagenet in 35 epochs. *arXiv preprint:1811.12019* (2018).

[207] Xinghao Pan, Maximilian Lam, Stephen Tu, Dimitris Papailiopoulos, Ce Zhang, Michael I. Jordan, Kannan Ramchandran, and Christopher Re. 2016. Cyclades: Conflict-free asynchronous machine learning. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS'16).* 2568–2576.

[208] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z. Berkay Celik, and Ananthram Swami. 2017. Practical black-box attacks against machine learning. In *ASIACCS 2017.* ACM, 506–519.

[209] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z. Berkay Celik, and Ananthram Swami. 2016. The limitations of deep learning in adversarial settings. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P'16).* IEEE.

[210] German I. Parisi, Ronald Kemker, Jose L. Part, Christopher Kanan, and Stefan Wermter. 2019. Continual lifelong learning with neural networks: A review. *Neural Netw.* 113 (2019), 54–71.

[211] Ronald Parr and Stuart J. Russell. 1998. Reinforcement learning with hierarchies of machines. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS'98).*

[212] Zhenghao Peng, Xuyang Chen, Chengwen Xu, Naifeng Jing, Xiaoyao Liang, Cewu Lu, and Li Jiang. 2018. AXNet: ApproXimate computing using an end-to-end trainable neural network. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'18).* ACM, 11.

[213] Alain Petrowski, Gerard Dreyfus, and Claude Girault. 1993. Performance analysis of a pipelined backpropagation parallel algorithm. *IEEE Trans. Neural Netw.* 4, 6 (1993), 970–981.

[214] Hieu Pham, Melody Y. Guan, Barret Zoph, Quoc V. Le, and Jeff Dean. 2018. Efficient neural architecture search via parameter sharing. *arXiv preprint arXiv:1802.03268* (2018).

[215] Joshua Plasse and Niall Adams. 2016. Handling delayed labels in temporally evolving data streams. In *Proceedings of the IEEE International Conference on Big Data (BigData'16).* IEEE, 2416–2424.

[216] Daniel Povey, Xioahui Zhang, and Sanjeev Khudanpur. 2017. Parallel training of DNNs with Natural Gradient and Parameter Averaging. *Proceedings of the International Conference on Learning Representations (ICLR'19) 2015.*

[217] Peter W. Protzel, Daniel L. Palumbo, and Michael K. Arras. 1993. Performance and fault-tolerance of neural networks for optimization. *IEEE Trans. Neural Netw.* 4, 4 (1993), 600–614.

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

[219] Aurick Qiao, Bryon Aragam, Bingjing Zhang, and Eric P. Xing. 2018. Fault tolerance in iterative-convergent machine learning. *arXiv preprint arXiv:1810.07354* (2018).

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

[221] Deirdre Quillen, Eric Jang, Ofir Nachum, Chelsea Finn, Julian Ibarz, and Sergey Levine. 2018. Deep reinforcement learning for vision-based robotic grasping: A simulated comparative evaluation of off-policy methods. In *Proceedings of the International Conference on Robotics and Automation (ICRA'18).*

[222] J. Ross Quinlan. 1986. Induction of decision trees. *Mach. Learn.* 1, 1 (1986), 81–106.

[223] J. Ross Quinlan. 2014. *C4. 5: Programs for Machine Learning.* Elsevier.

[224] Rajat Raina, Anand Madhavan, and Andrew Y. Ng. 2009. Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th Annual International Conference on machime Learning (ICML'09).* ACM.

[225] Nipun Ramakrishnan and Tarun Soni. 2018. Network Traffic Prediction Using Recurrent Neural Networks. In *Proceedings of the 17th IEEE International Conference on Machine Learning and Applications (ICMLA'18).* IEEE.

[226] Brian Randell. 1975. System structure for software fault tolerance. *IEEE Trans. Softw. Eng.* 2 (1975), 220–232.

[227] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. Xnor-net: Imagenet classification using binary convolutional neural networks. In *Proceedings of the European Conference on Computer Vision.* Springer.

