Distributed Machine Learning in Coalition Environments: Overview of Techniques

Tiffany Tuor*, Shiqiang Wang†, Kin K. Leung* and Kevin Chan‡
*Imperial College London, UK. Email: {tiffany.tuor14, kin.leung}@imperial.ac.uk
†IBM T. J. Watson Research Center, Yorktown Heights, NY, USA. Email: wangshiq@us.ibm.com
‡Army Research Laboratory, Adelphi, MD, USA. Email: kevin.s.chan.civ@mail.mil

***Abstract*—Many modern applications generate a significant amount of data in dispersed geographical areas. To analyze and make use of the data, data fusion and machine learning techniques are usually applied, which has the potential to greatly enhance the amount of information extracted from the data. These algorithms traditionally run in data center environments where all the data are available at a central location. It is challenging to run them in distributed coalition environments, where it is impractical to send all the raw data to a single place due to bandwidth and security constraints. This problem has gained notable attention recently. In this paper, we provide an overview of available techniques and recent results of performing data fusion and machine learning in a distributed coalition environment, without sharing the raw data among local processing nodes. We discuss techniques for distributed model training, scoring, and outline some applications where these techniques are applicable and beneficial.**

I. Introduction

Future military applications will largely benefit from agile analytics in tactical operating environments. Machine learning has been proposed for a myriad of applications. There is strong potential of machine learning techniques for use in military coalition environments to gain situation understanding through sensing in the operational environment, learning optimal configurations of networks and systems, characterizing and predicting adversarial entities [1]. Further, these techniques can be used to provide robustness against adversarial actions attempting to disrupt operations and even learning processes in military environments.

Military operational environments may provide challenges to the learning process, specifically due to resource constraints and strict requirements. In addition to the adversarial presence, these approaches will have to address various complexities of the data, including attempts to poison training data. The data may be sparse, which is distinct from other situations that have sufficient training data. Machine learning-enabled systems may have to operate in cases of limitations in the availability of data. Also, the data may be heterogeneous, as the data may come from a variety of different sources (e.g., images, video, text), requiring analysts to fuse the data in intelligent ways. Further, the sources of information may not produce the same amount of data, so distributed processors will likely learn at different rates.

In this paper, we consider a system shown in Fig. 1, where multiple coalition members have sensors carried by human or vehicles deployed in the field. These sensors may collect and store data. Analytics code may be located in either the same or different physical entities to process the data using machine learning techniques. We present an overview of techniques of performing machine learning in a distributed manner that does *not* require all local data to be transmitted to a central location. Such distributed machine learning techniques have the benefits of reducing the communication bandwidth consumption and reducing the risk of unwillingly disclosing sensitive information to other entities. Distributed machine learning is also able to make use of the computation and storage capability of multiple nodes, which can improve the performance compared to the centralized setting in resource-scarce environments [2], such as in emerging systems of mobile edge computing [3], [4] and Internet of Things [5], [6].

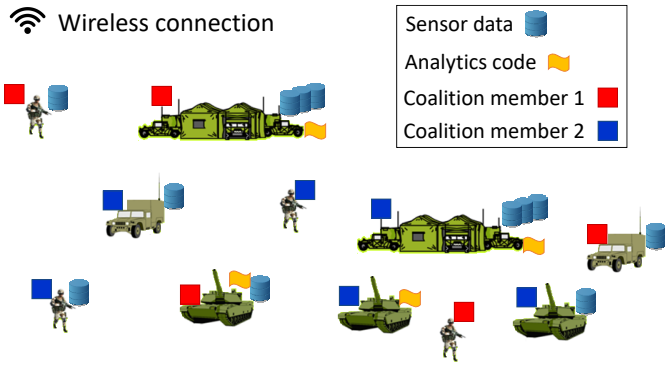

Fig. 1. System architecture.

Machine learning applications generally include two phases. The first phase is to train a model from training data, so that the model captures essential characteristics of the data. The second phase is to apply new data (often collected in real time) on the trained model, to make predictions from the data. This phase is also called scoring. We separately discuss these two phases in Sections II and III. Then, in Section IV, we discuss reinforcement learning with multiple agents, which can be seen as an extension of distributed machine learning to "real" artificial intelligence applications. Finally, in Section V, we outline some application scenarios where distributed machine learning and multi-agent reinforcement learning can be useful.

II. Distributed Model Training

We first explain the training of generic machine learning models, then summarize the basic procedure of distributed training.

A. Loss Functions

Machine learning models are trained using training data. For each data sample x with target output y, we define the loss function as $l(w, x, y)$, where w is the weight (parameter) of the model and w, x, and y can all be vectors.

We note that machine learning can be classified into supervised and unsupervised applications. In supervised machine learning applications, the goal is to make predictions from the input data. Here, the training data specifies the model input as well as the target model output, such as the class label of an image. The goal of model training is to adjust the model parameters so that the input data samples are mapped to their desired output values. After training, the model can be used to estimate the output value based on input data. Unsupervised machine learning does not have a desired output value in its training dataset. Here, the models are often trained to provide some form of a summary of the training data, and they can be used for applications such as outlier detection and source separation. The target output is only defined for supervised models; for unsupervised models, the value of y is undefined and can be ignored.

