Received October 20, 2021, accepted November 9, 2021, date of publication November 12, 2021,
date of current version November 23, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3127985

T-DFNN: An Incremental Learning Algorithm for Intrusion Detection Systems

MAHENDRA DATA[1,2] AND MASAYOSHI ARITSUGI[3], (Member, IEEE)
[1]Graduate School of Science and Technology, Kumamoto University, Kumamoto 860-8555, Japan
[2]Department of Computer Science, Brawijaya University, Malang 65145, Indonesia
[3]Faculty of Advanced Science and Technology, Kumamoto University, Kumamoto 860-8555, Japan

Corresponding author: Mahendra Data (mahendra.data@ub.ac.id)

ABSTRACT Machine learning has recently become a popular algorithm in building reliable intrusion detection systems (IDSs). However, most of the models are static and trained using datasets containing all targeted intrusions. If new intrusions emerge, these trained models must be retrained using old and new datasets to classify all intrusions accurately. In real-world situations, new threats continuously appear. Therefore, machine learning algorithms used for IDSs should have the ability to learn incrementally when these new intrusions emerge. To solve this issue, we propose T-DFNN. T-DFNN is an algorithm capable of learning new intrusions incrementally as they emerge. A T-DFNN model is composed of multiple deep feedforward neural network (DFNN) models connected in a tree-like structure. We examined our proposed algorithm using CICIDS2017, an open and widely used network intrusion dataset covering benign traffic and the most common network intrusions. The experimental results showed that the T-DFNN algorithm can incrementally learn new intrusions and reduce the catastrophic forgetting effect. The macro average of the F1-score of the T-DFNN model was over 0.85 for every retraining process. In addition, our proposed T-DFNN model has some advantages in several aspects compared to other models. Compared to the DFNN and Hoeffding tree models trained with a dataset containing only the latest targeted intrusions, our proposed T-DFNN model has higher F1-scores. Moreover, our proposed T-DFNN model has significantly shorter training times than a DFNN model trained using a dataset containing all targeted intrusions. Even though several factors can affect the duration of the training process, the T-DFNN algorithm shows promising results in solving the problem of ever-evolving network intrusion variants.

INDEX TERMS Network intrusion detection, incremental learning, catastrophic forgetting, deep learning, classification algorithm.

I. INTRODUCTION

Intrusion detection systems (IDSs) are crucial components in the current computing infrastructures to identify malicious computer network activities [1], [2]. Along with the growth of network-based applications and systems, the number of cyberthreats is increasing [3]. IDSs play a vital role in cybersecurity [4] by forewarning security administrators about malicious activities such as distributed denial-of-service (DDoS), port scan, and SQL injection attacks. Having reliable IDSs is a mandatory safeguard for protecting computing infrastructures against ever-increasing issues of intrusive activities [5].

The idea of creating reliable IDSs with improved accuracy and fewer requirements for human knowledge drives the development of machine learning-based IDSs. Machine learning algorithms such as artificial neural networks (ANNs), fuzzy logic, and support vector machines (SVMs) have become extensively used in IDS studies [6]–[8]. These machine learning algorithms can extract knowledge from datasets through complex pattern-matching processes [6]. Extracting this knowledge requires most machine learning algorithms to be trained using datasets containing all targeted intrusions [9].

The requirement of acquiring datasets containing all targeted intrusions raises an important issue. In real-world situations, security experts collect intrusion data incrementally because intrusions do not emerge at once but gradually over time. It is possible to create a new model for these new intrusions. However, training a model using a dataset containing all intrusions may take a long time. Additionally, it is difficult to modify the previously trained model to accommodate new

The associate editor coordinating the review of this manuscript and approving it for publication was Yi Zhang.

intrusion variants because the training process is static and only performed once using datasets containing all targeted intrusions. To solve this problem, we need to develop an algorithm that can learn incrementally with a shorter training time as new intrusions emerge. However, the *catastrophic forgetting* problem becomes the main challenge to realizing this idea [10]–[12].

Catastrophic forgetting is a classic problem faced by many machine learning models and algorithms [12]. Assume we have trained a classification model; then, we retrain this model using a new dataset containing new classes. In this situation, most current classification models may forget how to classify the old classes. Goodfellow *et al.* [12] explain that when we train a machine learning model with a convex objective, it will always end with the same configuration at the end of the training process, regardless of how it was initialized. For example, a support vector machine (SVM) that is trained on two different tasks will completely forget how to perform the first task. If this retrained SVM model can correctly classify some data from the old task, it is only due to the similarity of both old and new tasks.

This research aims to solve the two problems we previously mentioned: the problem of ever-evolving network intrusion variants and the catastrophic forgetting problem. To solve these issues, we propose an incremental learning algorithm capable of learning new intrusions incrementally as they emerge. Our proposed method is composed of multiple deep feedforward neural network (DFNN) models. Each neuron in the output layer of a DFNN model is linked with another DFNN model, creating a tree structure. Hence, we named our proposed method tree deep feedforward neural networks (T-DFNN). The tree structure in T-DFNN is expandable. New nodes can be added when learning new intrusion variants.

Note that we do not intend to propose our incremental learning model to replace the current standard models, which use a dataset containing all intrusions in the training process. Instead, we intend to propose a model that works alongside existing models. When new intrusions emerge, training a new model using a dataset containing all intrusions is a prolonged process. Our research solves this issue by providing an incremental learning model with a shorter training time without sacrificing the model's performance. While a current standard model is being prepared, this incremental learning model can be used during the interim. The reason is that even though the current standard model has a slow training process, it has a relatively simpler structure, thereby having a faster classification process. This simpler structure is beneficial when used in long-term scenarios.

T-DFNN is a supervised machine learning algorithm. It needs a labeled dataset to perform the training process. In this research, we used Canadian Institute for Cybersecurity's intrusion detection evaluation dataset 2017 (CICIDS2017) to evaluate our proposed algorithm. CICIDS2017 is a reliable and labeled network intrusion dataset that covers both benign and intrusion traffic. The intrusion traffic in this dataset consists of the most common network intrusions. The creator of this dataset [13] did not design this dataset specifically for the incremental learning problem. Therefore, we divided CICIDS2017 into several batches and then trained the models in our experiment using these batches sequentially to simulate the incremental learning process.

We should note that the incremental learning term has been used rather loosely in the literature. This term refers to several concepts, such as incremental network growing and pruning, online learning, or relearning of formerly misclassified instances [14]. The incremental learning term in this study refers to a machine learning algorithm that meets the following criteria:

1) It can learn new information, e.g., new network intrusion variants.
2) It can preserve previously acquired knowledge or, in other words, it should not suffer from the catastrophic forgetting problem.

In summary, we make the following contributions in this paper:

1) We propose T-DFNN: an incremental learning algorithm for IDSs. A T-DFNN model is composed of multiple DFNN models connected in a tree-like structure. This tree structure model can be partially retrained to accommodate new intrusion variants when they emerge.
2) The T-DFNN algorithm can reduce the catastrophic forgetting effect. When a dataset of new intrusions emerges, it preserves the trained nodes while expanding the model by adding new nodes to classify the new intrusions. This mechanism reduces the catastrophic forgetting effect on the model.
3) The T-DFNN algorithm can shorten the training time by limiting the old training dataset needed in the retraining process. In the T-DFNN algorithm, the training dataset in each node is selected based on the classification results of the parent node. Only data classified as the same parent node's output label are used in each node.

The remainder of this paper is arranged as follows: Section II presents related work; Section III describes our proposed incremental learning algorithm; Section IV explains the experimental setup of this research; Section V presents a summary of the experimental results; Section VI discusses the challenges of implementing the proposed incremental algorithm in the network intrusion detection problem; and Section VII provides our conclusion and future work.

II. RELATED WORK

A. INTRUSION DETECTION SYSTEMS

IDSs are security tools that identify malicious network activities on computer infrastructures. They monitor network traffic and system logs to find malicious network activities that conventional firewalls cannot filter [2], [6]. There are two main categories of IDSs based on their detection method: signature-based and anomaly-based IDSs [6], [15]. Signature-based IDSs use pattern-matching techniques to find known malicious network activities. Signature-based IDSs are also known as knowledge-based detection or misuse detection. In contrast, anomaly-based IDSs analyze network traffic to find a significant deviation between observed traffic and acknowledged traffic behavior. Anomaly-based IDSs interpret this deviation of behavior as an intrusion [2], [4], [6], [7], [15]. One approach in building anomaly-based IDSs is using machine learning algorithms [4].

Most of the machine learning models used in previous studies of IDSs are static models and trained using a dataset containing all targeted intrusions. Only a few of them raised the issue of ever-evolving network intrusion variants [16]. Studies by Constantinides *et al.* [16], Chen *et al.* [17], Yi *et al.* [18], Xu *et al.* [19], and Jiang *et al.* [20] are examples of those proposing an incremental learning method to solve the problem of ever-evolving network intrusion variants. Most of these studies utilized support vector machines (SVMs) in their proposed incremental learning methods. SVMs belong to the supervised machine learning algorithm category commonly used for classification problems. Despite the prominent properties of SVMs, the training complexity of SVMs is highly dependent on the size of a dataset. Thus, SVMs are not as favored for large-scale data mining as for pattern recognition [21].

To test the performance of proposed IDS methods, researchers often used publicly available intrusion datasets [15]. In this research, we used CICIDS2017. It is a newer IDS dataset than the KDD Cup 1999 dataset [22], which has been commonly used in previous IDS incremental learning studies [17]–[20]. We did not use the KDD Cup 1999 dataset because it has several deficiencies. One of the critical deficiencies of the KDD Cup 1999 dataset is the significant number of redundant records, which causes a bias toward the more frequent records [23]. Another unfortunate deficiency of the KDD Cup 1999 dataset is the fact that this dataset is very old [15]. It was created in 1999 for The Third International Knowledge Discovery and Data Mining Tools Competition. Hindy *et al.* [15] explained that depending solely on old datasets cannot help the advancement of IDSs. Thus, it is better to use newer intrusion datasets that cover recent variants of intrusions.

