Enhancing Intrusion Detection Systems through Advanced Machine Learning Techniques

Rafael González Cartagena
Master in Computer Science, Cybersecurity in Digital Evidence and Auditing
Dr. Alfredo Cruz
Electrical and Computer Engineering Department
Graduate Project EXPO May 2025

Abstract – *The rise in cyberattacks during the Covid – 19 pandemics has intensified the need for intrusion detection systems. This study investigates three machine learning algorithms: Random Forest Classifier, Convolutional Neural Network – Focal Loss, and Convolutional Neural Network – Cross Entropy Loss, under two project development lifecycles. Using the NSL – KDD Dataset, which contains network connection records classified into normal traffic or a multitude of different attacks, each of system is evaluated across multiple data handling staged, including scaling, balancing and feature selection. Results indicate that the Random Forest Classifier is best for binary classification, no matter the project development approach used (F1 Score: 99.92% in both cases); whereas Convolutional Neural Network – Cross Entropy Loss works best for multi – class classification when under the micro – detectors approach (F1 Score: 99.59%). As such, the findings of this research would offer practical guidance for designing machine learning systems for intrusion detection systems.*

Key Terms – *Convolutional Neural Networks, Disposable Micro – Detectors, Intrusion Detection Systems, and Random Forest Classifier.*

INTRODUCTION

The rapid shift to remote work and online digital services triggered by the Covid – 19 pandemics has been accompanied by a dramatic rise in cybercrimes, placing a renewed interest on Intrusion Detection Systems (IDS). Traditional IDS technologies, which tend to mostly rely on rule – based detection, log analysis, and filesystem monitoring, have been seen to struggle with increasingly sophisticated attacks and evasion techniques, some which have started to even use Artificial Intelligence (AI) technology. Yet, AI, in particular Machine Learning (ML), has also been seen to offer a promising approach for the enhancement of the performances of these systems by being able to learn complex patterns of both normal and anomalous network behavior directly from data inputs.

Notwithstanding, key challenges remain. Firstly, the number of cyberattack reports has increased, that is, since the start of the pandemic itself, organizations around the world have experienced a 50% increase in incidents [1], with ransomware attacks being the most common, and averaging $2.75 million in damages per event as of 2025 [2]. Secondly, most of the previous research combining IDSs with AI have only applied to Deep Learning models that used the traditional machine learning lifecycle methodology known as the Monolithic Approach. In particular, there seems to be almost no research on the performances of ML systems with other methodologies, such as that of the micro – detectors approach, in which separate model instances are trained and tested per attack class, so as to see which yields better results [3]. As such, due to these reasons, this study addresses these gaps by evaluating three ML systems, namely: Random Forest Classifier, Convolutional Neural Network – Focal Loss and Convolutional Neural Network – Cross Entropy Loss, under both the monolithic and micro – detectors approaches, using the NSL – KDD Dataset.

Henceforth, for the purpose of guiding the development of this study, the following research questions were posed:

1. What constitutes artificial intelligence, machine learning, and intrusion detection systems?
2. What types of data – centric attributes do the NSL – KDD Dataset offer for IDS development?
3. What are the key architectural and methodological distinctions between the monolithic and micro – detector approaches for ML design?
4. Do ML models developed under the micro – detector approach demonstrate superior performance compared to those which use the monolithic alternative?

5. Based on the findings of prior research, is it correct to assume that deep learning algorithms, like convolutional neural networks, will always outperform traditional methods?

By answering these questions, this study would provide the following: (1) an empirical analysis and comparison of traditional and deep learning machine learning algorithms for IDS; (2) an examination of the two monolithic and micro – detector approaches for machine learning project development, and (3) practical recommendations on how to design and implement ML systems for IDS environments.

RELATED WORK

Past studies, that are based on the usage of AI and ML technologies for IDSs, are comprised of a multitude of fundamental definitions and concepts, dataset information, traditional and deep learning methods, and project lifecycle methodologies. All of which are to be reviewed below.

Fundamental Concepts of Intrusion Detection Systems

As defined by the National Institute of Standards and Technology, Intrusion Detection Systems (IDS) are software tools that monitor events in a computer system or network to identify incidents, i.e.: events in which security policies are violated, such as malware infections or unauthorized access [4]. Traditional IDS methodologies include:

- Signature – Based Detection [4], which compares observed events against a database of known threat patterns. While highly accurate for known cyberattacks, it fails at detecting novel or obfuscated threats.
- Anomaly – Based Detection [4], which builds profiles of normal user and network behavior and flags anomalous deviations. This approach can catch zero – day exploits but often suffers from high false – positive rates.
- Stateful Protocol Analysis [4], which examines deviations from expected network protocols' state transitions.