The loss function captures how well the model with weight w fits the training data sample (x, y). For example [7], the loss function of a soft support vector machine (Soft-SVM) can be written as

$$l(w, x, y) = \lambda \|w\|^2 + \max\{0, 1 - y(w \cdot x)\}. \tag{1}$$

For loss function, the logistic regression is

$$l(w, x, y) = \log(1 + \exp(-y(w \cdot x))). \tag{2}$$

Both Soft-SVM and logistic regression with loss functions defined above are binary classifiers, where $y \in \{+1, -1\}$ is a scalar variable representing the target output label of the classifier, and $w \cdot x$ denotes the dot product of vectors w and x. These and other binary classifiers can be extended to multi-class classifiers using one-versus-one or one-versus-all voting mechanisms. The loss functions of unsupervised models do not include the target output label y. For example, the loss function of K-means can be defined as

$$l(w, x) = \min_k \|x - w_{(k)}\|^2 \tag{3}$$

where the weight vector $w = [w_{(1)}, w_{(2)}, ..., w_{(k)}, ..., w_{(K)}]^{\mathrm{T}}$ and $w_{(k)}$ is the k-th center of the K-means model.

We define the training dataset as set $\mathcal{D}$, where each element in $\mathcal{D}$ is a tuple of (x, y) representing the data sample. The goal of machine learning is to find the optimal model weight w^* that minimizes the sum of losses of all training data samples. Formally, we define the overall (global) loss function for model weight w as

$$L(w) = \frac{1}{|\mathcal{D}|} \sum_{(x,y)\in\mathcal{D}} l(w, x, y) \tag{4}$$

where $|\mathcal{D}|$ denotes the cardinality of the set $\mathcal{D}$, and we would like to find

$$w^* = \arg\min_w L(w). \tag{5}$$

B. Gradient Descent

1) Deterministic Gradient Descent: The solution to (5) can be found using gradient descent on the model weight w. At the t-th step, the value of the model weight $w(t)$ can be found from

$$w(t) = w(t-1) - \eta \nabla L(w(t-1)) \tag{6}$$

where η is the step size of gradient descent. At initialization, the model is initialized with weight equal to $w(0)$.

When $L(w)$ conforms to some convexity properties, it can be shown that the gradient descent procedure in (6) converges to the true optimal weight w^* after a sufficient number of iterations [8].

2) Stochastic Gradient Descent: In practice, the training dataset $\mathcal{D}$ is often very large. Hence, it is usually very time-consuming to evaluate the loss function and its gradient on the entire training data set as in (4) and (6). To resolve this problem, stochastic gradient descent is often used for training on large datasets [9].

In stochastic gradient descent, the gradient is computed on the loss function defined on a random subset of $\mathcal{D}$. We denote this random subset as $\widetilde{\mathcal{D}}$, which includes data samples that are randomly and uniformly sampled from the original dataset $\mathcal{D}$. In every new stochastic gradient descent step, a new $\widetilde{\mathcal{D}}$ is obtained. The size of $\widetilde{\mathcal{D}}$, i.e., $\left|\widetilde{\mathcal{D}}\right|$, is a fixed constant, which is referred to as the mini-batch size of stochastic gradient descent.

We can then define the loss function on the random subset $\widetilde{\mathcal{D}}$ as

$$\widetilde{L}(w) = \frac{1}{\left|\widetilde{\mathcal{D}}\right|} \sum_{(x,y)\in\widetilde{\mathcal{D}}} l(w, x, y) \tag{7}$$

where $\widetilde{L}(w)$ is a random function of w.

Theorem 1. *When the samples in $\widetilde{\mathcal{D}}$ are uniformly drawn (without replacement) from $\mathcal{D}$, we have*

$$\mathrm{E}\left(\widetilde{L}(w)\right) = L(w) \tag{8}$$

$$\mathrm{E}\left(\nabla\widetilde{L}(w)\right) = \nabla L(w) \tag{9}$$

where $\mathrm{E}(\cdot)$ denotes the expectation.

Proof. It is easy to see that for any $(x, y) \in \mathcal{D}$,

$$\Pr\left\{(x, y) \in \widetilde{\mathcal{D}}\right\} = \frac{\left|\widetilde{\mathcal{D}}\right|}{|\mathcal{D}|}.$$

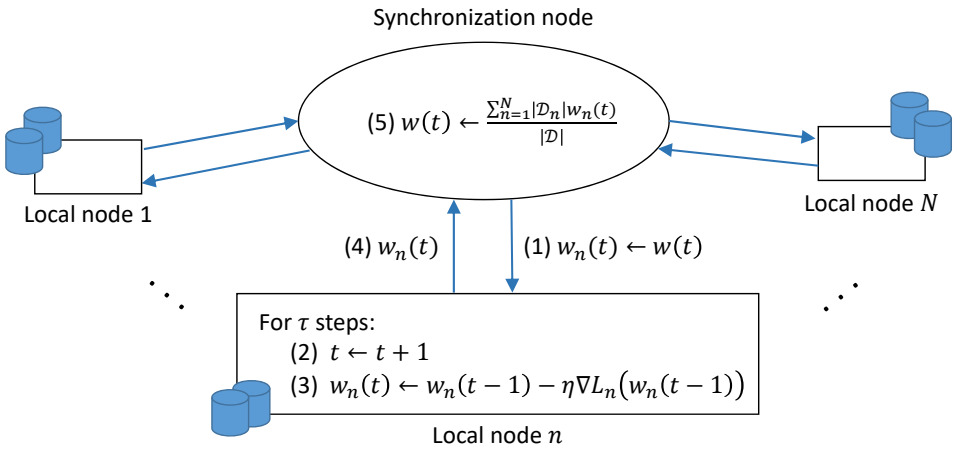

Fig. 2. Distributed deterministic gradient descent.