B. INCREMENTAL LEARNING

Recently, many studies have preferred deep learning using artificial neural networks (ANNs) to process large-scale data [24]–[26]. Deep learning using ANNs is one of the popular algorithms for learning information from complex datasets. Deep learning using ANNs can create complex models compared to traditional probabilistic machine learning techniques [24]. Therefore, they have been broadly used for IDSs [2], [6]–[8], [15].

Despite being broadly used, most deep learning studies in IDSs do not focus on incremental learning. Instead, they focus on improving the classification performance by utilizing deep learning algorithms. In contrast, incremental learning using deep learning algorithms is thriving in image processing fields. For example, Roy *et al.* [27] used deep convolutional neural networks (CNNs) to build a hierarchical model for incremental learning. Their proposed model organizes the incrementally available images into several superclasses based on features. In the training process, new classes of images are added to the hierarchical model as the subclasses. The retraining processes are limited in the affected superclasses to reduce the computational overhead. Sarwar *et al.* [28] also proposed an incremental learning algorithm using CNNs. Unlike Roy *et al.*'s approach, Sarwar *et al.* used a partial network sharing method in their incremental learning method. Inspired by transfer learning techniques, Sarwar *et al.*'s method splits CNN layers into shared and classification layers. The first several layers become the shared layers, and the rest become the classification layers. When the model is retrained using new image data, the classification layers are cloned to classify the new images. The result of this cloning process is a tree structure model of shared and classification layers. Both methods use different approaches to generate an incremental learning model. However, the structure of both models resembles a tree structure. Additionally, both methods limit the retraining process in the new branches of the tree structure to reduce the computational overhead. We adopted the idea of using a tree-structured model for incremental learning in our proposed method.

We used DFNN models in the T-DFNN algorithm. The DFNN model is one variant of ANNs used for deep learning. We combined a tree structure and DFNN models to build an incremental learning algorithm that can preserve knowledge from the previous training while reducing the retraining process's computational overhead. Multiple nodes of DFNN models are used in the T-DFNN model to classify the given input data. Unlike Roy *et al.*'s approach [27], we did not group similar intrusion classes into one superclass. Instead, we utilized a previously trained model to find the old and new classes classified as the same output label.

In our experiment, we compared our proposed method with another tree-based incremental learning algorithm. We chose a well-known incremental decision tree algorithm, namely, the Hoeffding tree algorithm [29]. The Hoeffding tree algorithm can learn from large-scale incremental data. It exploits the fact that a small portion of data is often enough to select an optimal splitting attribute of the dataset. Thus, this algorithm is commonly used to process incremental data [20]. The Hoeffding tree algorithm has been implemented in several popular machine learning libraries, such as scikit-multiflow [30] and Weka [31].

III. METHODS

The T-DFNN algorithm covers both training and classification processes. The training process of the T-DFNN algorithm generates an incremental learning model. This incremental learning model is then used in the classification process

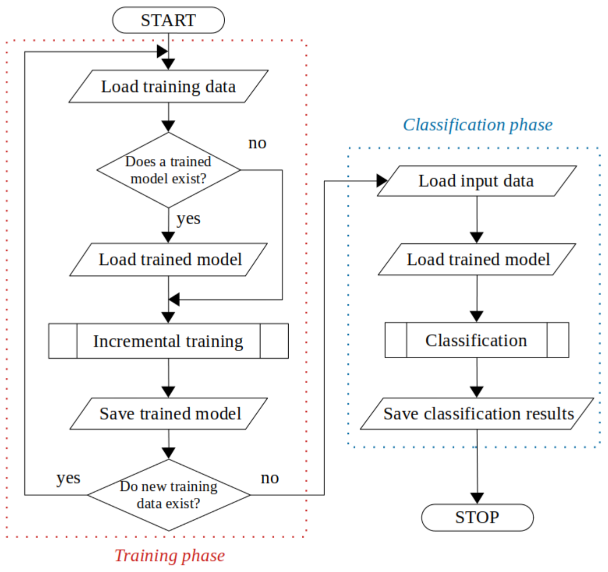


FIGURE 1. Training and classification flow of the T-DFNN algorithm.

to classify the input data. The T-DFNN algorithm aims to classify ever-growing network intrusions efficiently. Thus, the training process in the T-DFNN algorithm is designed to preserve the knowledge learned by the previous model while reducing the quantity of old training data used in the training processes.

T-DFNN model is a tree-structured model. It consists of a root node and may have several leaf and internal nodes. When a T-DFNN model is retrained, the trained nodes are not modified to preserve the previously learned knowledge. Instead, new nodes are created. These newly created nodes are then connected with the existing nodes to create a tree-structured model. The innovation of our proposed T-DFNN algorithm is its ability to distribute the training data to each node while limiting the number of old training data used to train these new nodes, thereby shortening the time needed to retrain the model.

Figure 1 shows the training and classification flow of the T-DFNN algorithm. The main feature of the T-DFNN algorithm in the training process is the mechanism to reuse a previously trained model to learn new training data. Thus, a trained model is saved after each incremental training process. Except for the first training process, the saved model is loaded along with the new training data. This saved model is then retrained to classify the new input data.

One of the essential components in the T-DFNN model is the T-DFNN node. There are two important items in the T-DFNN node: a DFNN model and a map of output labels. A DFNN model processes the input data and classifies them into several output labels. These output labels can be linked with other T-DFNN nodes using a map. A *map* is a data structure that consists of key-value pairs. In the map of the output label, the output labels become keys, and the values of these keys are either other nodes or NULL values. A NULL value indicates that the output label is not linked

Pseudocode 1 T-DFNN Node

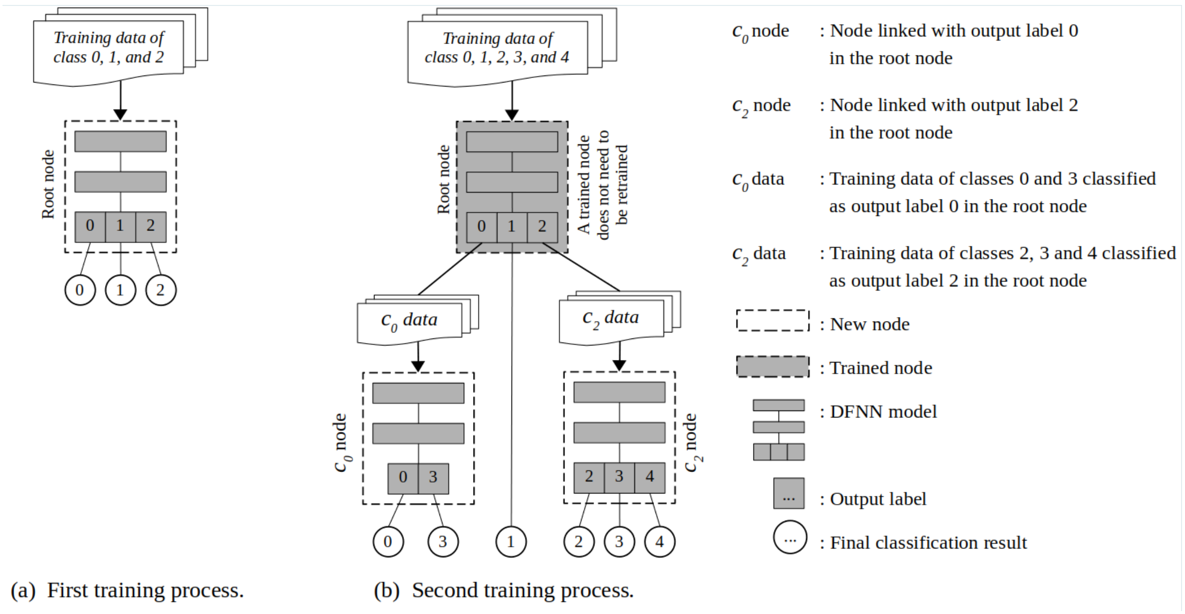
```
class TDFNNNode():
    attribute: $M$: DFNN model $\mu$: map of output labels
               of $M$

1       procedure Instantiation:
2           $M \leftarrow$ create a DFNN model
3           $\mu \leftarrow$ empty map
4       procedure Train($X$, $Y$):
            input: $X$: array of training data $Y$: array of labels
                   of training data
            // Train DFNN model $M$ using $X$
               and $Y$
5           DFNNTraining($M$, $X$, $Y$)
6           foreach $L \in$ output labels of $M$ do
                // Map output label $L$ to $NULL$
7               $\mu(L) \leftarrow NULL$
8           end
9       procedure Classify($X$):
            input: $X$: array of input data
            output: $YC$: array of labels of classified data
            // Classify data $X$ using DFNN
               model $M$
10          $YC \leftarrow$ DFNNClassification($M$, $X$)
11          return $YC$
12      procedure GetLinkedNode($L$):
            input: $L$: label
            output: $N$: TDFNNNode
            // Get the node linked with
               output label $L$
13          $N \leftarrow \mu(L)$
14          return $N$
15      procedure SetLinkedNode($L$, $N$):
            input: $L$: label $N$: TDFNNNode
            // Map output label $L$ to node $N$
16          $\mu(L) \leftarrow N$
```

with any node. Pseudocode 1 shows the implementation of a T-DFNN node.

As we previously mentioned, the innovation of the T-DFNN algorithm is its mechanism to distribute the training data to several nodes and limit the number of old training data used in the training process. In the retraining process, several new nodes can be created. In addition to the new training data, old training data are also used to train these new nodes. However, the T-DFNN algorithm limits the number of old training data used to train these new nodes. Each new node only uses old training data classified as its parent node's output label. The details of this training process are described in Section III-A.

The classification process in the T-DFNN algorithm is performed in several steps. First, the input data are processed using the root node of the T-DFNN model. Then, the outputs



Training data of class 0, 1, and 2
Root node
0 1 2
0 1 2
(a) First training process.
Training data of class 0, 1, 2, 3, and 4
Root node
A trained node does not need to be retrained
c_0 data
c_2 data
c_0 node
c_2 node
0 3
2 3 4
0 3 1 2 3 4
(b) Second training process.
c_0 node : Node linked with output label 0 in the root node
c_2 node : Node linked with output label 2 in the root node
c_0 data : Training data of classes 0 and 3 classified as output label 0 in the root node
c_2 data : Training data of classes 2, 3 and 4 classified as output label 2 in the root node
: New node
: Trained node
: DFNN model
: Output label
: Final classification result

FIGURE 2. **T-DFNN models in its first and second training processes.**

of the root node's classification process are used to determine whether the classification process will continue using the child nodes or be terminated. These processes have similarities with the classification process in the decision tree. The difference is that in the T-DFNN algorithm, we used the DFNN model outputs as the splitting rules. We discuss the details of the classification process in section III-B.