There are four commonly used IDS technologies, namely: Network – Based IDSs, Wireless IDSs, Network Behavior Analysis (NBA) Systems, and Host – Based IDSs. Network – Based IDSs monitor computer network traffic in real time, analyzing it at the Network, Transport and Application layers to identify suspicious activity [4]. Its core components include:

- Sensors, which can be appliance – based or software – based. Appliance – Based sensors are those that consist of specialized hardware, like specific network interface cards, and software dedicated to packet capture and analysis. On the other hand, Software – Based sensors are those that run on general – purpose hardware, relying on the host's system existing network interfaces for traffic analysis.
- Management Server and Console that would allow sensors to forward alerts and incident information to for administrator inspection and response.
- Signature – Based Detection, in which packet payloads and header patterns are matched against known cyberattack signatures.
- Anomaly – Based Detection, in which statistical profiles of known network behavior are established to flag deviations.
- Stateful Protocol Analysis, in which all the network traffic is verified for expected protocol transitions, like that of the correct TCP handshake sequences.
- Logging and Prevention Functions, like detailed packet logs which contain timestamps and session IDs.

Subsequently, Wireless IDSs focus exclusively on the monitoring and analysis of IEEE 802.11 WIFI network communications [4]. It is characterized by:

- Sensors, which can be dedicated – based, bundled AP – based and bundled wireless switch – based. A Dedicated Sensor is one which is a passive wireless device that is either fixed or mobile. Bundled AP Sensors are those which can monitor channel network traffic while providing network access. And a Bundled Wireless Switch Sensor is one which is comprised of a wireless switch equipped with IDS functionally. Of these, bundled wireless switch sensors are said to provide the weakest detection capabilities due to the resource contention between channel switching and intrusion monitoring.

- Logging and Detection Functions, which include MAC and SSID information gathering, and the detection of unauthorized or poorly secured WLANs, unusual network usage patterns, and DoS attacks.
- Prevention Functions for both wireless and wired communications. In the cases of Wireless and Wired Preventions, sensors can respectively terminate client – to – AP associations for rogue network devices and APs by inserting unauthentication frames, and block traffic based on the MAC addresses or switchports used by malicious STAs.

NBA IDSs are those that inspect computer network flow statistics rather than full packet payloads for the detection of anomalous network traffic patterns indicative of DoS, worm and policy violations attacks [4]. As such, their main characteristics include:

- Sensors that can be flow – based and packet – level based. Flow – Based Sensors are those that collect NetFlow records from routers and switches, summarizing each TCP / UDP session by source and destination IPs, source and destination ports, and protocols, along with bytes and packet counts. Packet – Level Sensors are used to do deeper protocol parsing with packet capture data.
- Logging and Detection Functions, which are comprised of the reconstruction of timeframes and passive fingerprinting, and the identification of DoS, worms, eavesdropping and policy violations attacks, respectively.
- Prevention Functions that include automatic IDS reconfigurations inline firewalling and session teardowns whenever a cyberattack is detected.

And, lastly, Host – Based IDSs are IDSs that operate at host devices, like computers and servers, using locally installed agents or dedicated appliances that monitor system calls, filesystem changes and the behaviors of applications [4]. Their main features are:

- Sensors, called agents, that monitor and report the activity of every host in a network to a central management server via the usage of dedicated appliance hardware.
- Management Server and Console, which are used to combine all IDS events into a human – readable dashboard.
- Logging and Detection Function that include, respectively, the logging of timestamps, event or alert types, application information, file names, directory paths and network interface statistics, and the detection of network incidents via code, network, filesystem monitoring, log and network configuration analyses.
- Prevention Functions that include code execution prevention, network packet filtering, automatic incident response scripts and the protection of critical system files and directories.