We have

$$\begin{aligned}
&\mathrm{E}\left(\widetilde{L}(w)\right) \\
&= \mathrm{E}\left(\frac{1}{|\widetilde{\mathcal{D}}|}\sum_{(x,y)\in\widetilde{\mathcal{D}}} l(w,x,y)\right) \\
&= \frac{1}{|\widetilde{\mathcal{D}}|}\sum_{(x,y)\in\mathcal{D}} \Pr\left\{(x,y)\in\widetilde{\mathcal{D}}\right\}\cdot l(w,x,y) \\
&= \frac{1}{|\mathcal{D}|}\sum_{(x,y)\in\mathcal{D}} l(w,x,y) \\
&= L(w)
\end{aligned}$$

and

$$\mathrm{E}\left(\nabla\widetilde{L}(w)\right) = \nabla\mathrm{E}\left(\widetilde{L}(w)\right) = \nabla L(w)$$

due to the linearity of the expectation and gradient operators. □

The stochastic gradient descent iteration is given by:

$$w(t) = w(t-1) - \eta\nabla\widetilde{L}(w(t-1)) \tag{10}$$

where a new sample of $\widetilde{L}(w(t-1))$ is obtained for every new step t. From Theorem 1, it is easy to see that when $w(t-1)$ is given, the conditional expectation of $w(t)$ is

$$\mathrm{E}\left(w(t)|w(t-1)\right) = w(t-1) - \eta\nabla L(w(t-1)) \tag{11}$$

where the right hand side is the same as (6). The stochastic gradient descent mechanism has been shown to converge under certain conditions [10].

C. Distributed Gradient Descent

In the distributed setting, assume there are N different nodes. Each node n has a local training dataset $\mathcal{D}_n$. The model training procedure is based on gradient descent and can be performed in a distributed manner *without* sending the training data to a central location. As shown in Fig. 2, distributed gradient descent includes individual gradient descent steps at local nodes, and the synchronization of model weights (after one or multiple local iterations) among all local nodes through a synchronization node. We define $\mathcal{D} = \cup_{n=1}^{N}\mathcal{D}_n$ as the union of all local datasets, and assume that $\mathcal{D}_n \cap \mathcal{D}_{n'} = \emptyset$ for $n \neq n'$. The data distribution can be arbitrary, and we do not assume any specific distribution of the data. We also define

$$L_n(w_n) = \frac{1}{|\mathcal{D}_n|}\sum_{(x,y)\in\mathcal{D}_n} l(w_n,x,y) \tag{12}$$

as the local loss function of node n defined on its local dataset $\mathcal{D}_n$, where w_n is the local model weight. The distributed gradient descent mechanism includes a local computation phase and a synchronization phase.

1) Local Computation: The local computation phase at node n performs gradient descent on the loss function defined on the local dataset $\mathcal{D}_n$, and updates the local model weight w_n:

$$w_n(t) = w_n(t-1) - \eta\nabla L_n(w_n(t-1)). \tag{13}$$

The expression in (13) is for distributed deterministic gradient descent.

For distributed stochastic gradient descent, we define $\widetilde{\mathcal{D}} = \cup_{n=1}^{N}\widetilde{\mathcal{D}}_n$, where $\widetilde{\mathcal{D}}_n$ is a sampled subset of the local dataset $\mathcal{D}_n$. The stochastic gradient descent iteration can be obtained by replacing $L_n(\cdot)$ in (13) with $\widetilde{L}_n(\cdot)$, which is the local loss function defined on a sampled dataset $\widetilde{\mathcal{D}}_n$, yielding

$$w_n(t) = w_n(t-1) - \eta\nabla\widetilde{L}_n(w_n(t-1)). \tag{14}$$

The local gradient descent in (13) or (14) is performed for τ steps, where $\tau \geq 1$ is an integer.

2) Synchronization: In the synchronization phase, a weighted average of the local model weights at all nodes is computed, through the assistance of the synchronization node (see Fig. 2). Then, the local weights are updated with the weighted averaged (global) weight. The weighted average is computed according to

$$w(t) = \frac{\sum_{n=1}^{N}|\mathcal{D}_n|\, w_n(t)}{|\mathcal{D}|} \tag{15}$$

where $|\mathcal{D}| = \sum_{n=1}^{N} |\mathcal{D}_n|$ as defined above. Then, the value of $w(t)$ is assigned back to each node so that the local weights $\{w_n(t) : \forall n\}$ are updated. The new value of $w_n(t)$ is then used for the next local iteration.

With the above convention, we assume that synchronization is performed *at the end of step* t, if there have been τ local iterations since the last synchronization. When the number of local iterations τ is larger than one, the weights are *not* synchronized in all steps. See Fig. 2 for an illustration of the complete procedure.

3) Rationale of This Approach: The reason behind the above procedure is that, when $\tau = 1$, i.e., when synchronization is performed after every local iteration, distributed gradient descent provides the same mathematical progression as centralized gradient descent. This is shown in the below theorem and corollary.