A. INCREMENTAL TRAINING ALGORITHM

After loading the first batch of training data, we process it using the incremental learning procedure shown in Pseudocode 2. In this first training process, we create a node. Consequently, this node becomes the root node of the T-DFNN model. This root node is then trained using the given training data. Last, we map the output labels of the DFNN model to NULL to indicate that the output labels are not linked with any node. We describe the initial training process in Pseudocode 2, Lines 1 to 3.

The retraining process begins by loading the training data and the trained T-DFNN model. We do not retrain existing nodes to prevent the catastrophic forgetting effect. Instead, we use the existing nodes to find suitable output labels to be linked with the new nodes. These new nodes are trained to classify the new training data.

We describe the process of finding suitable output labels to be linked with new nodes in Pseudocode 2, Lines 4 to 18. The process begins by classifying the new training data using the root node's DFNN model, as shown in Pseudocode 2 Line 5. The root node's DFNN model misclassifies new training classes as old classes because we did not train the root node's DFNN model to classify these new training data. We illustrate this condition in Figure 2(b). In Figure 2(b), the root node's DFNN model misclassifies class 4 training data as output label 2. It is also possible that the root node's DFNN model misclassifies a new training class as two or more old classes. For example, in Figure 2(b), the root node's DFNN model misclassifies part of the class 3 training data as output label 0 and the other part as output label 2.

There are two possible conditions when multiple classes are classified as the same output label:

1) The output label is linked with a node.
2) The output label is not linked with any node.

In the first condition, where an output label is linked with another node, we rerun the training process using the linked node as the new root. This recursive training process continues until we find an output label with no linked node. In other words, the second condition occurs. In this second condition, we create and train a new node. Only the training data classified as the same parent node's output label are used in the training process. Last, we link the parent node's output label with this newly trained node by mapping the parent node's output label to the new node. We describe these steps in Pseudocode 2, Lines 9 to 15.

One key component of the T-DFNN incremental training algorithm is selecting appropriate training data used in each node. This selection process limits the quantity of training data in each node. We describe the training data selection process in Pseudocode 3. Training data in each node are previously classified as the same parent node's output label. For example, in Figure 2(b), the training data used in the c_0 node are class 0 and part of the class 3 training data because they are classified as output label 0 in the root node. Likewise, the training data used in the c_2 node are class 2, part of class 3, and class 4 training data because they are classified as output label 2 in the root node. Training data of a class

Pseudocode 2 T-DFNN Incremental Training

```
procedure IncrementalTraining($X$, $Y$, $N$):
    input: $X$: array of training data $Y$: array of labels of training data $N$: TdfnnNode
    output: $N$: TdfnnNode
1   if $N = NULL$ then
2       $N \leftarrow$ create a TdfnnNode instance // create TdfnnNode and run Instantiation procedure
3       Call Train($X$, $Y$) procedure of $N$ // Train DFNN model of $N$ and map its output labels
        to $NULL$
4   else
5       $YC \leftarrow$ Call Classify($X$) procedure of $N$ // Run Classify procedure in node $N$
6       foreach $L \in unique(YC)$ do // Loop for each unique value of predicted output labels
7           $XL, YL \leftarrow$ SelectTrainingData($X, Y, YC, L$) // Select training data classified
            as $L$
8           if $any\ values\ in\ YL \neq L$ then // If there are new classes
9               $linkedN \leftarrow$ Call GetLinkedNode($L$) procedure of $N$ // Get TdfnnNode linked with
                output label $L$
10              if $linkedN \neq NULL$ then // Output label L is linked to $linkedN$
11                  $linkedN \leftarrow$ IncrementalTraining($XL, YL, linkedN$)
12              else // Output label L is not linked to any node
13                  $newN \leftarrow$ IncrementalTraining($XL, YL, NULL$)
14                  Call SetLinkedNode($L, newN$) procedure of $N$ // Map output label $L$ in node $N$
                    to node $newN$
15              end
16          end
17      end
18  end
19  return $N$
```

Pseudocode 3 Training Data Selection

```
procedure SelectTrainingData($X$, $Y$, $YC$, $L$):
    input: $X$: array of training data $Y$: array of labels of
           training data $YC$: array of labels of classified
           data $L$: label
    output: $XL$: array of selected data $YL$: array of
            labels of selected data
    // initiate empty array
1   $XL, YL \leftarrow$ empy array
2   $j \leftarrow 0$
3   $num \leftarrow$ number of row in $X$
4   for $i \leftarrow 0 \ldots (num - 1)$ do
        // If row $i$ of array $YC$ is $L$
5       if $row(YC, i) = L$ then
            // Copy row $i$ of array $X$
6           $row(XL, j) = row(X, i)$
            // Copy row $i$ of array $Y$
7           $row(YL, j) = row(Y, i)$
8           $j = j + 1$
9       end
10  end
11  return $XL, YL$
```

may be divided into several parts and trained using different nodes. For example, class 3 training data used in the c_0 node were previously classified as output label 0 in the root node. In contrast, class 3 training data used in the c_2 node were previously classified as output label 2 in the root node. Thus, there are no overlap training data used in those two nodes.

We do not need to use the training data of all old classes in the training process of new training data. Training data of some old classes are only needed when the old and new classes are classified as the same output label. For example, in Figure 2(b), we do not need to use class 1 training data because when we classify the new training data using the root node, there are no new classes classified as output label 1. This method reduces the quantity of old training data used for the training process of the new classes.

B. CLASSIFICATION ALGORITHM

The T-DFNN classification algorithm is a recursive process. As shown in Pseudocode 4, the first step is classifying the input data using the root node's DFNN model. Then, we check the linked node for each output label. If the output labels of the root node's DFNN model are linked with other nodes, we run the classification algorithm recursively using the linked nodes. This recursive process stops when the

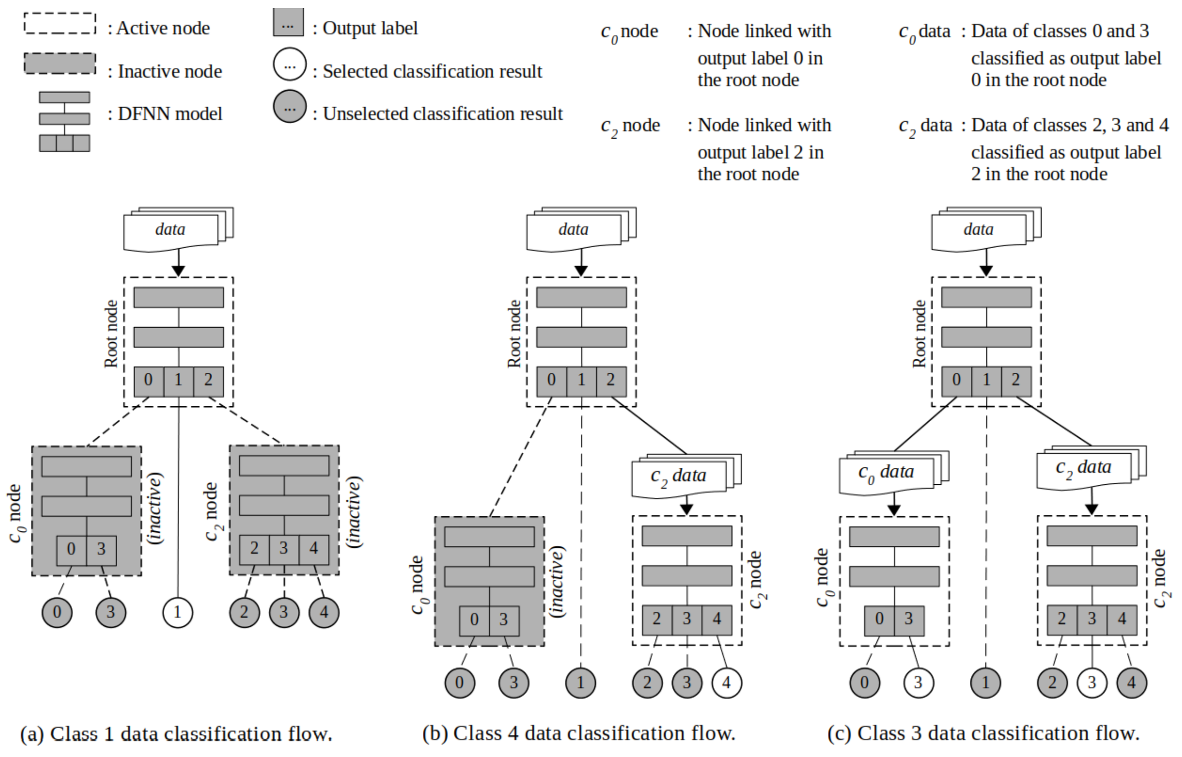

: Active node
: Inactive node
: DFNN model
: Output label
: Selected classification result
: Unselected classification result
c0 node : Node linked with output label 0 in the root node
c2 node : Node linked with output label 2 in the root node
c0 data : Data of classes 0 and 3 classified as output label 0 in the root node
c2 data : Data of classes 2, 3 and 4 classified as output label 2 in the root node
data
Root node
c0 node
c2 node
(inactive)
c0 data
c2 data
(a) Class 1 data classification flow.
(b) Class 4 data classification flow.
(c) Class 3 data classification flow.

FIGURE 3. T-DFNN model classification processes.

DFNN model's output label in the last node is not linked with any node.

When the DFNN model of a node classifies the input data, there are two possible conditions regarding each output label:

1) The output label is not linked with any node.
2) The output label is linked with a node.

In the first condition, the input data classified as this output label is not processed further. Thus, this output label becomes the final classification result. We illustrate this condition in the root node's output label 1 in Figure 3(a). In Figure 3(a), the root node classifies the input data as output label 1. Because there is no node linked with output label 1, the final classification result is class 1.