[228] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V. Le. 2019. Regularized evolution for image classifier architecture search. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

[229] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. 2011. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS'11)*. 693–701.

[230] Liangliang Ren, Xin Yuan, Jiwen Lu, Ming Yang, and Jie Zhou. 2018. Deep reinforcement learning with iterative shift for visual tracking. In *Proceedings of the European Conference on Computer Vision (ECCV'18)*. 684–700.

[231] Konrad Rieck, Philipp Trinius, Carsten Willems, and Thorsten Holz. 2011. Automatic analysis of malware behavior using machine learning. *J. Comput. Secur.* 19, 4 (2011), 639–668.

[232] Henriette Röger and Ruben Mayer. 2019. A comprehensive survey on parallelization and elasticity in stream processing. *ACM Comput. Surveys* 52, 2 (2019), 36.

[233] Rodrigo Roman, Jianying Zhou, and Javier Lopez. 2013. On the features and challenges of security and privacy in distributed internet of things. *Comput. Netw.* 57, 10 (2013), 2266–2279.

[234] Adriana Romero, Nicolas Ballas, Samira Ebrahimi Kahou, Antoine Chassang, Carlo Gatta, and Yoshua Bengio. 2014. Fitnets: Hints for thin deep nets. *arXiv preprint arXiv:1412.6550* (2014).

[235] Frederik Ruelens, Bert J. Claessens, Stijn Vandael, Bart De Schutter, Robert Babuška, and Ronnie Belmans. 2016. Residential demand response of thermostatically controlled loads using batch reinforcement learning. *IEEE Trans. Smart Grid* 8, 5 (2016), 2149–2159.

[236] Vahideh Saeidi, Biswajeet Pradhan, Mohammed O. Idrees, and Zulkiflee Abd Latif. 2014. Fusion of airborne lidar with multispectral spot 5 image for enhancement of feature extraction using Dempster–Shafer theory. *IEEE Transactions on Geoscience and Remote Sensing* 52, 10 (2014), 6017–6025.

[237] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'18)*.

[238] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).

[239] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 2014. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In *Proceedings of the Annual Conference of the International Speech Communication (INTERSPEECH'14)*.

[240] Alexander Sergeev and Mike Del Balso. 2018. Horovod: Fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799* (2018).

[241] Burr Settles. 2012. Active Learning. *Synth. Lect. Artific. Intell. Mach. Learn.* 6, 1 (2012), 1–114.

[242] Ali Shafahi, W. Ronny Huang, Mahyar Najibi, Octavian Suciu, Christoph Studer, Tudor Dumitras, and Tom Goldstein. 2018. Poison frogs! Targeted clean-label poisoning attacks on neural networks. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS'18)*.

[243] Wenling Shang, Kihyuk Sohn, Diogo Almeida, and Honglak Lee. 2016. Understanding and improving convolutional neural networks via concatenated rectified linear units. In *Proceedings of the International Conference on Machine Learning (ICML'16)*. 2217–2225.

[244] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake Hechtman. 2018. Mesh-TensorFlow: Deep learning for supercomputers. In *Proceedings of the International Conference on Neural Information Processing Systems.*

[245] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. 2016. Edge computing: Vision and challenges. *IEEE Internet Things J.* 3, 5 (2016), 637–646.

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

[247] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. 2017. Membership inference attacks against machine learning models. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'17)*. IEEE, 3–18.

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

[249] Sabrina Sicari, Alessandra Rizzardi, Luigi Alfredo Grieco, and Alberto Coen-Porisini. 2015. Security, privacy and trust in Internet of Things: The road ahead. *Comput. Netw.* 76 (2015), 146–164.

[250] Laurent Sifre and Stéphane Mallat. 2014. Rigid-motion scattering for image classification. Ph. D. Dissertation. Ecole Normale Superieure, Paris, France.

[251] Amit Kumar Sikder, Hidayet Aksu, and A. Selcuk Uluagac. 2017. 6thsense: A context-aware sensor-based attack detector for smart devices. In *Proceedings of the 26th USENIX Security Symposium*. 397–414.