Theorem 2. *When $w_n(t-1) = w(t-1)$ for all n, after performing local deterministic gradient descent according to (13) and computing $w(t)$ according to (15), we have (6).*

Proof.

$$
\begin{aligned}
w(t) &= \frac{\sum_{n=1}^{N} |\mathcal{D}_n| \, w_n(t)}{|\mathcal{D}|} && \text{(from (15))} \\
&= \frac{\sum_{n=1}^{N} |\mathcal{D}_n| \left(w_n(t-1) - \eta \nabla L_n(w_n(t-1))\right)}{|\mathcal{D}|} && \text{(from (13))} \\
&= \frac{\sum_{n=1}^{N} |\mathcal{D}_n| \left(w(t-1) - \eta \nabla L_n(w(t-1))\right)}{|\mathcal{D}|} && \text{(assumption that } w_n(t-1) = w(t-1)\text{)} \\
&= w(t-1) - \eta \nabla \left(\frac{\sum_{n=1}^{N} |\mathcal{D}_n| \, L_n(w(t-1))}{|\mathcal{D}|} \right) && \text{(linearity of gradient)} \\
&= w(t-1) - \eta \nabla \left(\frac{\sum_{n=1}^{N} \sum_{(x,y)\in\mathcal{D}_n} l(w(t-1), x, y)}{|\mathcal{D}|} \right) && \text{(definition of } L_n(\cdot) \text{ in (12))} \\
&= w(t-1) - \eta \nabla \left(\frac{\sum_{(x,y)\in\mathcal{D}} l(w(t-1), x, y)}{|\mathcal{D}|} \right) && (\mathcal{D} = \cup_{n=1}^{N} \mathcal{D}_n \text{ and } \mathcal{D}_n \cap \mathcal{D}_{n'} = \emptyset \text{ for } n \neq n') \\
&= w(t-1) - \eta \nabla L(w(t-1)) && \text{(definition of } L(\cdot) \text{ in (4))}
\end{aligned}
$$

□

For distributed stochastic gradient descent, we enforce that $\frac{|\widetilde{\mathcal{D}}_n|}{|\mathcal{D}_n|} = \frac{|\widetilde{\mathcal{D}}|}{|\mathcal{D}|}$ for all n. This means that the percentages of random samples at all nodes must be the same. Then, using a similar proof as above, we can get the following result.

Corollary 1. *When $w_n(t-1) = w(t-1)$ and $\frac{|\widetilde{\mathcal{D}}_n|}{|\mathcal{D}_n|} = \frac{|\widetilde{\mathcal{D}}|}{|\mathcal{D}|}$ for all n, after performing local stochastic gradient descent according to (14) and computing $w(t)$ according to (15), we have (10).*

A main difference between distributed and centralized stochastic gradient descent is that, in distributed stochastic gradient descent, the data samples in $\widetilde{\mathcal{D}}_n$ is drawn at each node independently, from its local dataset $\mathcal{D}_n$. Using a similar proof as that for Theorem 1, we can show the unbiasedness of the local loss $\widetilde{L}_n(w)$ and its gradient $\nabla \widetilde{L}_n(w)$ at each node n. For the global loss $\widetilde{L}(w)$ and its gradient $\nabla \widetilde{L}(w)$, we also have the unbiasedness result shown below.

Corollary 2. *When the samples in $\widetilde{\mathcal{D}}_n$ are uniformly drawn (without replacement) from $\mathcal{D}_n$ at each node n, then (8) and (9) hold if $\frac{|\widetilde{\mathcal{D}}_n|}{|\mathcal{D}_n|} = \frac{|\widetilde{\mathcal{D}}|}{|\mathcal{D}|}$ for all n.*

Proof. We know that for any $(x, y) \in \mathcal{D}_n$,

$$\Pr\left\{(x,y) \in \widetilde{\mathcal{D}}_n\right\} = \frac{|\widetilde{\mathcal{D}}_n|}{|\mathcal{D}_n|} = \frac{|\widetilde{\mathcal{D}}|}{|\mathcal{D}|}.$$

We have

$$
\begin{aligned}
&\mathrm{E}\left(\widetilde{L}(w)\right) \\
&= \mathrm{E}\left(\frac{1}{|\widetilde{\mathcal{D}}|} \sum_{(x,y)\in\cup_{n=1}^{N}\widetilde{\mathcal{D}}_n} l(w,x,y) \right) \\
&= \mathrm{E}\left(\frac{1}{|\widetilde{\mathcal{D}}|} \sum_{n=1}^{N} \sum_{(x,y)\in\widetilde{\mathcal{D}}_n} l(w,x,y) \right) && (\mathcal{D}_n \cap \mathcal{D}_{n'} = \emptyset \text{ for } n \neq n') \\
&= \frac{1}{|\widetilde{\mathcal{D}}|} \sum_{n=1}^{N} \mathrm{E}\left(\sum_{(x,y)\in\widetilde{\mathcal{D}}_n} l(w,x,y) \right) \\
&= \frac{1}{|\widetilde{\mathcal{D}}|} \sum_{n=1}^{N} \sum_{(x,y)\in\mathcal{D}_n} \Pr\left\{(x,y) \in \widetilde{\mathcal{D}}_n\right\} \cdot l(w,x,y) \\
&= \frac{1}{|\widetilde{\mathcal{D}}|} \sum_{n=1}^{N} \sum_{(x,y)\in\mathcal{D}_n} \frac{|\widetilde{\mathcal{D}}|}{|\mathcal{D}|} \cdot l(w,x,y) \\
&= \frac{1}{|\mathcal{D}|} \sum_{(x,y)\in\mathcal{D}} l(w,x,y) \\
&\qquad (\mathcal{D} = \cup_{n=1}^{N}\mathcal{D}_n \text{ and } \mathcal{D}_n \cap \mathcal{D}_{n'} = \emptyset \text{ for } n \neq n') \\
&= L(w)
\end{aligned}
$$