Traditional Machine Learning Case Studies for IDS

Early investigations have applied a variety of supervised classification systems to the NSL – KDD Dataset for IDSs applications. In the paper A Detailed Analysis on NSL – KDD Dataset Using Various Machine Learning Techniques for Intrusion Detection, by Revathi & Malathi, various ML models are evaluated, these being: Random Forest Classifier, Support Vector Machine (SVM), Naïve Bayes, J48 and CART [5]. Of these, regarding the accuracy evaluation metric, the Random Forest Classifier performed the best, with average scores of 97.9% and 98.9% when being trained with 41 and 15 features, respectively.

Deep Learning Case Studies for IDS

With the rise of Deep Learning, Convolutional Neural Networks (CNN) have been investigated for IDS purposes. Zhao et al, with their paper titled Application of deep learning – based Intrusion Detection System (IDS) in network traffic detection, reported that CNNs trained with the NSL – KDD Dataset and with both Cross – Entropy Loss and Focal Loss far outperformed those of the traditional methods, specifically: Random Forest Classifier, SVM and Decision Trees [6]. Yet, it should be noted that this study lacks any form of quantitative comparisons or results.

Machine Learning Project Lifecycles Case Studies

As already established, most prior ML – IDS research uses the traditional machine learning project lifecycle known as the monolithic approach, where a single ML system is trained on all classes of the target column, undergoing dataset selection, preprocessing, model selection, hyperparameter tuning and evaluation in a

centralized learning pipeline [3]. By contrast, Saad et al. propose a new alternative approach which they coin as Disposable Micro – Detectors, or Micro – Detectors for short, were specialized detectors or instances of a given ML system are instantiated over an attack class [3]. That is, each detector is trained, evaluated and maintained independently, while, potentially, reducing overall system's costs and adaptability to evermore evolving cyber threats.

DATASET AND EXPERIMENTAL SETUP

The NSL – KDD Dataset

The NSL – KDD Dataset is an enhanced successor to the widely used KDD CUP 1999 Dataset, addressing issues of target column redundant records and skewed distributions [7]. It is comprised of 148,517 records and 43 columns, where 29 are numeric and 14 are categorical. The ATTACK_LABEL column, which is the target or label of the ML systems themselves, is comprised of 40 different network attack classes, including, but not limited to: Neptune, ipsweep, satan, smurf, nmap and mailbomb attacks. Additionally, compared to the KDD CUP 1999, the NSL – KDD Dataset introduces its 43rd column, DIFFICULTY_CLASS, which rates each record on a 1 – 21 scale based on its classification difficulty for ML classification systems. Table 1 illustrates the names and descriptions of each of some of the columns found in the NSL – KDD Dataset [8].

Furthermore, in the case of the NSL – KDD training set, it was discovered that it included duplicated records, 390 to be exact, or a 0.30% of the set itself.

Table 1
NSL – KDD Columns Descriptions

Column Name	Description
DURATION	Length, in seconds, of the network connection.
PROTOCOL_TYPE	Type of network protocol that was used.
SERVICE	Type of network service that was used.
FLAG	Status flag, that is that was used during the network connection. In other words, it signals how the TCP handshake will proceed.
SRC_BYTES	Number of data bytes from source to destination.
DST_BYTES	Number of data bytes from destination to source.
LAND	Indicator of whether the source IP address and source port are equal to the destination IP address and destination port, respectively. A *1* indicates that such values are the same; otherwise, a *0* is used.
WRONG_FRAGMENT	Number of out of order fragments throughout the network connection.
URGENT	Number of urgent packets sent.
HOT	Number of administrative indicators in each packet payload. Examples of indicators include file access, file creation and file deletion commands.
NUM_FAILED_LOGINS	Number of failed login attempts.
LOGGED_IN	Binary indicator of whether a user has logged in with the network connection. A *1* indicates as such, otherwise, *0*.
NUM_COMPROMISED	Number of security incidents.
ROOT_SHELL	Binary of *1* if root access was acquired; otherwise, *0*.
SU_ATTEMPTED	Number of *sudo*, superuser or administrative, command attempts.
NUM_ROOT	Number of successfully executed root – access commands.
NUM_FILE_CREATIONS	Number of created files during a network connection.
NUM_SHELLS	Number of shell attempts undertaken.
NUM_ACCESSED_FILES	Number of accessed files during a network connection.
NUM_CMD_OUTBOUND	Number of outbound commands in an FTP or shell network session.
IS_HOST_LOGGED_IN	Binary of *1* if host is logged in via an account; otherwise, *0*.
IS_GUEST_LOGGED_IN	Binary of *1* if host is logged in via a guest account; otherwise, *0*.
COUNT	Number of network connections to the same host as the current connection in the last two seconds.
SRV_COUNT	Number of connections to the same network service.
SERROR_RATE	Percentage of network connections that have SYN errors.