In the second condition, we run the classification algorithm recursively using a node linked with the output label. Not all input data are used in this recursive process. Only input data classified as the output label of the linked node will be selected. We describe this data selection process in Pseudocode 5. This recursive classification process stops when an output label with no linked node is found. Thus, the last output label with no linked node becomes the final classification result. To do that, we need to update the current node's classification result using the recursive process's output. Only the classification results of the input data used in the recursive process are updated. We describe the process of updating the classification results in Line 7 of Pseudocode 4, which is explained in more detail in Pseudocode 6.

We illustrate the recursive classification process in Figure 3(b). In Figure 3(b), the root node's DFNN model classifies input data as output label 2. Because output label 2 in the root node is linked with the c_2 node, we recursively run the classification process using the c_2 node. Then, the c_2 node classifies the input data as output label 4. Because there is no node linked with output label 4 in the c_2 node, the final classification result of the input data is class 4.

The T-DFNN model may classify a new class using two or more nodes. For example, in Figure 3(c), the root node's DFNN model classifies input data as output labels 0 and 2. Both output labels are linked with different nodes. Output labels 0 and 2 in the root node are linked with c_0 and c_2 nodes, respectively. Under this condition, the input data are split into two groups. The first group is the input data classified as output label 0 by the root node's DFNN model, while the second group is the input data classified as output label 2 by the root node's DFNN model. These two groups of input data are processed further using different nodes. The first group is processed by the c_0 node, while the other group is processed by the c_2 node. Finally, both DFNN models of nodes c_0 and c_2 classify the input data as output label 3. Because there is no node linked with output label 3 in both nodes, the final classification result of the input data is class 3.

The classification processes of input data in the T-DFNN model involve several nodes at different tree levels. Thus, the time needed to classify the input data is the accumulation of node classifications from the root to the leaf node. In detail,

Pseudocode 4 Classification

```
procedure CLASSIFICATION($X$, $N$):
    input: $X$: array of input data $N$: TDFNNNODE
    output: $YC$: array of labels of classified data
1   $YC \leftarrow$ Call Classify($X$) procedure of $N$        // Run Classify procedure in node $N$
2   foreach $L \in unique(YC)$ do        // Loop for each unique value of predicted output labels
3       $linkedN \leftarrow$ Call GetLinkedNode($L$) procedure of $N$        // Get TDFNNNODE linked with output
        label $L$
4       if $linkedN \neq NULL$ then        // Output label $L$ is linked to $linkedN$
5           $XL, YI \leftarrow$ SelectData($X$, $YC$, $L$)        // Select data labelled as $L$
6           $newYC \leftarrow$ CLASSIFICATION($XL$, $linkedN$)        // Recursively run CLASSIFICATION
7           $YC \leftarrow$ UPDATELABEL($YC$, $newYC$, $YI$)        // Update the classification results $YC$
8       end
9   end
10  return $YC$
```

Pseudocode 5 Data Selection

```
procedure SelectData($X$, $YC$, $L$):
    input: $X$: array of input data $YC$: array of labels of
           classified data $L$: label
    output: $XL$: array of selected data $YI$: array of row
            index of selected data
    // initiate empty array
1   $XL, YI \leftarrow$ empy array
2   $j \leftarrow 0$
3   $num \leftarrow$ number of row in $X$
4   for $i \leftarrow 0 \ldots (num - 1)$ do
        // If row $i$ of array $YC$ is $L$
5       if $row(YC, i) = L$ then
            // Copy row $i$ of array $XL$
6           $row(XL, j) = row(X, i)$
            // Copy $i$ as row $j$ of array $YI$
7           $row(YI, j) = i$
8           $j = j + 1$
9       end
10  end
11  return $XL, YI$
```

the classification time of a T-DFNN model is estimated by calculating the classification time of its root node using Equation 1.

$$time(N) = \begin{cases} t & d(N) = 0 \\ t + max(\{time(n) | n \in child(N)\}) & d(N) > 0 \end{cases} \quad (1)$$

where

- N is a node in the T-DFNN model,
- t is the DFNN model's classification time in node N,
- $d(N)$ is the degree of node N,
- $child(N)$ is a set of node N's child nodes.

If the degree of node N is 0, i.e., node N does not have any child node, the classification time of node N is equal to its DFNN model's classification time. Otherwise, the classification time of node N is the sum of its DFNN model's classification time and the longest classification time of its child nodes.

IV. EXPERIMENTAL SETUP

A. DATASET

In this experiment, we used CICIDS2017. It is a reliable and labeled publicly available network intrusion dataset [13]. CICIDS2017 contains benign and common network intrusion flows, which match the features of a reliable benchmark dataset proposed by Gharib *et al.* [32]. These features are anonymity, attack diversity, available protocols, complete interaction, complete capture, complete traffic, complete network configuration, labeling, feature set, heterogeneity, and metadata. Thus, researchers are attracted to develop machine learning models and algorithms [33].

CICIDS2017 consists of 84 network traffic features extracted from raw network packets using CICFlowMeter software [34], which is publicly available on the Canadian Institute for Cybersecurity website. Similar to the previous study [35], we decided to remove six features from the dataset: Flow ID, Protocol, Timestamp, Source IP, Destination IP, and Source Port. From the network topology perspective, the values of these features will differ from real-world scenarios because this dataset was generated at an isolated network. Additionally, we removed 288,602 rows with missing labels from the dataset. After the unused and unlabeled data were removed, the final dataset consisted of 2,830,743 rows and 78 features. There are fifteen classes in the CICI2017 dataset, one of which is benign traffic, and the others are fourteen different types of intrusion traffic. Table 1 shows the traffic distributions in this dataset.

We implemented several preprocessing steps to CICIDS2017. These preprocessing steps are required for two main reasons. First, our proposed model uses DFNN

TABLE 1. Dataset batches and distributions.

Batch	Class label	Encoded class label	Sample		
			Training	Evaluation	Total
1	Benign	0	1,818,477	454,620	2,273,097
	FTP-patator	1	6,350	1,588	7,938
	Bot	2	1,573	393	1,966
	Web attack-XSS	3	522	130	652
2	DoS-hulk	4	184,858	46,215	231,073
	Port scan	5	127,144	31,786	158,930
	DDoS	6	102,421	25,606	128,027
	Web attack-brute force	7	1,206	301	1,507
3	DoS-golden eye	8	8,234	2,059	10,293
	SSH-patator	9	4,718	1,179	5,897
	DoS-slowloris	10	4,637	1,159	5,796
	DoS-slowhttptest	11	4,399	1,100	5,499
4	Infiltration	12	29	7	36
	Web attack-SQL injection	13	17	4	21
	Heartbleed	14	9	2	11
	Total		2,264,594	566,149	2,830,743

Pseudocode 6 Updating Data's Label

```
procedure UpdateLabel($YC, newYC, YI$):
    input: $YC$: array of labels of classified data $newYC$:
           array of labels of new classified data $YI$:
           array of row index of selected data
    output: $YC$: array of labels of classified data
1   $num \leftarrow$ number of row in $newYC$
2   for $i \leftarrow 0 \ldots (num - 1)$ do
        // Copy row $i$ of array $YI$
3       $j \leftarrow row(YI, i)$
        // Copy row $i$ of array $newYC$
4       $row(YC, j) \leftarrow row(newYC, i)$
5   end
6   return $YC$
```

models to classify data in the nodes. Several preprocessing steps, such as normalization, should be applied to this dataset before further processing. Second, the CICIDS2017 was not explicitly designed for incremental learning. To simulate the incremental learning process, we divided this dataset into several batches.

The preprocessing steps used in this experiment are as follows:

1) Replace missing values on each feature using the mean value of its class.
2) Replace infinite values on each feature using the maximum value of its class.
3) Replace negative values on each feature using the minimum of its class.
4) Normalize the features using unity-based normalization.
5) Group the dataset into several batches.
6) Split the dataset into training and evaluation groups.

Equation 2 is used to normalize the feature by restricting the range of the values between 0 and 1. x_i, x_{min}, and x_{max} in Equation 2 represent the value, the minimum value, and the maximum value of each feature, respectively. The purpose of this normalization is to prevent a feature from outweighing the other features. To simulate the incremental learning process, we divide the dataset into several batches. Each batch contains several classes. After dividing the dataset into several batches, we split the data in each batch into training and evaluation data. The ratio between training and evaluation data is 4 to 1. Table 1 shows the batches and data distribution in each batch. In the training process, some classes from old batches were used. For example, when we created a new node that needs to classify old and new training classes, some old training data are used to train this new node. We illustrate this case in Figure 2(b). In Figure 2(b), the training data used in the c_0 node are training data of class 0 and class 3 classified as output label 0 in the root node.

$$x_i = \frac{x_i - x_{min}}{x_{max} - x_{min}} \tag{2}$$

B. EXPERIMENTAL SCENARIOS

We built and trained our T-DFNN model using Keras 2.2.4, which runs on top of the machine learning platform TensorFlow 2.0. The DFNN model used in a T-DFNN node consists of three dense layers. The number of neurons in the first layer is 120, and the number of neurons in the second layer is 78. While the numbers of neurons in the first and second layers are static, the number of neurons in the last layer is dynamic. The last layer of the DFNN model is a classification layer. Its number of neurons depends on the number of classes classified in this node. We illustrate this behavior in Figure 2(b). In Figure 2(b), the numbers of neurons of the last layer in c_0 and c_2 nodes are different; the c_0 node has two, while the c_2 node has three.

The hyperparameters used by the DFNN model in every node of the T-DFNN model are identical. For the optimizer, we used the Adam optimizer with a 0.001 learning rate. By default, the training process continues until 1000 epochs. However, we used the early stopping method, which monitors the classification loss. If there was no improvement in 50 epochs, then the training process stopped.

In the experiment, we compared the T-DFNN model to two DFNN models. We named these two DFNN models DFNN-batch and DFNN-all. DFNN-batch and DFNN-all models had an identical network structure and hyperparameters to the DFNN model used in the T-DFNN nodes. However, these two DFNN models and the T-DFNN model utilized the training data in different ways. The DFNN-batch model only used training data from the current batch, while the DFNN-all model used training data from old and current batches. Similar to the DFNN-all model, the T-DFNN model also used training data from the old and current batches. However, not all data from the old batches were used in the T-DFNN. The old training data were only used if they were needed to train new nodes, as we described in section III-A.