The rest is the same as the proof of Theorem 1. □

We note that only when $\tau = 1$, Theorem 2 and Corollary 1 hold for all t. When $\tau > 1$, they only hold for those steps t where synchronization is performed at the end of the previous step $t-1$, because in all the other steps, the condition of $w_n(t-1) = w(t-1)$ may *not* hold. We also note that the unbiasedness in Corollary 2 is only stated for identical model weights at all nodes (see (8) and (9)), which is only the case after synchronization and before the next local computation step.

In general, distributed and centralized gradient descents are *not* equivalent when there are multiple steps of local iteration between synchronization, i.e., when $\tau > 1$. This is because starting from the second step of local iteration

after synchronization, the local gradients at different nodes are computed at different model weights (obtained from the first step of local iteration), thus the average of these gradients may not match with any gradient of the global loss function.

The divergence of the local gradient from the global gradient after $\tau > 1$ steps of local iteration depends on the diversity of data distribution at different local nodes, as discussed in [2]. In the extreme case where all nodes have the same (identical) dataset, the local gradient is always equal to the global gradient and τ can be made arbitrarily large. Using a large value of τ has the benefit of saving the communication bandwidth (used for sharing model weights in the synchronization phase) between nodes, but it may negatively impact the learning depending on how different the datasets at different nodes are.

When the value of τ is bounded, it can be shown that under some convexity assumptions, distributed gradient descent converges to the optimal loss function value as the number of iterations goes to infinity [2], [11]–[14]. However, different amounts of communication and computation resources may be needed to reach the same convergence error with different values of τ. It is important that τ is optimized so that the resource consumption is minimized for a target learning error. A control mechanism that adapts τ according to the communication and computation time in each round was proposed in [2], where it is shown that in general, one should choose a smaller value of τ if the datasets among local nodes are more diverse, and vice versa. Other methods to reduce the communication overhead without impacting the learning error too much also exist. For example, [11] proposed a method that sends the model weights to other nodes only when the change in the local model weight exceeds a threshold.

III. Distributed Data Scoring

The data scoring using (already trained) machine learning models can also be performed in a distributed manner. Here, we note that a full scoring pipeline usually includes multiple stages. For example, a pipeline for face/object recognition from images can include image pre-processing, feature extraction, and classification. Each of these modules may run on a separate node to achieve the most efficient use of communication and computation resources [15]. Similarly, deep neural networks include multiple layers with different amounts of computations and different input/output data sizes. It can be beneficial to run the first few layers of neural network computation locally at the edge, and run the other layers at the remote cloud [16], [17].

As the convolutional layer computations are usually less intensive than computations in the fully connected layers, it is often feasible to run the convolutional layers at the edge. Meanwhile, the data size after a few convolutional layers is usually smaller than the size of the original image. Thus, computing the first few layers locally can save the communication bandwidth, compared to sending the raw data (such as images) to the remote cloud directly. As such, joint local and remote computation can balance the workload and reduce the processing time needed for data scoring. Sending raw data directly to the remote cloud for processing may also pose a risk of privacy leakage, whereas it is generally harder to precisely extract the raw data from the data that has been processed by several convolutional layers. Therefore, performing some amount of computation at the local edge node may also prevent privacy leakage. Obviously, the most secure way of data scoring is to run the entire model on the local edge node. However, edge nodes usually have limited computation and storage capacities and are incapable of running the entirety of a complex machine learning model, such as a deep learning model.

In the general case, a machine learning model can be partitioned and run on more than two (i.e., beyond local vs. remote) nodes. The partitioning of a model onto different nodes is usually dependent on the availability of different types of resources, such as CPU cycles, communication bandwidth, memory size, storage (disk) capacity, and battery life. One can find the way of model partitioning either in a heuristic manner, or more rigorously, using an optimization algorithm. The partition strategy and optimization algorithm can take into account specific aspects of machine learning models, such as in [16]–[19], or alternatively, use abstract computation models with annotated resource demands and apply solutions for generic service placement problems [15], [20]–[25].

It is worth noting that in addition to using partitioned models for efficient data scoring, a model can also be trained or fine-tuned after it has been partitioned onto multiple physical nodes, as discussed in [16], [17]. In a large-scale edge-based distributed learning environment, one can envision that model partition and distributed gradient descent (discussed in Section II) can be jointly used for model training.

IV. Multi-Agent Reinforcement Learning

Reinforcement learning is an area of machine learning which allows a (software) agent to automatically determine the best action to take in order to maximize some notion of cumulative reward in a specific environment. Reinforcement learning is often useful to build systems that learn to perform complex sequential decision tasks. The agents learn how to achieve successful strategies, which lead to the highest long-term rewards, by interacting with its environment through trial and error. Reinforcement learning is situated between supervised and unsupervised learning. In contrast to supervised and unsupervised learning which both have data on which to learn, reinforcement learning makes its own data through experience.