SRV_SERROR_RATE	Percentage of network connections with SYN errors to a given same service.
RERROR_RATE	Percentage of network connections with REJECTION errors.
SRV_RERROR_RATE	Percentage of network connections with REJECTION errors to a given same service.
SAME_SRV_RATE	Percentage of connections to a given service as the current network connection.
DIFF_SRV_RATE	Percentage of network connections using different services.
SRV_DIFF_HOST_RATE	Percentage of network connections using a given service, but with different hosts.
DST_HOST_COUNT	Number of connections to a given destination host, as the current host.
DST_HOST_SRV_COUNT	Number of connections to the same network service on a destination host.
DST_HOST_SAME_SRV_RATE	Percentage of connections to a destination host with a given same network service.
DST_HOST_DIFF_SRV_RATE	Percentage of connections to a destination host with different network services.
DST_HOST_SERROR_RATE	Percentage of network connections to a destination host with SYN errors.
DST_HOST_SRV_SERROR_RATE	Percentage of connections to a destination host's service with SYN errors.
DST_HOST_RERROR_RATE	Percentage of connections to a destination host with REJECTION errors.
DST_HOST_SRV_RERROR_RATE	Percentage of network connections to a destination host's service with REJECTION errors.
ATTACK_LABEL	Label indicating connection type. "*Normal"* indicates that the connection is secure, while anything else represents the opposite, an *Attack Category*.

Data Handling Process

Before the training of all ML systems, the NSL – KDD Dataset, possibly, underwent through various stages in a data preprocessing pipeline or process. The first of these steps is Basic Preprocessing, which all models underwent. It is comprised of data loading and cleaning, where raw NSL – KDD Dataset CSV files are converted into training and testing Pandas DataFrames sets, and their categorical columns, i.e.: PROTOCOL_TYPE, SERVICE, FLAG and ATTACK_LABEL, converted to one – hot encoded versions using Scikit – Learn's OneHotEncoder function. Additionally, four versions of the ATTACK_LABEL label were created:

- One in which all categories were simplified into Normal (0) and Anomalous (1) classes.
- One in which all categories were first simplified into the previously described ones and the encoded via one – hod encoding.
- One in which all categories were simplified into five different classes, i.e.: Normal (0), DoS (1), U2R (2), R2L (3) and Probe (4). [5]
- One in which all categories were first simplified into the previously five described ones and then encoded via one – hot encoding.

The second of these steps, which was progressively used, is Feature Scaling, which scales all features of both the training and testing sets using Scikit – Learn's StandardScaler function [9]. It should be noted that StandardScaler scales the values of datasets using the Standard Deviation, defined at (1).

$$\sigma = \sqrt{\frac{\sum(x_i - \bar{x})^2}{N}} \quad (1)$$

Where N is the total number of data points, x_i is each individual data point, and $\bar{x}$ is the mean.

Thirdly, Balancing refers to the weighing of records with respect to the respective distributions of the different versions of the ATTACK_LABEL label. For this, the technique known as Synthetic Minority Over – Sampling Technique (SMOTE) [10] was used on the training set only. With SMOTE, the sample sizes of the minority classes until the overall distribution of all classes is approximately uniform.

The last of the data handling processes that was used is Feature Selection, which, in the case of this experiment, selects the most important 85% of features for a given ML system using a Mutual Information Classifier, a L1 – Regularized Linear Regression (LASSO) and then, a Random Forest

Classifier. The equation for the mutual information classifier is shown in (2), while equation (3) is for l1 – regularized linear regression.

$$I(X;Y) = \sum_{x \in X} \sum_{y \in Y} p(x,y) \log \left(\frac{p(x,y)}{p(x)p(y)}\right) \quad (2)$$

$$F_{objective} = (B_0 + B_1 x_1 + B_2 x_2 + \ldots + B_n x_n + \varepsilon) + \lambda \times (|B_1| + |B_2| + \ldots + |B_n|) \quad (3)$$

Where $p(x,y)$ is the joint probability of x and y, $p(x)$ and $p(y)$ are the marginal probabilities of x and y, respectively; B_i are the coefficients to be estimated; x_i are the features; ε is an error term; and λ is LASSO's regularization hyperparameter [11] - [12].