[252] Kiran Jot Singh and Divneet Singh Kapoor. 2017. Create your own Internet of Things: A survey of IoT platforms. *IEEE Consum. Electron. Mag.* 6, 2 (2017), 57–68.

[253] Sukhpal Singh and Inderveer Chana. 2016. A survey on resource scheduling in cloud computing: Issues and challenges. *J. Grid Comput.* 14, 2 (2016), 217–264.

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

[255] Samuel L. Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V. Le. 2017. Don't decay the learning rate, increase the batch size. *arXiv preprint arXiv:1711.00489* (2017).

[256] Evan R. Sparks, Ameet Talwalkar, Daniel Haas, Michael J. Franklin, Michael I. Jordan, and Tim Kraska. 2015. Automating model search for large scale machine learning. In *Proceedings of the 6th ACM Symposium on Cloud Computing*. ACM.

[257] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.* 15, 1 (2014), 1929–1958.

[258] Jacob Steinhardt, Pang Wei W. Koh, and Percy S. Liang. 2017. Certified defenses for data poisoning attacks. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS' (2017).* 3517–3529.

[259] Sebastian Urban Stich. 2019. Local SGD converges fast and communicates little. In *Proceedings of the International Conference on Learning Representations (ICLR'19).*

[260] Ion Stoica, Dawn Song, Raluca Ada Popa, David Patterson, Michael W. Mahoney, Randy Katz, Anthony D. Joseph, Michael Jordan, Joseph M. Hellerstein, Joseph E. Gonzalez, et al. 2017. A Berkeley view of systems challenges for ai. *arXiv preprint arXiv:1712.05855* (2017).

[261] Nikko Strom. 2015. Scalable distributed DNN training using commodity GPU cloud computing. In *Proceedings of the Annual Conference of the International Speech Communication (INTERSPEECH'15).*

[262] Octavian Suciu, Radu Marginean, Yigitcan Kaya, Hal Daume III, and Tudor Dumitras. 2018. When does machine learning FAIL? Generalized transferability for evasion and poisoning attacks. In *Proceedings of the USENIX Security Symposium (USENIXSecurity'18).* 1299–1316.

[263] Salmin Sultana, Daniele Midi, and Elisa Bertino. 2014. Kinesis: A security incident response and prevention system for wireless sensor networks. In *Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SenSys'14).* ACM.

[264] James Supancic III and Deva Ramanan. 2017. Tracking as online decision-making: Learning a policy from streaming videos with reinforcement learning. In *Proceedings of the International Conference on Computer Vision (ICCV'17).* 322–331.

[265] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction.* MIT Press.

[266] Chuanqi Tan, Fuchun Sun, Tao Kong, Wenchang Zhang, Chao Yang, and Chunfang Liu. 2018. A survey on deep transfer learning. In *Proceedings of the International Conference on Artificial Neural Networks.* Springer, 270–279.

[267] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V. Le. 2019. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'19).*

[268] Hao Tang, Wei Wang, Songsong Wu, Xinya Chen, Dan Xu, Nicu Sebe, and Yan Yan. 2019. Expression conditional gan for facial expression-to-expression translation. In *Proceedings of the International Conference on Image Processing (ICIP'19).* IEEE, 4449–4453.

[269] TensorFlow. 2019. High performance inference with TensorRT Integration. Retrieved from https://medium.com/tensorflow/high-performance-inference-with-tensorrt-integration-c4d78795fbfe.

[270] Sebastian Thrun and Anton Schwartz. 1995. Finding structure in reinforcement learning. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS'95).*

[271] James T. Townsend. 1971. Theoretical analysis of an alphabetic confusion matrix. *Percept. Psychophys.* 9, 1 (1971), 40–50.

[272] Florian Tramer, Fan Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2016. Stealing machine learning models via prediction apis. In *Proceedings of the 25th USENIX Security Symposium (USENIXSecurity'16).* 601–618.