Multi-agent reinforcement learning is referred to multiple agents interacting in the same environment. Multi-agent systems are useful to model many complex problems, such as urban or air traffic control, multi-robot coordination, distributed sensing, resource management [26]. Multi-agent reinforcement learning is useful to model scenarios where control is distributed among different entities. For example, autonomous driving requires multi-agent settings as the control is distributed among each vehicle's host.

A. Fundamentals of Reinforcement Learning

A common way to model reinforcement learning problems is using a Markov decision process, which can be described with a tuple (S, A, T, R), where:

- S is the state space
- A is the action space
- $T(s, a, s')$ is the transition dynamics, describing the probability of transitioning from state s to s' after taking action a in state s
- $R(s, a, s')$ is the reward received when action a is chosen in state s resulting in a transition in state s'.

The problem of solving a Markov decision process is to find a policy (often denoted π) mapping states to actions, which maximizes the accumulated reward. When the transition dynamics T and rewards R are known, the optimal policy can be found using dynamic programming. In the case where T and R are not known, reinforcement learning can be used to learn a near-optimal mapping from states to actions [27].

In reinforcement learning problems, the agent does not know the environment dynamics (i.e., T and R) and needs to learn the optimal policy, which maximizes its long-term expected reward, by interacting with the environment. The agent interacts with the environment by perceiving its states, taking actions and observing the effect of those actions. Feedback (in the form of rewards) allows the agent to adjust its policy: actions that yield to a positive effect will have a higher chance of being selected in the future.

Two popular classes of reinforcement learning techniques are *policy gradient* and *Q-learning*. Policy gradient methods parameterize policies and optimize the policy space using gradient descent. The gradient is estimated by observing the trajectories of executions obtained by following the current policy. The actions that empirically lead to better returns are reinforced [28]. Q-learning parametrizes an action-value functions, $Q(s, a)$[1], and the policy is generated directly from this action-value function. The idea of Q-learning is that if we have a good approximation of $Q(s, a)$ for all state and action pairs, we can obtain the optimal policy π^* by directly selecting the action that maximizes the action-value function, i.e., $\pi^*(s) = \arg\max_a Q(s, a)$. Q learning, unlike policy gradient, updates the Q values without making any assumptions about the actual policy being followed. More details can be found in [29].

For both classes of techniques, the agent has to balance the exploitation/exploration trade-off when picking actions. The agent can either choose to exploit actions that have yielded high reward in the past, or choose to gather more information (continue to explore) in order to achieve potentially better results in the future. The best long-term strategy may sacrifice in the short term.

The Markov decision process framework assumes a single agent is taking actions in the environment. When multiple agents interact simultaneously with environment, the model needs to be extended, as discussed in the next subsection.

[1] $Q(s, a)$ often denoted $Q^\pi(s, a)$ is the expected total reward from state s and action a under policy π: $Q^\pi(s, a) = \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r_t | s, a, \pi]$

B. Multi-Agent Setting

Multiple agents learning simultaneously in an environment gives rise to highly dynamic and non-deterministic environments: each agent has some effect on the environment and so the values of an action depend also on what other agents are doing. From the point of view of each agent, the environment is no longer stationary, i.e., state transitions and rewards are affected by the actions of all the agents. As a result, reinforcement techniques used in single-agent systems (described in IV-A) cannot be applied as they are designed to solve reinforcement learning problems in a stationary environment. Stochastic games are an example of extension of Markov decision processes to multi-agent domain [30].

In multi-agent problems, defining the learning goal can be challenging. Maximization of individual rewards might not lead to the best overall solution. Two different approaches to multi-agent learning can be distinguished depending on whether agents are self-interested and make individual choices (independent learning) or they are teams of cooperative agents (joint-action learning) [31].

- Independent reinforcement learners mutually ignore each other, and perceive interaction with other agents as noise. The advantage of this approach is that single-agent learning algorithms can straightforwardly be applied to a multi-agent setting. However, the stochasticity of the environment means that convergence guarantees from the single-agent setting are lost.
- Joint action learners share a joint return by learning in the space of joint actions, rather than in their individual action space only. Consequently, learners explicitly take the presence of other agents into account. Some examples of joint action learners are minimax-Q [30], and Nash-Q [32].
- Gradient Ascent (or Descent): These methods, based on gradient ascent algorithms, are situated in between independent learning and joint-action learning but are worth mentioning separately as we presented gradient descent techniques in Section II-B. Gradient ascent (descent) methods can be adapted for multi-agent learning by adapting the agents' policies according to the gradient of their individual expected rewards in order to find their local optimum. Some examples of gradient ascent algorithms are infenitesimal gradient ascent [33], generalized infinitesimal gradient ascent [34] and the more recent weighted policy learning [35].

V. Applications

A. Video-Analytics

Distributed machine learning can be used for the training of machine learning models (e.g., deep neural networks) to perform video analytics. For example, in order to increase the security of cities, distributed sensors can capture a huge amount of surveillance videos. Each sensor may capture

different videos which vary depending on the location/angle of the sensors. In order to extract useful information from all sensors (all locations), videos captured from distributed sensors need to be processed and analyzed. Using some form of distributed machine learning has the benefit to enable sensors to collaboratively learn a shared model, built with a rich amount of data, while keeping the videos stored locally.