Monolithic vs. Micro – Detectors Designs

The monolithic and micro – detectors machine learning project methodologies are distinctively implemented to train maintain the ML systems using them. The monolithic lifecycle, where a single instance of a model is trained to classify all target labels simultaneously, includes the following steps [13]:

1. Dataset Selection, in which the NSL – KDD Dataset is identified and selected for the study in question.
2. Dataset Visualization, where the trends in the data found in the NSL – KDD Dataset are studied.
3. Data Preparation, where the data handling steps discussed in the Data Handling Process section are undertaken.
4. Model Selection, which is comprised of the selection of the current model to be analyzed.
5. Mode Fine – Tuning, in which some, or all, the hyperparameters of a given ML system are configured for best system's results.
6. Model Evaluation, where all the models are evaluated against the test set via the accuracy, precision, recall, F1 and Receiver Operating Characteristic Area Under the Curve (ROC AUC).
7. Model Maintenance, in which the ML systems are periodically evaluated, retrained and tested in a production environment for new patterns of the target label.

On the other hand, the disposable micro – detectors lifecycle, while using the same procedure as the monolithic approach, is based on the distinct instantiation of one separate model pert target class [5]. In this case, each model instance, also known as detector, is fine – tuned separately on the same data handling pipeline, and as a result, each can use different hyperparameters despite sharing an overall architectural methodology. Additionally, all performance scores for each of the detectors are respectively aggregated and averaged.

Notwithstanding, comparing each, the advantage of the monolithic approach is that it is simpler to maintain and lower memory requirements, whereas micro – detectors allow for isolated model instance retraining and hyperparameter variation. Considering their drawbacks, individual cyberattack classes cannot be studied in isolation when using the monolithic approach, and the execution times of models is expensive when using micro – detectors.

Machine Learning Algorithms

Three different ML systems are evaluated for IDSs, two of which are based on Convolutional Neural Networks (CNN) and one traditional ensemble method, under distinct machine learning project development approaches.

The traditional ensemble method Random Forest Classifier is a ML algorithm which uses a combination of decision trees, generally trained via a bagging or pasting method, typically with its maximum sample size hyperparameter set to the size of the training set [13]. This system introduces a type of extra randomization when growing its trees; instead of searching for the very best feature when splitting a node, it searches for the best feature among a random subset of features, denoted by $\sqrt{n}$, where n is the number of features [13]. Additionally, the version of the system that is being used is Scikit – Learn's; henceforth, it makes use of the following mathematical equations (4), (5) and (6).

$$G = 1 - \sum_{i=1}^{C} p_i^2 \quad (4)$$

$$\Delta i = \frac{N_t}{N}\left(i_t - \frac{N_{t_L}}{N_t} i_{t_L} - \frac{N_{t_R}}{N_t} i_{t_R}\right) \quad (5)$$

$$FI = \sum_{t=1}^{T} \sum_{n}^{N_t} \frac{N_n}{N} \Delta i_n \quad (6)$$

Where G is the Gini impurity, Δi is Scikit Learn's total impurity measure, and FI is the feature importance metric of the ML system itself. Moreover, C is the number of classes, p_i is the probability of class i at a particular node, N is the total number of instances, N_{t_L} and N_{t_R} are the number of samples in the left and right child nodes, N_t is the number of samples at a given current node, i_t, i_{t_L} and i_{t_R} are the impurity measures for the current, left and right nodes, N_t is the set of nodes in tree t where a given feature is used, N_n is the number of samples reaching node n, and Δi_n is Δ_i at a given node n [9].

In the case of the convolutional neural networks, these are systems designed to process data that has known grid – like topologies, such as images and videos [13]. They are comprised of a multitude of layers: (1) Input Layer, where the inputs of the models are handled, these being images represented as 2D or 3D (color) matrices; (2) Convolutional and Activation Layer, where both the filtering of inputs for feature extraction, and an activation function that would allow the overall system to detect complex patterns, would be processed; (3) Pooling Layer, which handles the down – sampler or simplification of the feature images, reducing their dimensions and computation loads; (4) Flattening Layer, which handles the conversion of the 2D feature images into 1D vector for the next layer; (5) Fully Connected Layer, which deals with the input assignments of all neurons in the CNN systems themselves; and (6) Output Layer, where the final output probabilities of each target class are finally generated.