[273] Amin Ullah, Jamil Ahmad, Khan Muhammad, Muhammad Sajjad, and Sung Wook Baik. 2018. Action recognition in video sequences using deep bi-directional LSTM With CNN features. *IEEE Access* 6 (2018), 1155–1166.

[274] Michel Vacher, Benjamin Lecouteux, Javier Serrano Romero, Moez Ajili, Francois Portet, and Solange Rossato. 2015. Speech and speaker recognition for home automation: Preliminary results. In *Proceedings of the International Conference on Speech Technology and Human-Computer Dialogue (SpeD'15).* IEEE.

[275] Leslie G. Valiant. 1984. A theory of the learnable. In *Proceedings of the 16th Annual ACM Symposium on Theory of Computing.* ACM, 436–445.

[276] Vincent Vanhoucke, Andrew Senior, and Mark Z. Mao. 2011. Improving the speed of neural networks on CPUs. In *Proceedings of the Deep Learning and Unsupervised Feature Learning Workshop (NIPS'11).*

[277] Joaquin Vanschoren. 2018. Meta-learning: A survey. *arXiv preprint arXiv:1810.03548* (2018).

[278] Subhashini Venugopalan, Huijuan Xu, Jeff Donahue, Marcus Rohrbach, Raymond Mooney, and Kate Saenko. 2014. Translating videos to natural language using deep recurrent neural networks. *arXiv preprint arXiv:1412.4729* (2014).

[279] Riccardo Volpi, Hongseok Namkoong, Ozan Sener, John C. Duchi, Vittorio Murino, and Silvio Savarese. 2018. Generalizing to unseen domains via adversarial data augmentation. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS'18)*. 5334–5344.

[280] Gang Wang, Jinxing Hao, Jian Ma, and Lihua Huang. 2010. A new approach to intrusion detection using artificial neural networks and fuzzy clustering. *Expert Syst. Appl.* 37, 9 (2010), 6225–6232.

[281] Jason Wang and Luis Perez. 2017. The effectiveness of data augmentation in image classification using deep learning. *arXiv preprint arXiv:1712.04621*.

[282] Meng Wang and Xian-Sheng Hua. 2011. Active learning in multimedia annotation and retrieval: A survey. *ACM Trans. Intell. Syst. Technol.* 2, 2 (2011), 1–21.

[283] Meng Wang, Xian Sheng Hua, Tao Mei, Jinhui Tang, Guo Jun Qi, Yan Song, and Li Rong Dai. 2007. Interactive video annotation by multi concept multi modality active learning. In *Proceedings of the IEEE International Conference on Semantic Computing (ICSC'07)*.

[284] Minjie Wang, Chien-chin Huang, and Jinyang Li. 2019. Supporting very large models using automatic dataflow graph partitioning. In *Proceedings of the European Conference on Computer Systems (EuroSys'19)*. 1–17.

[285] Shiqiang Wang, Tiffany Tuor, Theodoros Salonidis, Kin K. Leung, Christian Makaya, Ting He, and Kevin Chan. 2018. When edge meets learning: Adaptive control for resource-constrained distributed machine learning. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM'18)*. IEEE, 63–71.

[286] Denis Weerasiri, Moshe Chai Barukh, Boualem Benatallah, Quan Z. Sheng, and Rajiv Ranjan. 2017. A taxonomy and survey of cloud resource orchestration techniques. *ACM Comput. Surveys* 50, 2 (2017), 26.

[287] Jinliang Wei, Wei Dai, Aurick Qiao, Qirong Ho, Henggang Cui, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. 2015. Managed communication and consistency for fast data-parallel iterative analytics. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC'15)*. ACM, 381–394.

[288] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2017. Terngrad: Ternary gradients to reduce communication in distributed deep learning. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS'17)*. 1509–1519.

[289] Zhenyu Wen, Pramod Bhatotia, Ruichuan Chen, Myungjin Lee, et al. 2018. Approxiot: Approximate analytics for edge computing. In *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS'18)*. IEEE.