When using resource constrained systems for video analytics, one needs to consider the model training in networks with poor connectivity and devices with limited resources. Here, learning may need to be offloaded to devices with more resources [18], [19]. In some cases, the videos/data may not be shareable, thus systems must rely on distributed machine learning [2]. In this case, the devices must be prudent in filtering out unnecessary data to process. An example of this is the case of multiple cameras of the same view and determining which sources to process and which are considered redundant.

B. Resource Management in Distributed Infrastructure

Large-scale distributed systems[2] are becoming increasingly complicated and highly dynamic, which makes them challenging to model, predict, and control. Availability of resources (e.g., CPU, memory, storage) at each node may change spontaneously over time, which makes resource management tasks extremely complex. In a cloud computing environment for example, multiple users share cloud resources and the resources need to be re-allocated dynamically and on demand, which make the prediction and management of resources on virtual machines challenging [36].

As infrastructures become increasingly complex, distributed machine learning methods can be applied to solve various problems related to resource management. Distributed machine learning techniques can be used to build an intelligent framework that collects a vast amount of real-time data arising from different nodes across the infrastructure, captures the complex relationship among infrastructure objects (e.g., CPU, network, storage), and extract actionable insights from these distributed data.

Reinforcement learning techniques are also becoming popular for resource management in large computing infrastructures. For example, in [37], multi-agent reinforcement learning is applied to optimize resource allocation in large-scale cluster networks. The learning is distributed to each cluster and each cluster learns only using its local information, without access to the global reward. The empirical results obtained in [37] underline the suitability of multi-agent learning for resource allocation in a large distributed system.

C. Resource Monitoring and Representation in Coalition Environments

Although communication, computation and storage resources in coalition infrastructures can be shared through communication networks among partners, the resources usually belong to individual partners. Despite the coalition relationships, partners may not prefer to reveal details of their infrastructures and conditions, including network topology, resource usage and performance measures, to other partners. (Similar situation also happens in commercial infrastructures of multiple service providers.) In these cases, traditional resource-allocation methods may not apply without detailed knowledge of infrastructures and resources. On the other hand, to facilitate resource sharing, a coalition partner may use machine-learning techniques to monitor and represent the structure and condition of his own infrastructure by the underlying models. In turn, a partner shares the models and associated parameters with other partners so that the latter can decide how to make use of others resources. For example, [38] proposes a method to model the relationship between network performance and usage of communication resources as an arbitrary function, which is approximated by the first and second order terms, and measurements are used to estimate the associated coefficients. One can apply machine-learning techniques to extend the representation capability by using an underlying model more sophisticated than a single-valued function. Furthermore, the model parameters can be determined properly by the online measurements. Knowing the underlying models and parameters for the infrastructure owned by each partner enables all partners to share their resources efficiently. Issues for future investigation include appropriate definition of underlying models and the associated parameter estimations for infrastructure monitor and usage, tradeoffs between model complexity and system performance, and information leaking for the chosen models from partner to partner.

[2]A large-scale distributed system is defined as a collection of loosely coupled processors interconnected by a communication network.

ACKNOWLEDGMENT

This research was sponsored by the U.S. Army Research Laboratory and the U.K. Ministry of Defence under Agreement Number W911NF-16-3-0001. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Army Research Laboratory, the U.S. Government, the U.K. Ministry of Defence or the U.K. Government. The U.S. and U.K. Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

REFERENCES

[1] A. Preece, F. Cerutti, D. Braines, S. Chakraborty, and M. Srivastava, "Cognitive computing for coalition situational understanding," in *Proc. of Workshop on Distributed Analytics InfraStructure and Algorithms for Multi-Organization Federations (DAIS)*, Aug. 2017.

[2] S. Wang, T. Tuor, T. Salonidis, K. K. Leung, C. Makaya, T. He, and K. Chan, "When edge meets learning: Adaptive control for resource-constrained distributed machine learning," in *Proc. of IEEE INFOCOM*, 2018. [Online]. Available: https://ibm.co/2DaHbDs

[3] P. Mach and Z. Becvar, "Mobile edge computing: A survey on architecture and computation offloading," *IEEE Communications Surveys Tutorials*, vol. 19, no. 3, pp. 1628–1656, 3Q 2017.

[4] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "A survey on mobile edge computing: The communication perspective," *IEEE Communications Surveys Tutorials*, vol. 19, no. 4, pp. 2322–2358, 4Q 2017.

[5] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of things: A survey on enabling technologies, protocols, and applications," *IEEE Communications Surveys Tutorials*, vol. 17, no. 4, pp. 2347–2376, 4Q 2015.

[6] M. Chiang and T. Zhang, "Fog and IoT: An overview of research opportunities," *IEEE Internet of Things Journal*, vol. 3, no. 6, pp. 854–864, Dec. 2016.

[7] S. Shalev-Shwartz and S. Ben-David, *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.

[8] S. Bubeck, "Convex optimization: Algorithms and complexity," *Foundations and trends in Machine Learning*, vol. 8, no. 3-4, 2015.

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

[10] O. Shamir and T. Zhang, "Stochastic gradient descent for non-smooth optimization: Convergence results and optimal averaging schemes," in *International Conference on Machine Learning*, 2013, pp. 71–79.