Subsequently, CNNs also use what is known as Loss Functions, i.e.: mathematical tools that are used to quantify the difference between the actual label values and the system's predictions, for the purposes of error minimization. As such, the two that are studied are Cross Entropy Loss and Focal Loss.

Cross Entropy Loss is a convolutional neural network criterion designed to evaluate the difference between predicted probability distributions and the true, label distributions. Its equation is shown in (7).

$$CE(y, \hat{y}) = -\sum_{i=1}^{C} y_i \log(\hat{y_i}) \quad (7)$$

Where y_i are one – hot encoded true labels, $\hat{y_i}$ are predicted probabilities and C is the number of classes. Notwithstanding, Focal Loss is an alternative to Cross Entropy Loss which is used to handle class imbalance [14].

Focal Loss is an alternative to Cross Entropy Loss which is used for class imbalance; shown in (8).

$$FL(p_t) = -a_t(1 - p_t)^{\gamma} \log(p_t) \quad (8)$$

Where p_t are the predicted probabilities of the system's true positives and true negatives, a_t is the weighing factor of t, and γ is a focusing parameter [15].

Now, since the Random Forest Classifier was implemented using Scikit – Learn's RandomForestClassifier class [9], and the convolutional neural networks were implemented with PyTorch, the code of the latter is discussed below.

The Focal_Loss class implements the Focal Loss function. Its constructor has two parameters: alpha and gamma, which correspond to $-a_t$ and γ, respectively. There is also an internal variable to the class called bce representing the log loss measure or $\log(p_t)$. Moreover, the forward function exists to undertake the computations of the Focal Loss function itself. It achieves this by first calculating the standard binary cross – entropy loss between the network inputs and the NSL – KDD target labels. It then, defines pt as the model's estimated probability for the true class values. And, finally, it used both pt and BCE_Loss to compute the loss via its equation $-a_t(1 - p_t)^{\gamma} \log(p_t)$, returning the scalar, that is, average, version of it.

```
class Focal_Loss(nn.Module):
    def __init__(self, alpha=0.25, gamma=2.0):
        super(Focal_Loss, self).__init__()
        self.alpha = alpha
        self.gamma = gamma
        self.bce = nn.BCELoss(reduction="mean")
    def forward(self, inputs, targets):
        BCE_Loss = self.bce(inputs, targets)
        pt = torch.where(targets == 1, inputs, 1 - inputs)
```

```
        loss = self.alpha * (1 - pt) ** self.gamma * BCE_Loss
        return loss.mean()
```

In the cases of the CNN_Binary and CNN_Nominal classes, these exist to create two versions of the same convolutional neural networks, one for binary classification and another for multi – class classification, respectively. Thus, they both show implementations of the CNN layers previously discussed.

```
class CNN_Binary(nn.Module):
    def __init__(self, input_dim):
        super(CNN_Binary, self).__init__()
        self.network = nn.Sequential(
            #  First Layer
            nn.Conv1d(in_channels=1, out_channels=32, kernel_size=3, padding=1),
            nn.BatchNorm1d(32),
            nn.ReLU(),
            #  Second Layer
            nn.Conv1d(32, 64, kernel_size=5, padding=1),
            nn.BatchNorm1d(64),
            nn.ReLU(),
            #  Third Layer
            nn.AdaptiveAvgPool1d(1),
            #  Fourth Layer
            nn.Flatten(),
            #  Fifth Layer
            nn.Dropout(0.5),
            #  Sixth Layer
            nn.Linear(64, 1),
            nn.Sigmoid()
        )
    def forward(self, X):
        return self.network(X)
class CNN_Nominal(nn.Module):
    def __init__(self, input_dim, num_classes):
        super(CNN_Nominal, self).__init__()
        self.network = nn.Sequential(
            nn.Conv1d(1, 32, kernel_size=3, padding=1),
            nn.BatchNorm1d(32),
            nn.ReLU(),
            nn.Conv1d(32, 64, kernel_size=5, padding=1),
            nn.BatchNorm1d(64),
            nn.ReLU(),
            nn.AdaptiveAvgPool1d(1),
            nn.Flatten(),
            nn.Dropout(0.5),
            nn.Linear(64, num_classes)
        )
    def forward(self, x):
        return self.network(x)
```

It should be noted that there is no function for cross entropy since PyTorch already has an implementation of it.