[290] Zheng Wen, Daniel O'Neill, and Hamid Maei. 2015. Optimal demand response using device-based reinforcement learning. *IEEE Trans. Smart Grid* 6, 5 (2015), 2312–2324.

[291] Zhenyu Wen, Renyu Yang, Peter Garraghan, Tao Lin, Jie Xu, and Michael Rovatsos. 2017. Fog orchestration for internet of things services. *IEEE Internet Comput.* 21, 2 (2017), 16–24.

[292] Shuang Wu, Guoqi Li, Feng Chen, and Luping Shi. 2018. Training and inference with integers in deep neural networks. In *International Conference on Learning Representations*.

[293] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint:1609.08144* (2016).

[294] Han Xiao, Huang Xiao, and Claudia Eckert. 2012. Adversarial label flips attack on support vector machines. In *Proceedings of the European Conference on Artificial Intelligence (ECAI'12)*. 870–875.

[295] Eric P. Xing, Qirong Ho, Wei Dai, Jin Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. 2015. Petuum: A new platform for distributed machine learning on big data. *IEEE Trans. Big Data* 1, 2 (2015), 49–67.

[296] Caiming Xiong, Victor Zhong, and Richard Socher. 2017. Dcn+: Mixed objective and deep residual coattention for question answering. *arXiv preprint arXiv:1711.00106* (2017).

[297] Xiaobin Xu, Jin Zheng, Jian-bo Yang, Dong-ling Xu, and Yu-wang Chen. 2017. Data classification using evidence reasoning rule. *Knowl.-Based Syst.* 116 (2017), 144–151.

[298] Zhiyuan Xu, Yanzhi Wang, Jian Tang, Jing Wang, and Mustafa Cenk Gursoy. 2017. A deep reinforcement learning-based framework for power-efficient resource allocation in cloud RANs. In *Proceedings of the IEEE International Conference on Communications (ICC'17)*. IEEE.

[299] Neeraja J. Yadwadkar, Bharath Hariharan, Joseph E. Gonzalez, and Randy Katz. 2016. Multi-task learning for straggler avoiding predictive job scheduling. *J. Mach. Learn. Res.* 17, 1 (2016), 3692–3728.

[300] Chaoqun Yang, Li Feng, Heng Zhang, Shibo He, and Zhiguo Shi. 2018. A novel data fusion algorithm to combat false data injection attacks in networked radar systems. *IEEE Trans. Signal Info. Process. Netw.* 4, 1 (2018), 125–136.

[301] Emre Yigitoglu, Mohamed Mohamed, Ling Liu, and Heiko Ludwig. 2017. Foggy: A framework for continuous automated IoT application deployment in fog computing. In *Proceedings of the Administrative Information Management Services (AIMS'17)*. IEEE, 38–45.

[302] Yang You, Igor Gitman, and Boris Ginsburg. 2017. Large batch training of convolutional networks. *arXiv preprint arXiv:1708.03888* (2017).

[303] Hao Yu, Sen Yang, and Shenghuo Zhu. 2018. Parallel restarted SGD for non-convex optimization with faster convergence and less communication. *arXiv preprint:1807.06629* 2, 4 (2018), 7.

[304] Hao Yu, Sen Yang, and Shenghuo Zhu. 2019. Parallel restarted SGD with faster convergence and less communication: Demystifying why model averaging works for deep learning. In *Proceedings of the AAAI Conference on Artificial Intelligence.*

[305] Sangdoo Yun, Jongwon Choi, Youngjoon Yoo, Kimin Yun, and Jin Young Choi. 2017. Action-decision networks for visual tracking with deep reinforcement learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'17).* 2711–2720.

[306] Sergey Zagoruyko et al. 2016. Paying more attention to attention: Improving performance of convolutional neural networks via attention transfer. *arXiv:1612.03928* (2016).

[307] Shuangfei Zhai, Yu Cheng, Zhongfei Mark Zhang, and Weining Lu. 2016. Doubly convolutional neural networks. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS' (2016).* 1082–1090.