[11] K. Hsieh, A. Harlap, N. Vijaykumar, D. Konomis, G. R. Ganger, P. B. Gibbons, and O. Mutlu, "Gaia: Geo-distributed machine learning approaching LAN speeds," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, pp. 629–647.

[12] A. Agarwal and J. C. Duchi, "Distributed delayed stochastic optimization," in *Advances in Neural Information Processing Systems*, 2011, pp. 873–881.

[13] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing, "More effective distributed ml via a stale synchronous parallel parameter server," in *Advances in neural information processing systems*, 2013, pp. 1223–1231.

[14] W. Dai, A. Kumar, J. Wei, Q. Ho, G. A. Gibson, and E. P. Xing, "High-performance distributed ml at scale through parameter server consistency models." in *AAAI*, 2015, pp. 79–87.

[15] S. Wang, M. Zafer, and K. K. Leung, "Online placement of multi-component applications in edge computing environments," *IEEE Access*, vol. 5, pp. 2514–2533, 2017.

[16] S. Teerapittayanon, B. McDanel, and H. T. Kung, "Distributed deep neural networks over the cloud, the edge and end devices," in *IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, June 2017, pp. 328–339.

[17] J. H. Ko, T. Na, M. F. Amir, and S. Mukhopadhyay, "Edge-host partitioning of deep neural networks with feature space encoding for resource-constrained internet-of-things platforms," *arXiv preprint arXiv:1802.03835*, 2018.

[18] Z. Lu, K. S. Chan, and T. L. Porta, "A computing platform for video crowdprocessing using deep learning," in *Proc. of IEEE INFOCOM*, 2018.

[19] Z. Lu, K. S. Chan, R. Urgaonkar, and T. L. Porta, "On-demand video processing in wireless networks," in *2016 IEEE 24th International Conference on Network Protocols (ICNP)*, Nov 2016, pp. 1–10.

[20] Y. Xiao and M. Krunz, "QoE and power efficiency tradeoff for fog computing networks with fog node cooperation," in *IEEE INFOCOM 2017*, May 2017, pp. 1–9.

[21] L. Tong and W. Gao, "Application-aware traffic scheduling for workload offloading in mobile clouds," in *IEEE INFOCOM 2016*, Apr. 2016.

[22] S. Wang, R. Urgaonkar, T. He, K. Chan, M. Zafer, and K. K. Leung, "Dynamic service placement for mobile micro-clouds with predicted future costs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 4, pp. 1002–1016, Apr. 2017.

[23] L. Wang, L. Jiao, J. Li, and M. Muhlhauser, "Online resource allocation for arbitrary user mobility in distributed edge clouds," in *Proc. of ICDCS*, June 2017, pp. 1281–1290.

[24] S. Wang, R. Urgaonkar, M. Zafer, T. He, K. Chan, and K. K. Leung, "Dynamic service migration in mobile edge-clouds," in *Proc. of IFIP Networking*, May 2015, pp. 1–9.

[25] I.-H. Hou, T. Zhao, S. Wang, and K. Chan, "Asymptotically optimal algorithm for online reconfiguration of edge-clouds," in *Proceedings of the 17th ACM International Symposium on Mobile Ad Hoc Networking and Computing*, ser. MobiHoc '16, New York, NY, USA, 2016, pp. 291–300.

[26] D. Bloembergen, K. Tuyls, D. Hennes, and M. Kaisers, "Evolutionary dynamics of multi-agent learning: A survey," *Journal of Artificial Intelligence Research*, 2015.

[27] R. Sutton and A. Barton, "Reinforcement learning: An introduction," *MIT Press*, 1998.

[28] R. Sutton, D. McAllester, S.P.Singh, and Y.Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *NIPS*, 1999.

[29] C. J. Watkins and P. Dayan, "Q-learning," *Machine Learning*, 1992.

[30] M.L.Littman, "Markov games as a framework for multi-agent reinforcement learning," *ICML*, vol. 94, pp. 157–163, 2015.

[31] C. Claus and C.Boutilier, "The dynamics of reinforcement learning in cooperative multiagent systems," *In AAAI/IAAI*, pp. 746–752, 1998.

[32] J. Hu and M. Wellman, "Nash q-learning for general-sum stochastic games," *The Journal of Machine Learning Research*, vol. 4, pp. 1039–1069, 2015.

[33] S. Singh, M. Kearns, and Y. Mansour, "Nash convergence of gradient dynamics in general-sum games," in *Sixteenth conference on Uncertainty in artificial intelligence*, 2000, pp. 541–548.

[34] M. Zinkevich, "Online convex programming and generalized infinitesimal gradient ascent." in *Intl. Conf. on Machine Learning (ICML-2003)*, 2003.

[35] S. Abdallah and V. Lesser, "A multiagent reinforcement learning algorithm with non-linear dynamics," *Journal of Artificial Intelligence Research*, pp. 521–549, 2008.

[36] A. Saraswathi, Y. Kalaashri, and S. Padmavathi, "Dynamic resource allocation scheme in cloud computing," *Procedia Computer Science*, vol. 47, pp. 30–36, 2015.

[37] C. Zhang, V. Lesser, and P. Shenoy, "A multi-agent learning approach to online distributed resource allocation," in *IJCAI09*, 2009.

[38] C. Liu, K. Leung, and A. Gkelias, "A generic admission-control methodology for packet networks," *IEEE Transactions on Wireless Communications*, vol. 13, pp. 604–617, 2014.