Model Evaluation Metrics

To ensure the fair evaluation of all models and machine learning project development approaches, the following evaluation metrics are used:

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN} \quad (9)$$

$$Precision = \frac{TP}{TP+FP} \quad (10)$$

$$Recall = \frac{TP}{TP+FN} \quad (11)$$

$$F1\ Score = \frac{2\times Precision\times Recall}{Precision+Recall} \quad (12)$$

Where for (9), (10) and (11), TP, TN, FP and FN are true positives, true negatives, false positives and false negatives, respectively [13].

Furthermore, of these metrics, the most important one will be the F1 Score, as it is not prone to outliers, class imbalance and does not generalize results. There is also another metric, Receiver Operating Characteristic Area Under the Curve (ROC AUC) [13] score that is used to identify best models only when the harmonic mean performances of different these are the same.

RESULTS

This section presents the best – final experimental findings for each ML system under the two project development methodologies and two target label configurations. Table 2 shows the best ML F1 Scores performance metrics for each of the models, where

it is seen that, regardless of the project development architecture, random forest classifiers were best suited for binary classification tasks (Monolithic – 2 Class F1 Score: 99.92% and Micro Detectors – 2 Class F1 Score: 99.92%). Additionally, it is also seen that for multiclass classification tasks, convolutional neural networks with cross – entropy loss were most useful when used with micro – detectors (Micro – Detectors F1 Score: 99.59%); otherwise, random forest classifiers can be used but with a performance reduction of 2%, at least.

Table 2
Final Model ML Performance Results (F1 Scores %)

Approach	RF	CNN – F	CNN – C
Mono 2	99.92	98.42	98.78
Mono 5	97.66	82.57	81.65
Micro 2	99.92	98.39	98.82
Micro 5	97.79	99.56	99.59

In the case of accuracy, Table 3 shows each of the models' scores:

Table 3
Final Model ML Performance Results (AC Scores %)

Approach	RF	CNN – F	CNN – C
Mono 2	99.92	98.48	98.83
Mono 5	99.86	98.60	98.53
Micro 2	99.92	98.40	98.82
Micro 5	99.94	99.68	99.70

Of these best models, their hyperparameters were:

- Random Forest Classifier: For both the monolithic and micro – detectors approaches and 2 classes, n_estimators = 150, max_depth = None for every iteration of the model itself. In the case of the 5 classes, the values of max_depth were the same; however, n_estimators equals (150) for monolithic and (100, 150, 100, 50, 25) for micro – detectors.
- Convolutional Neural Network – Cross: For its test case using the micro – detectors approach with 5 classes, its batch size was (128, 128, 128, 128, 128), epoch is (128, 128, 128, 128, 128), learning_rate is (0.005, 0.005, 0.005, 0.005, 0.005). It also has a patience parameter to cause it to stop when the performances of each consequential epoch do not change, which is set to 5.
- It is important to consider that all CNNs worked best during their feature scaling data handling stage, whereas Random Forest Classifier worked best in feature selection, with balancing close second.

Regarding the execution times of all models, Random Forest Classifier constantly outperformed the rest, ranging from 139.64 – 1,762.30 seconds. CNN – Cross Loss came second, with execution times in the ranges of 538.71 – 10,282.05 seconds. And CNN – Focal Loss came last with 1,358.01 – 24,431.95 seconds.

Conclusions and Future Work

The results of this study illuminate several key insights on how traditional and deep – learning approaches perform under different ML project development methodologies and preprocessing phases. It was discovered that traditional ML methods, like the Random Forest Classifier, performed best with binary classification tasks, while deep learning solutions, like CNNs, worked well for multi – class classification given proper data handling procedures.

Additionally, it was also seen that given the data handling procedures used, ML systems may gain an increase in performance, as seen with the Random Forest Classifier and CNNs, whenever the data was in its feature selection and feature scaling stages, respectively. For example, by having a monolithic and multiclass (5 – class) Random Forest Classifier whose inputs have been previously basically preprocessed, feature scaled, balanced and feature selected, it was possible to attained an accuracy score of 99.86%, outperforming Revathi & Malathi's 97.9% (41) and 98.9% (15) counterparts [5].

There is also Zhao, Li, Niu, Shi & Song research [6], which was put into question given this study's results showing that CNNs do not outperformed random forest classifiers in almost all test cases.