The purpose of comparing the T-DFNN model with these two DFNN models was to measure the effectiveness of the T-DFNN algorithm in addressing ever-evolving network intrusion variants and catastrophic forgetting problems. The comparison between the T-DFNN model and the DFNN-batch model demonstrated how severe the catastrophic forgetting problem affects the performance of network intrusion detection. It also demonstrated the effectiveness of the T-DFNN algorithm in reducing the effect of the catastrophic forgetting problem. The purpose of comparing the T-DFNN with DFNN-all models is to analyze the advantages and training performance of the proposed incremental learning algorithm.

We also compared our proposed T-DFNN model with a Hoeffding tree model [29]. As discussed in Section II, the Hoeffding tree model is a well-known incremental decision tree algorithm capable of learning from large-scale incremental data. To implement the Hoeffding tree algorithm, we used the HoeffdingTreeClassifier method provided by the scikit-multiflow library [30], which is based on MOA [36]. For the hyperparameters of this Hoeffding tree model, we used default hyperparameters provided by the scikit-multiflow library. Unlike the proposed T-DFNN model, the Hoeffding tree model only used the training data from the current batch, which is similar to the DFNN-batch model. The comparison with this well-established algorithm can help us understand the advantages and disadvantages of the proposed T-DFNN model.

We used precision, recall, and F1-score as the classification metrics to compare the performance of the models. We applied these metrics to each evaluation class. Using these metrics, we could measure the classification performance of the models for each class. To measure the performance of the models in each evaluation batch, we calculated the macro and weighted average of the precision, recall, and F1-score. Equations 3 and 4 show the formula to calculate the macro average (*MA*) and weighted average (*WA*) metrics, respectively.

$$MA_m = \frac{1}{C}\sum_{i=1}^{C} m_i \tag{3}$$

$$WA_m = \sum_{i=1}^{C} \frac{n_i}{N} \times m_i \tag{4}$$

where

- m is the classification metric, which is *precision*, *recall*, or $F1$-*score*.
- m_i is the value of classification metric m of class i.
- C is the number of classes.
- n_i is the number of data of class i.
- N is the number of data of all classes.

Finally, we ran the experiment ten times and then calculated the average value of all metrics we previously mentioned.

The specification of the computer we used in this experiment is as follows:

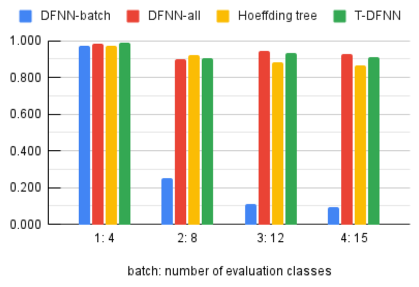
- CPU: Intel(R) Xeon(R) Gold 5122 CPU @ 3.60 GHz
- GPU: NVIDIA TITAN V 12 GB
- RAM: 196 GB
- OS: Ubuntu 16.04.6 LTS

V. EXPERIMENTAL RESULTS

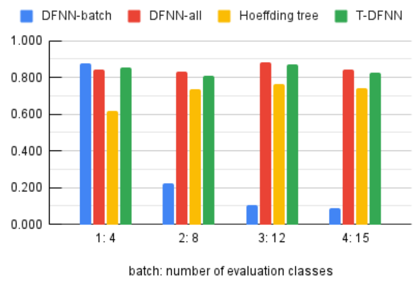
Figure 4 shows the average values of our ten experimental trials. It compares two key aspects of our experiment. First, it compares the evaluation metrics, i.e., precision, recall, and F1-score, of the proposed T-DFNN model to DFNN-batch and DFNN-all models. Second, it compares the macro and weighted averages of the evaluation metrics of each model. These comparisons show how well the models classify the data. These comparisons also help us understand the factors that affect the T-DFNN model in classifying the data.

In Figure 4, the x-axis represents the batch order. This batch order also correlates to the number of evaluation classes used in each batch. In Table 1, we can count that the numbers of training classes in batches 1, 2, 3, and 4 are 4, 4, 4, and 3, respectively. However, in Figure 4, the numbers of evaluation classes in batches 1, 2, 3, and 4 are 4, 8, 12, and 15, respectively. The number of evaluation classes increases in every batch because it also contains the old batch's evaluation data. Thus, in the last batch, all evaluation data were used. We included the classes from the old batch in these evaluation data to simulate the incremental learning process. As we described in Section I, the model for IDSs should be able to classify both old and new intrusion classes.

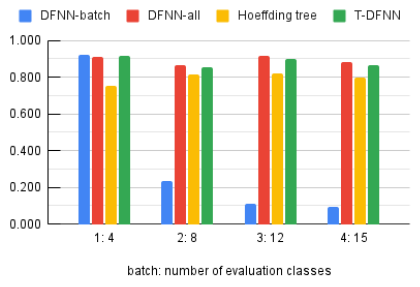
The catastrophic forgetting problem on the DFNN-batch model became apparent after the retraining process. The macro average of the DFNN-batch model's F1-score in Figure 4 declined sharply in each retraining process. As can be seen in Table 2, most of the F1-scores of the DFNN-batch model are below 0.25 after the retraining process. The model misclassified many new classes as new classes or vice versa.



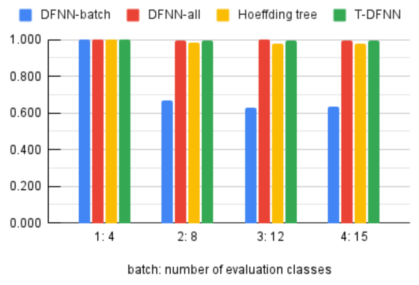
(a) The macro average of precision.



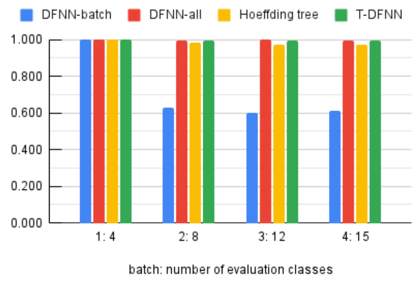
(b) The macro average of recall.



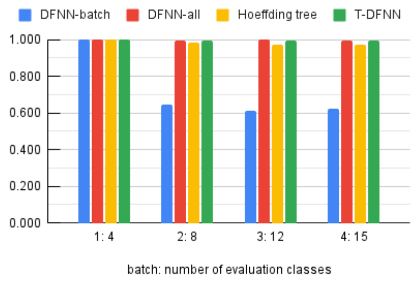
(c) The macro average of F1-score.



(d) The weighted average of precision.



(e) The weighted average of recall.



(f) The weighted average of F1-score.

FIGURE 4. Experimental results.

The only class that had an F1-score above 0.8 after the retraining process was the benign class. However, we should note that the benign class is the largest in the dataset. 80% of the data in CICIDS2017 is benign class. Thus, even though many benign class data were misclassified, it did not severely affect its F1-score compared to the other minor classes.

The most straightforward approach to avoid a catastrophic forgetting problem is retraining the model using the dataset that contains all targeted classes. We tested this approach in our experiment using the DFNN-all model. All three evaluation metrics of the DFNN-all model in Figure 4 show consistent results. For all batches, the macro and weighted average of precision, recall, and F1-score of the DFNN-all model are above 0.8. However, this approach has a severe drawback. As shown in Table 3, most of the training times of the DFNN-all model are the longest compared to the other approaches. The training time of the DFNN-all model increases along with the number of used training data.

The Hoeffding tree model reduced the catastrophic forgetting problem in DFNN batches, and the long training time in DFNN-all models was quite good. We can see in Figure 4 that the Hoeffding tree model is far less affected by the catastrophic forgetting problem compared to the DFNN-batch model. Additionally, the training processes in the Hoeffding tree model were shorter than those in the DFNN-batch and DFNN-all models because it only used the latest training data in its training process.

The experimental results in Table 3 show that the training times of the Hoeffding tree model are shorter than those of the DFNN-batch and DFNN-all models with a reasonably good macro average of F1-scores. However, if we look closely at the evaluation metrics of the Hoeffding tree model in Table 2, we can see that the Hoeffding tree model could not classify several minority classes correctly. The recall values of the Hoeffding tree model for web attack-XSS and web attack-brute force are below 0.45 in all batches. Moreover, as shown in Table 3, the evaluation times of the Hoeffding tree model increased significantly after each retraining process. When we implement the Hoeffding tree model in the network intrusion detection field, these issues become concerning because, in reality, some critical attacks may not have many samples to be analyzed. Additionally, the model may not be feasible for use for a long period because the classification process may become too slow.

The T-DFNN model solved the catastrophic forgetting problem in DFNN batches and the long training time in DFNN-all models. It also had faster classification processes than the Hoeffding tree model without compromising the classification performance. We can see in Figure 4 that the T-DFNN model had better F1-scores than the Hoeffding tree model. Additionally, as we can see in Table 3, the T-DFNN model had faster classification times than the Hoeffding tree model in all baches.

The T-DFNN training algorithm does not use entire old training data in its training process. Instead, it selects the training data based on the output labels of each node's parent node. The training data used in each node are the training data classified as the same parent node's output label. Splitting the training data and distributing them into several nodes speeds up the training process because it reduces the quantity of

TABLE 2. The classification results of evaluation classes in each batch.