[308] Zhi-Hui Zhan, Xiao-Fang Liu, Yue-Jiao Gong, Jun Zhang, Henry Shu-Hung Chung, and Yun Li. 2015. Cloud computing resource scheduling and a survey of its evolutionary approaches. *ACM Comput. Surveys* 47, 4 (2015), 63.

[309] Ce Zhang, Arun Kumar, and Christopher Ré. 2016. Materialization optimizations for feature selection workloads. *ACM Trans. Database Syst.* 41, 1 (2016), 2.

[310] Chaoyun Zhang, Paul Patras, and Hamed Haddadi. 2019. Deep learning in mobile and wireless networking: A survey. *IEEE Commun. Surveys Tutor.* (2019).

[311] Hantian Zhang, Jerry Li, Kaan Kara, Dan Alistarh, Ji Liu, and Ce Zhang. 2017. Zipml: Training linear models with end-to-end low precision, and a little bit of deep learning. In *Proceedings of the International Conference on Machine Learning (ICML'17).* JMLR.org, 4035–4043.

[312] Jian Zhang, Christopher De Sa, Ioannis Mitliagkas, and Christopher Ré. 2016. Parallel SGD: When does averaging help? *arXiv preprint arXiv:1606.07365* (2016).

[313] Quan-shi Zhang and Song-Chun Zhu. 2018. Visual interpretability for deep learning: A survey. *Front. Info. Technol. Electron. Eng.* 19, 1 (2018), 27–39.

[314] Sixin Zhang, Anna E Choromanska, and Yann LeCun. 2015. Deep learning with elastic averaging SGD. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS'15).* 685–693.

[315] Weishan Zhang, Wuwu Guo, Xin Liu, Yan Liu, Jiehan Zhou, Bo Li, Qinghua Lu, and Su Yang. 2018. LSTM-based analysis of industrial IoT equipment. *IEEE Access* 6 (2018), 23551–23560.

[316] Wen-An Zhang, Bo Chen, and Michael Z. Q. Chen. 2016. Hierarchical fusion estimation for clustered asynchronous sensor networks. *IEEE Trans. Automat. Control* 61, 10 (2016), 3064–3069.

[317] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. 2018. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'18).* 6848–6856.

[318] Caidan Zhao, Caiyun Chen, Zeping He, and Zhiqiang Wu. 2018. Application of auxiliary classifier wasserstein generative adversarial networks in wireless signal classification of illegal unmanned aerial vehicles. *Appl. Sci.* 8, 12 (2018), 2664.

[319] Caidan Zhao, Mingxian Shi, Zhibiao Cai, and Caiyun Chen. 2018. Research on the open-categorical classification of the Internet-of-things based on generative adversarial networks. *Appl. Sci.* 8, 12 (2018), 2351.

[320] Dongbin Zhao, Yaran Chen, and Le Lv. 2016. Deep reinforcement learning with visual attention for vehicle classification. *IEEE Trans. Cogn. Dev. Syst.* 9, 4 (2016), 356–367.

[321] Binbin Zhou, Jiannong Cao, Xiaoqin Zeng, and Hejun Wu. 2010. Adaptive traffic light control in wireless sensor network-based intelligent transportation system. In *Proceedings of the IEEE Vehicular Technology Conference (VTC'10).* IEEE, 1–5.

[322] Shusen Zhou, Qingcai Chen, and Xiaolong Wang. 2013. Active deep learning method for semi-supervised sentiment classification. *Neurocomputing* 120 (2013), 536–546.

[323] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. 2016. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160* (2016).

[324] Yanqi Zhou, Siavash Ebrahimi, Sercan Ö. Arık, Haonan Yu, Hairong Liu, and Greg Diamos. 2018. Resource-efficient neural architect. *arXiv preprint arXiv:1806.07912* (2018).

[325] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex J. Smola. 2010. Parallelized stochastic gradient descent. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS'10).* 2595–2603.

Received October 2019; revised March 2020; accepted May 2020