Notwithstanding, for possible future work, the following can be done: (1) Expand the developed

Python code to add new techniques or processes that may further enhance the results of this study or create new ones; and (2) Implement other machine learning systems that can be used to compare the results of this study.

REFERENCES

[1] BusinessWire, "Cyber Threats Have Increased 81% Since Global Pandemic," Nov. 9, 2021. [Online]. Available: https://www.businesswire.com/news/home/20211108005775/en/Cyber-Threats-Have-Increased-81-Since-Global-Pandemic. [Accessed: December 3, 2024].

[2] R. Sobers, "Ransomware Statistics, Data, Trends, and Facts [updated 2024]," *Varonis*, Nov. 13, 2024. [Online]. Available: https://www.varonis.com/blog/ransomware-statistics. [Accessed: December 3, 2024].

[3] S. Saad, W. Briguglio and H. Elmiligi, "The Curious Case of Machine Learning In Malware Detection," *ArXiv*, May 18, 2019. [Online]. Available: https://doi.org/10.48550/arXiv.1905.07573. [Accessed: August 29, 2024].

[4] K. Scarfone and P. Mell, "Guide to Intrusion Detection," *National Institute of Standards and Technology*, February 2007. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-94.pdf. [Accessed: April 20, 2025].

[5] S. Revathi and A. Malathi, "A Detailed Analysis on NSL-KDD Dataset Using Various Machine Learning," *International Journal of Engineering Research & Technology (IJERT)*, vol. 2, no. 12, December 2013. [Online]. Available: https://www.ijert.org/research/a-detailed-analysis-on-nsl-kdd-dataset-using-various-machine-learning-techniques-for-intrusion-detection-IJERTV2IS120804.pdf. [Accessed: December 3, 2024].

[6] F. Zhao, H. Li, K. Niu, J. Shi and R. Song, "Application of deep learning-based Intrusion Detection System (IDS) in network anomaly traffic detection," *Applied and Computational Engineering*, vol. 86, Aug. 14, 2024. [Online]. Available: https://doi.org/10.54254/2755-2721/86/20241604. [Accessed: December 3, 2024].

[7] M. H. Zaib, "NSL-KDD," *Kaggle*, 2018. [Online]. Available: https://www.kaggle.com/datasets/hassan06/nslkdd. [Accessed: December 3, 2024].

[8] Y. Sahli, "A comparison of the NSL-KDD dataset and its predecessor the KDD Cup '99 dataset," *International Journal of Scientific Research and Management (IJSRM)*, vol. 10, no. 4, April 23, 2022. Available: https://doi.org/10.18535/ijsrm/v10i4.ec05. [Accessed: February 17, 2025].

[9] Scikit-learn Development and Maintenance Team, "Scikit-learn: Machine Learning in Python," 2025. [Online]. Available: https://scikit-learn.org/stable/. [Accessed: February 17, 2025].

[10] Imbalanced Learn Development Team, "SMOTE," 2024. [Online]. Available: https://imbalanced-learn.org/stable/references/generated/imblearn.over_sampling.SMOTE.html#r001eabbe5dd7-1. [Accessed: February 18, 2025].

[11] Geeks For Geeks, "Information Gain and Mutual Information for Machine Learning," 2024. [Online]. Available: https://www.geeksforgeeks.org/information-gain-and-mutual-information-for-machine-learning/. [Accessed: May 15, 2025].

[12] A. I. Aramendia, "L1 and L2 Regularization (Part 1): A Complete Guide," *Medium*, March 31, 2024. [Online]. Available: https://medium.com/@alejandro.itoaramendia/l1-and-l2-regularization-part-1-a-complete-guide-51cf45bb4ade. [Accessed: May 15, 2025].

[13] A. Geron, *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*, 3rd edition, Boston: O'Rilley Media, 2022.

[14] K. Pykes, "Cross-Entropy Loss Function in Machine Learning: Enhancing Model Accuracy," *DataCamp*, August 10, 2024. [Online]. Available: https://www.datacamp.com/tutorial/the-cross-entropy-loss-function-in-machine-learning. [Accessed: May 6, 2025].

[15] T.-Y. Lin, P. Goyal, R. Girshick, K. He and P. Doll´ar, "Focal Loss for Dense Object Detection," *ArXiv*, February 7, 2018. [Online]. Available: https://arxiv.org/pdf/1708.02002v2. [Accessed: May 6, 2025].