Batch	Evaluation classes	Average precision				Average recall				Average F1-score			
		DFNN-batch	DFNN-all	Hoeffding tree	T-DFNN	DFNN-batch	DFNN-all	Hoeffding tree	T-DFNN	DFNN-batch	DFNN-all	Hoeffding tree	T-DFNN
1	Benign	1.000	0.999	0.999	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
	FTP-patator	1.000	1.000	0.964	1.000	0.998	0.998	0.993	0.998	0.999	0.999	0.978	0.999
	Bot	0.911	0.958	0.945	0.962	0.537	0.458	0.436	0.458	0.676	0.620	0.597	0.621
	Web attack-XSS	0.980	0.986	0.971	0.984	0.962	0.908	0.037	0.958	0.971	0.946	0.071	0.971
	macro avg	0.973	0.986	0.970	0.986	0.875	0.841	0.617	0.854	0.921	0.908	0.754	0.915
	weighted avg	1.000	0.999	0.999	0.999	1.000	0.999	0.999	0.999	1.000	0.999	0.999	0.999
2	Benign	0.774	0.999	0.999	0.999	0.751	0.997	0.980	0.998	0.762	0.998	0.989	0.998
	FTP-patator	0.571	1.000	0.964	1.000	0.293	0.999	0.993	0.998	0.387	0.999	0.978	0.999
	Bot	0.001	0.843	0.944	0.962	0.158	0.587	0.435	0.458	0.001	0.692	0.595	0.621
	Web attack-XSS	0.101	0.681	0.900	0.584	0.015	0.105	0.033	0.065	0.026	0.183	0.064	0.118
	DoS-hulk	0.286	0.988	0.967	0.985	0.067	0.996	0.999	0.995	0.109	0.992	0.983	0.990
	Port scan	0.143	0.984	0.991	0.994	0.159	1.000	0.998	0.999	0.151	0.992	0.994	0.997
	DDoS	0.091	1.000	0.803	0.999	0.038	0.999	0.997	0.999	0.054	0.999	0.890	0.999
	Web attack-brute force	0.022	0.708	0.801	0.705	0.305	0.954	0.440	0.973	0.040	0.813	0.568	0.818
	macro avg	0.249	0.900	0.921	0.904	0.223	0.830	0.734	0.811	0.235	0.864	0.817	0.855
	weighted avg	0.665	0.997	0.986	0.997	0.626	0.997	0.982	0.997	0.645	0.997	0.984	0.997
3	Benign	0.758	0.999	0.991	0.999	0.741	0.998	0.975	0.998	0.749	0.999	0.983	0.998
	FTP-patator	0.044	1.000	0.996	0.998	0.230	0.998	0.926	0.997	0.073	0.999	0.960	0.998
	Bot	0.001	0.910	0.944	0.962	0.104	0.590	0.435	0.458	0.001	0.715	0.595	0.621
	Web attack-XSS	0.023	0.801	0.856	0.582	0.009	0.069	0.033	0.064	0.013	0.127	0.064	0.115
	DoS-hulk	0.173	0.990	0.969	0.985	0.024	0.995	0.927	0.995	0.043	0.993	0.948	0.990
	Port scan	0.073	0.994	0.990	0.994	0.020	1.000	0.997	0.999	0.031	0.997	0.994	0.997
	DDoS	0.006	1.000	0.802	0.999	0.001	1.000	0.997	0.999	0.002	1.000	0.889	0.999
	Web attack-brute force	0.002	0.709	0.816	0.706	0.025	0.997	0.090	0.963	0.004	0.828	0.162	0.814
	DoS-golden eye	0.155	0.994	0.896	0.996	0.022	0.993	0.987	0.991	0.039	0.993	0.940	0.994
	SSH-patator	0.080	0.986	0.849	0.977	0.103	0.990	0.896	0.980	0.090	0.988	0.872	0.978
	DoS-slowloris	0.006	0.992	0.734	0.990	0.005	0.993	0.903	0.993	0.005	0.993	0.810	0.991
	DoS-slowhttptest	0.000	0.977	0.772	0.979	0.000	0.992	0.967	0.992	0.000	0.984	0.858	0.985
	macro avg	0.110	0.946	0.884	0.931	0.107	0.884	0.761	0.869	0.108	0.914	0.818	0.899
	weighted avg	0.628	0.998	0.978	0.997	0.600	0.998	0.972	0.997	0.613	0.998	0.975	0.997
4	Benign	0.764	0.999	0.989	0.999	0.756	0.998	0.975	0.998	0.760	0.999	0.982	0.998
	FTP-patator	0.174	0.999	0.996	0.998	0.294	0.999	0.926	0.997	0.219	0.999	0.960	0.998
	Bot	0.001	0.936	0.944	0.962	0.164	0.518	0.435	0.458	0.003	0.667	0.595	0.621
	Web attack-XSS	0.008	0.759	0.856	0.582	0.004	0.065	0.033	0.064	0.005	0.119	0.064	0.115
	DoS-hulk	0.165	0.990	0.969	0.985	0.036	0.994	0.927	0.995	0.059	0.992	0.948	0.990
	Port scan	0.102	0.994	0.990	0.994	0.017	1.000	0.997	0.999	0.029	0.997	0.994	0.997
	DDoS	0.003	1.000	0.799	0.999	0.002	1.000	0.964	0.999	0.002	1.000	0.874	0.999
	Web attack-brute force	0.002	0.706	0.826	0.705	0.005	0.985	0.090	0.963	0.003	0.823	0.162	0.814
	DoS-golden eye	0.177	0.991	0.896	0.996	0.021	0.993	0.982	0.991	0.038	0.992	0.937	0.994
	SSH-patator	0.004	0.984	0.849	0.977	0.022	0.988	0.896	0.980	0.007	0.986	0.872	0.978
	DoS-slowloris	0.009	0.991	0.734	0.990	0.001	0.991	0.903	0.993	0.002	0.991	0.810	0.991
	DoS-slowhttptest	0.000	0.977	0.772	0.979	0.000	0.992	0.967	0.992	0.000	0.984	0.858	0.985
	Infiltration	0.000	0.696	1.000	0.805	0.000	0.671	0.529	0.700	0.000	0.684	0.692	0.749
	Web attack-SQL injection	0.000	0.860	0.450	0.725	0.025	0.475	0.725	0.275	0.000	0.612	0.555	0.399
	Heartbleed	0.000	1.000	0.900	0.967	0.000	1.000	0.800	1.000	0.000	1.000	0.847	0.983
	macro avg	0.094	0.926	0.865	0.911	0.090	0.845	0.743	0.827	0.092	0.883	0.799	0.867
	weighted avg	0.634	0.998	0.977	0.997	0.612	0.998	0.970	0.997	0.623	0.998	0.974	0.997

training data processed in each node. As shown in Table 3, the total quantity of training data used by the T-DFNN model is less compared to the DFNN-all model, while the training times of the T-DFNN model are shorter than the DFNN-all model.

The number of new nodes in each batch of the T-DFNN training process is dynamic. In our study, we ran the experiment ten times. The average numbers of new nodes in batches 1, 2, 3, and 4 of these experiments were 1, 2.6, 7.6, and 4.8, respectively. The number of new nodes generated in each batch depends on the new training data classification result in the previously trained node. Thus, the numbers of new nodes in each batch of the training processes in these experiments were different.

In Figure 5, we visualize a map of the output labels of the first experiment we conducted. For simplification, we use the encoded version of class labels in this visualization. The conversion from the original to the encoded version of class labels is listed in Table 1. We list the number of used training data and the training time of each node in this first experiment in Table 4. In Figure 5 and Table 4, we can observe how the training data were split and trained in several nodes.

TABLE 3. The numbers of used data and processing times in each batch.

Process	Batch	Number of classes	Average number of used data: DFNN-batch	DFNN-all	Hoeffding tree	T-DFNN	Average duration (second): DFNN-batch	DFNN-all	Hoeffding tree	T-DFNN
Training	1	4	1,826,922.0	1,826,922.0	1,826,922.0	1,826,922.0	1,634.72	1,339.07	527.80	1,386.16
	2	8	415,629.0	2,242,551.0	415,629.0	2,236,129.7	313.94	1,962.86	124.61	1,611.91
	3	12	21,988.0	2,264,539.0	21,988.0	2,242,195.2	37.31	2,704.96	16.22	2,388.26
	4	15	55.0	2,264,594.0	55.0	1,935,994.8	5.90	2,309.37	0.05	1,356.44
Evaluation	1	4	456,731.0	456,731.0	456,731.0	456,731.0	17.82	17.31	67.54	17.25
	2	8	560,639.0	560,639.0	560,639.0	560,639.0	24.13	20.16	108.01	42.02
	3	12	566,136.0	566,136.0	566,136.0	566,136.0	26.50	21.20	132.28	68.66
	4	15	566,149.0	566,149.0	566,149.0	566,149.0	28.11	21.56	140.20	84.41

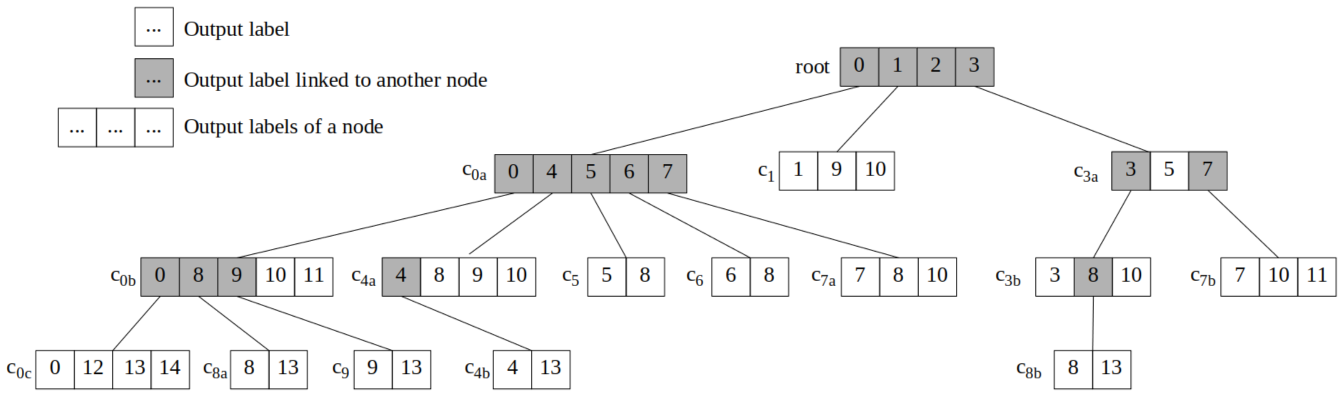


FIGURE 5. The visualization of the map of output labels in the T-DFNN model first experiment.

Additionally, we can observe the relationship between the number of training data points used and the training time of each node. Even though the map of the output labels and the numbers of nodes in our ten experiments were different, they showed a similar pattern, in which the training time tends to increase along with the number of training data points in each node.

The training process of each node in the T-DFNN model was independent. Thus, the training processes were run in parallel. The training time of each batch equals the longest training time of a node in each batch. For example, the training time of batch 4 of our first experiment shown in Table 4 is 1,367.35 seconds because it is the longest training time in batch 4.

VI. DISCUSSION

The experimental results have shown that the T-DFNN algorithm has the potential to be used for incremental learning. However, we found several factors that affect the performance of the T-DFNN algorithm shown by the evaluation metrics. These factors are the class similarity problem, scarcity of the data, and computational overhead.

From the experimental results, we noticed that the class similarity problem and the scarcity of the data could affect the precision, recall, and F1-score of all tested models. We can observe this problem in the classification results of web attack-brute force, web attack-XSS, and web attack-SQL injection classes in Table 2. Those classes are similar types of intrusion. However, as we can see in Table 1, the quantity of data of those classes is severely imbalanced. Some

TABLE 4. The numbers of used training data and training times in each node of the T-DFNN model first experiment.

Batch	Node ID	Encoded output labels	Number of used training data	Training time (second)
1	$root$	0, 1, 2, 3	1,826,922	2,340.65
2	c_{0a}	0, 4, 5, 6, 7	2,233,123	847.84
2	c_{3a}	3, 5, 7	1,488	6.83
3	c_{0b}	0, 8, 9, 10, 11	1,836,206	3,042.64
3	c_1	1, 9, 10	6,730	48.21
3	c_{7a}	7, 8, 10	189	21.87
3	c_{7b}	7, 10, 11	965	22.45
3	c_{4a}	4, 8, 9, 10	183,238	187.35
3	c_5	5, 8	127,112	350.51
3	c_6	6, 8	102,770	231.35
3	c_{3b}	3, 8, 10	67	21.68
4	c_{0c}	0, 12, 13, 14	1,815,410	1,367.35
4	c_{4b}	4, 13	182,896	62.43
4	c_{8a}	8, 13	7,335	32.67
4	c_{8b}	8, 13	34	7.40
4	c_9	9, 13	4,672	12.02

of them also have a limited quantity of data. In the first batch classification result, the F1-score of web attack-XSS using T-DFNN models was relatively high, 0.971. However, in the second batch classification result, the F1-score value dropped to 0.118 because many data were misclassified as web attack-brute force class, which has a larger number of data. The same problem also occurred in the classification result of the web attack-SQL injection class. The model falsely classified the data of minority classes as other majority classes of the same intrusion type.

The classification results of the heartbleed and infiltration classes in Table 2 show an interesting result. These classes have scarce quantities of data. Heartbleed and infiltration data are only 0.00039% and 0.00127% of the total data, respectively. However, the T-DFNN model's precision, recall, and F1-score of these classes were quite decent. The DFNN-all and Hoeffding tree models also showed similar results. The reason those models can classify these classes correctly is the characteristic of the intrusions. Heartbleed attacks using the Transport Layer Security (TLS) protocol through a security bug in the OpenSSL cryptography library. Infiltration intrusion scans victims from the internal network of infected clients [13]. Both have no other similar type of intrusions in the dataset. These results suggest that data scarcity does not always contribute to the F1-score reduction in the model. Instead, the characteristics of the data have more influence on the F1-score of the model.

Another factor that affects the precision, recall, and F1-score of the proposed T-DFNN model shown by the evaluation metrics was its additional computational overhead. Because the T-DFNN model consists of several nodes in a tree-like structure, the data might need to be classified using several nodes before obtaining the final classification result. This process created a computational overhead that can be observed in the classification time of the T-DFNN model in Table 3. The T-DFNN model classification times were longer in every batch because the height of the tree structure in the T-DFNN model increased.

We can estimate the classification time of the T-DFNN model using Equation 1. For example, we can estimate the classification times of each batch in Table 5 using Equation 1. The estimation results show that the classification times of batches 1, 2, 3, and 4 are 16.425, 41.118, 71.702, and 80.869, respectively. These estimation results are close to the actual classification times from the experimental results presented in Table 6. These classification times indicate that the computational overhead caused by the growth of the T-DFNN model's tree structure always increases after each training.

The computational overhead caused by the tree structure of the T-DFNN model did not occur in the DFNN-batch or DFNN-all model. Thus, the classification time of the DFNN-batch and DFNN-all models did not increase significantly. However, we should note that the computational overhead caused by the growth of the tree structure also occurred in the Hoeffding tree model. The T-DFNN algorithm manages to minimize this computational overhead.

TABLE 5. The number of evaluation data and classification times in each node of the T-DFNN model first experiment.

Batch	Node ID	Node level	Number of evaluation data	Classification time (second)
1	$root$	1	456,731	16.425
2	$root$	1	560,639	21.042
2	c_{0a}	2	558,489	20.076
2	c_{3a}	2	380	0.145
3	$root$	1	566,136	26.568
3	c_{0a}	2	563,878	21.768
3	c_{3a}	2	387	0.125
3	c_1	2	1,688	0.220
3	c_{0b}	3	459,801	23.366
3	c_{3b}	3	24	0.121
3	c_{4a}	3	46,368	4.722
3	c_5	3	31,973	3.782
3	c_6	3	25,695	3.184
3	c_{7a}	3	41	0.064
3	c_{7b}	3	363	0.388
4	$root$	1	566,149	20.335
4	c_{0a}	2	563,891	20.063
4	c_{3a}	2	387	0.120
4	c_1	2	1,688	0.213
4	c_{0b}	3	459,814	20.218
4	c_{3b}	3	24	0.135
4	c_{4a}	3	46,368	4.713
4	c_5	3	31,973	3.815
4	c_6	3	25,695	3.325
4	c_{7a}	3	41	0.064
4	c_{7b}	3	363	0.136
4	c_{0c}	4	454,595	20.253
4	c_{4b}	4	46,296	2.754
4	c_{8a}	4	7	0.047
4	c_{8b}	4	1,853	0.232
4	c_9	4	1,191	0.197

TABLE 6. The numbers of used evaluation data and total classification times in each batch of the T-DFNN model first experiment.

Batch	Number of evaluation data	Classification time (second)
1	456,731	16.438
2	560,639	41.144
3	566,136	71.734
4	566,149	80.912

As shown in Table 3, the evaluation times of the T-DFNN model are at least 39% shorter than those of the Hoeffding tree model.

Despite all challenging factors affecting the T-DFNN model, we should note that the T-DFNN model has several advantages. These advantages are that it can reduce the catastrophic forgetting effect and shorten the training time. We can observe these advantages in Tables 2 and 3. The T-DFNN model was less affected by the catastrophic forgetting problem. For each class in Table 2, the T-DFNN model can classify the evaluation data in every batch. Even though the precision, recall, or F1-score of the T-DFNN model was slightly lower than that of the DFNN-all model in some classes, the training time of the T-DFNN model was much shorter than that of the DFNN-all model. Additionally, the classification times of the T-DFNN model are shorter than those of the Hoeffding tree model.

VII. CONCLUSION

Incremental learning in IDSs is a challenging problem. The main problems facing incremental learning are the ever-evolving network intrusion variants and catastrophic forgetting problems. We solved both problems by proposing the T-DFNN algorithm, which combines a tree data structure and DFNN models. The experimental results showed that the model produced by the proposed T-DFNN algorithm can learn and classify the network intrusions incrementally without being severely affected by the catastrophic forgetting effect. Moreover, the T-DFNN algorithm can shorten the training time. However, the T-DFNN algorithm requires more computational steps that increase the classification time.

Other factors that affected the precision, recall, and F1-score of the model are the similarity between classes and the scarcity of the data. These factors affected not only the T-DFNN model but also other models in general. Therefore, we suggest more comprehensive research on these factors as future work to improve the performance of the T-DFNN algorithm.

REFERENCES

[1] A. Shenfield, D. Day, and A. Ayesh, "Intelligent intrusion detection systems using artificial neural networks," *ICT Exp.*, vol. 4, no. 2, pp. 95–99, Jun. 2018. [Online]. Available: https://doi.org/10.1016/j.icte.2018.04.003

[2] P. Mishra, E. S. Pilli, V. Varadharajan, and U. Tupakula, "Intrusion detection techniques in cloud environment: A survey," *J. Netw. Comput. Appl.*, vol. 77, pp. 18–47, Jan. 2017. [Online]. Available: https://doi.org/10.1016/j.jnca.2016.10.015

[3] N. S. Chandolikar and V. D. Nandavadekar, "Efficient algorithm for intrusion attack classification by analyzing KDD cup 99," in *Proc. 9th Int. Conf. Wireless Opt. Commun. Netw. (WOCN)*, Sep. 2012, pp. 1–5, doi: 10.1109/WOCN.2012.6335546.

[4] E. Hodo, X. Bellekens, A. Hamilton, C. Tachtatzis, and R. Atkinson, "Shallow and deep networks intrusion detection system: A taxonomy and survey," Tech. Rep., 2017. [Online]. Available: https://strathprints.strath.ac.U.K./63256/

[5] A. Pektas and T. Acarman, "A deep learning method to detect network intrusion through flow-based features," *Int. J. Netw. Manage.*, vol. 29, no. 3, 2019, Art. no. e02050. [Online]. Available: https://doi.org/10.1002/nem.2050

[6] A. Khraisat, I. Gondal, P. Vamplew, and J. Kamruzzaman, "Survey of intrusion detection systems: Techniques, datasets and challenges," *Cybersecurity*, vol. 2, no. 1, pp. 1–22, Dec. 2019, doi: 10.1186/s42400-019-0038-7.

[7] C. Modi, D. Patel, B. Borisaniya, H. Patel, A. Patel, and M. Rajarajan, "A survey of intrusion detection techniques in cloud," *J. Netw. Comput. Appl.*, vol. 36, no. 1, pp. 42–57, 2013. [Online]. Available: https://doi.org/10.1016/j.jnca.2012.05.003

[8] T. Daniya, K. S. Kumar, B. S. Kumar, and C. S. Kolli, "A survey on anomaly based intrusion detection system," *Mater. Today: Proc.*, pp. 1–4, Apr. 2021, doi: 10.1016/j.matpr.2021.03.353.

[9] H. Du, S. Teng, M. Yang, and Q. Zhu, "Intrusion detection system based on improved SVM incremental learning," in *Proc. Int. Conf. Artif. Intell. Comput. Intell.*, 2009, pp. 23–28, doi: 10.1109/AICI.2009.254.

[10] M. Mermillod, A. Bugaiska, and P. Bonin, "The stability-plasticity dilemma: Investigating the continuum from catastrophic forgetting to age-limited learning effects," *Frontiers Psychol.*, vol. 4, p. 504, Aug. 2013. [Online]. Available: https://doi.org/10.3389/fpsyg.2013.00504

[11] K. James, R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska, D. Hassabis, C. Clopath, D. Kumaran, and R. Hadsell, "Overcoming catastrophic forgetting in neural networks," *Proc. Nat. Acad. Sci. USA*, vol. 114, no. 13, pp. 3521–3526, Mar. 2017. [Online]. Available: https://doi.org/10.1073/pnas.1611835114

[12] I. J. Goodfellow, M. Mirza, D. Xiao, A. Courville, and Y. Bengio, "An empirical investigation of catastrophic forgetting in gradient-based neural networks," 2013, *arXiv:1312.6211*.

[13] I. Sharafaldin, A. H. Lashkari, and A. A. Ghorbani, "Toward generating a new intrusion detection dataset and intrusion traffic characterization," in *Proc. 4th Int. Conf. Inf. Syst. Secur. Privacy*, vol. 1, 2018, pp. 108–116. [Online]. Available: https://doi.org/10.5220/0006639801080116

[14] R. Polikar, L. Upda, S. S. Upda, and V. Honavar, "Learn++: An incremental learning algorithm for supervised neural networks," *IEEE Trans. Syst., Man, Cybern., C (Appl. Rev.)*, vol. 31, no. 4, pp. 497–508, Nov. 2001, doi: 10.1109/5326.983933.

[15] H. Hindy, D. Brosset, E. Bayne, A. Seeam, C. Tachtatzis, R. Atkinson, and X. Bellekens, "A taxonomy of network threats and the effect of current datasets on intrusion detection systems," *IEEE Access*, vol. 8, pp. 104650–104675, 2020, doi: 10.1109/ACCESS.2020.3000179.

[16] C. Constantinides, S. Shiaeles, B. Ghita, and N. Kolokotronis, "A novel online incremental learning intrusion prevention system," in *Proc. 10th IFIP Int. Conf. New Technol., Mobility Secur. (NTMS)*, Jun. 2019, pp. 1–6, doi: 10.1109/NTMS.2019.8763842.

[17] M.-H. Chen, P.-C. Chang, and J.-L. Wu, "A population-based incremental learning approach with artificial immune system for network intrusion detection," *Eng. Appl. Artif. Intell.*, vol. 51, pp. 171–181, Jan. 2016. [Online]. Available: https://doi.org/10.1016/j.engappai.2016.01.020

[18] Y. Yi, J. Wu, and W. Xu, "Incremental SVM based on reserved set for network intrusion detection," *Expert Syst. Appl.*, vol. 38, no. 6, pp. 7698–7707, Jun. 2011, doi: 10.1016/j.eswa.2010.12.141.

[19] B. Xu, S. Chen, H. Zhang, and T. Wu, "Incremental K-NN SVM method in intrusion detection," in *Proc. 8th IEEE Int. Conf. Softw. Eng. Service Sci. (ICSESS)*, Nov. 2017, pp. 712–717, doi: 10.1109/ICSESS.2017.8343013.

[20] F. Jiang, Y. Sui, and C. Cao, "An incremental decision tree algorithm based on rough sets and its application in intrusion detection," *Artif. Intell. Rev.*, vol. 40, no. 4, pp. 517–530, Dec. 2013, doi: 10.1007/s10462-011-9293-z.

[21] H. Yu, J. Yang, and J. Han, "Classifying large data sets using SVMs with hierarchical clusters," in *Proc. 9th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining (KDD)*, 2003, pp. 306–315, doi: 10.1145/956750.956786.

[22] KDD99. (1999). *KDD Cup 1999 Data*. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/KDD+Cup+1999+Data

[23] M. Tavallaee, E. Bagheri, W. Lu, and A. A. Ghorbani, "A detailed analysis of the KDD CUP 99 data set," in *Proc. IEEE Symp. Comput. Intell. Secur. Defense Appl.*, Jul. 2009, pp. 1–6, doi: 10.1109/CISDA.2009.5356528.

[24] A. Fonseca and B. Cabral, "Prototyping a GPGPU neural network for deep-learning big data analysis," *Big Data Res.*, vol. 8, pp. 50–56, Jul. 2017. [Online]. Available: https://doi.org/10.1016/j.bdr.2017.01.005

[25] S. Lu, X. Wei, Y. Li, and L. Wang, "Detecting anomaly in big data system logs using convolutional neural network," in *Proc. IEEE 16th Int. Conf. Dependable, Autonomic Secure Comput., 16th Int. Conf. Pervasive Intell. Comput., 4th Int. Conf. Big Data Intell. Comput. Cyber Sci. Technol. Congr. (DASC/PiCom/DataCom/CyberSciTech)*, Aug. 2018, pp. 151–158, doi: 10.1109/DASC/PiCom/DataCom/CyberSciTec.2018.00037.

[26] B. Ait Hammou, A. Ait Lahcen, and S. Mouline, "Towards a real-time processing framework based on improved distributed recurrent neural network variants with fastText for social big data analytics," *Inf. Process. Manage.*, vol. 57, no. 1, Jan. 2020, Art. no. 102122, doi: 10.1016/j.ipm.2019.102122.

[27] D. Roy, P. Panda, and K. Roy, "Tree-CNN: A hierarchical deep convolutional neural network for incremental learning," *Neural Netw.*, vol. 121, pp. 148–160, Apr. 2020. [Online]. Available: https://doi.org/10.1016/j.neunet.2019.09.010

[28] S. S. Sarwar, A. Ankit, and K. Roy, "Incremental learning in deep convolutional neural networks using partial network sharing," *IEEE Access*, vol. 8, pp. 4615–4628, 2020, doi: 10.1109/ACCESS.2019.2963056.

[29] G. Hulten, L. Spencer, and P. Domingos, "Mining time-changing data streams," in *Proc. 7th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining (KDD)*, 2001, pp. 97–106, doi: 10.1145/502512.502529.

[30] J. Montiel, J. Read, A. Bifet, and T. Abdessalem, "Scikit-multiflow: A multi-output streaming framework," *J. Mach. Learn. Res.*, vol. 19, no. 72, pp. 1–5, 2018. [Online]. Available: http://jmlr.org/papers/v19/18-251.html

[31] I. H. Witten, E. Frank, and M. A. Hall, "The WEKA workbench," in *Online Appendix for 'Data Mining: Practical Machine Learning Tools and Techniques'*. San Mateo, CA, USA: Morgan Kaufmann, 2016. [Online]. Available: https://www.cs.waikato.ac.nz/ml/weka/Witten_et_al_2016_appendix.pd f

[32] A. Gharib, I. Sharafaldin, A. H. Lashkari, and A. A. Ghorbani, "An evaluation framework for intrusion detection dataset," in *Proc. Int. Conf. Inf. Sci. Secur. (ICISS)*, Dec. 2016, pp. 1–6, doi: 10.1109/ICISSEC.2016.7885840.

[33] R. Panigrahi and S. Borah, "A detailed analysis of CICIDS2017 dataset for designing intrusion detection systems," *Int. J. Eng. Technol.*, vol. 7, no. 3, pp. 479–482, 2018. [Online]. Available: https://www.sciencepubco.com/index.php/ijet/article/view/22797

[34] A. Habibi Lashkari, G. Draper Gil, M. S. I. Mamun, and A. A. Ghorbani, "Characterization of tor traffic using time based features," in *Proc. 3rd Int. Conf. Inf. Syst. Secur. Privacy*, vol. 1, 2017, pp. 253–262, doi: 10.5220/0006105602530262.

[35] R. Abdulhammed, H. Musafer, A. Alessa, M. Faezipour, and A. Abuzneid, "Features dimensionality reduction approaches for machine learning based network intrusion detection," *Electronics*, vol. 8, no. 3, p. 322, 2019. [Online]. Available: https://doi.org/10.3390/electronics8030322

[36] A. Bifet, G. Holmes, R. Kirkby, and B. Pfahringer, "MOA: Massive online analysis," *J. Mach. Learn. Res.*, vol. 11, no. 52, pp. 1601–1604, May 2010. [Online]. Available: http://jmlr.org/papers/v11/bifet10a.html

MAHENDRA DATA received the B.S. degree in computer science from Brawijaya University, Indonesia, in 2010, and the M.S. degree in computer science from the Sepuluh Nopember Institute of Technology, Indonesia, in 2014. He is currently pursuing the Ph.D. degree with the Graduate School of Science and Technology, Kumamoto University, Japan.

In 2011, he started his career in the educational field as a Lecturer at Brawijaya University, where he was promoted to an Assistant Professor, in 2015. He is doing research in the field of network intrusion detection systems. Specifically, he is focusing his research on the utilization of deep learning algorithms to enhance the capabilities of network intrusion detection systems.

MASAYOSHI ARITSUGI (Member, IEEE) received the B.E. and D.E. degrees in computer science and communication engineering from Kyushu University, Japan, in 1991 and 1996, respectively.

From 1996 to 2007, he was with the Department of Computer Science, Gunma University, Japan. Since 2007, he has been a Professor at Kumamoto University, Japan. His research interests include database systems and parallel/distributed data processing.

Prof. Aritsugi is a Senior Member of IPSJ and IEICE and a member of ACM and DBSJ. He was a recipient of the COMPSAC 2015 Best Paper Award, the Best Paper Award in Image Processing and Understanding in 13th IEEE International Conference on Signal Processing (ICSP2016), and the Best Paper Award in the 2019 IEEE International Cyber Science and Technology Congress (CyberSciTech).