A Network Intrusion Detection System realized using Neural Networks

1st Alvise de' Faveri Tron
Advanced Data Exploration and Optimization
EISTI
Cergy, France
defaveritr@eisti.eu

***Abstract*—Faced with the increase of more advanced attacks targeting information systems, a defense system has become vital. An intrusion detection system provides a first line of defense. A intrusion detection system monitor events within an information system or in one of the organs of the information system The objective of this research project is to design a lightweight intrusion detection system using artificial intelligence techniques, in particular deep learning techniques. The neural network will be trained and tested with the NSL KDD dataset.**

***Index Terms*—Neural Networks, Intrusion Detection, Computer Security**

I. Introduction

A. Intrusion Detection Systems

Intrusion Detection is a key concept in modern computer networks security. Rather than protecting a network against known malware by preventing the connection to enter the network, as in Intrusion Prevention Systems, Intrusion Detection is aimed to analyzing the current state of a network in real-time and identify potential anomalies that are happening in the system, reporting them as soon as they are identified. This enables the possibility to detect previously unknown malware [1].

Intrusion Detection Systems are generally classified according to the following categories [2]:

- **Anomaly Detection vs Misuse Detection**: In misuse detection, each instance in a data set is labeled as 'normal' or 'intrusive' and a learning algorithm is trained over the labeled data. Anomaly detection approaches, on the other hand, build models of normal data and detect deviations from the normal model in observed data.
- **Network-based vs Host-based**: Network intrusion detection systems (NIDS) are placed at a strategic point or points within the network to monitor traffic to and from all devices on the network, while Host intrusion detection systems (HIDS) run on individual hosts or devices on the network.

The object of this work, in particular, is the production of a NIDS trained on labelled data which is able to recognize suspect behaviour in a network and classify each connection as normal or anomalous.

B. Artificial Neural Networks

Artificial Neural Networks are supervised machine learning algorithms inspired by the human brain. The main idea is to have many simple units, called *neurons*, organized in *layers*. In particular, in a feed-forward artificial neural network all neurons of a layer are connected to all the neurons of the following layer, and so on until the last layer, which contains the *output* of the neural network.

This kind of networks are a popular choice among Data Mining techniques in now days, and have already been proven to be a valuable choice for Intrusion Detection [3, 4].

In this work we are using feed-forward neural networks trained on the NSL-KDD dataset to classify network connections as belonging to one of two possible categories: *normal* or *anomalous*. The goal of this work is to maximize the accuracy in recognizing new data samples, while also avoid *overfitting*, which happens when the algorithm is too attached to the data it learned and is not capable of correctly generalizing on previously unseen data.

C. The NSL-KDD Dataset

The dataset used for training and validation of the neural network is the NSL-KDD dataset, which is an improved version of the KDD CUP '99 dataset [5, 6]. This data set is a well known benchmark in the field of Network Intrusion Detection techniques, providing 42 features for each example and many anomalous examples.

The dataset has been taken from the University of New Brunswick's website.[7].

A detailed analysis of the dataset is provided in the next Section.

II. Dataset Analysis

This section provides a complete analysis of the NSL-KDD dataset.

A. Dataset Summary

The NSL-KDD dataset is provided in two forms: `.arff` files, with binary labels, and `.csv` files, with categorical labels for each instance.

Since the object of this work is to build a binary classifier, we will focus only on the `.arff` files. The provided `.arff` files are:

- **KDDTrain+.ARFF**: The full NSL-KDD train set with binary labels in ARFF format

- **KDDTrain+_20Percent.ARFF**: A 20% subset of the KDDTrain+.arff file
- **KDDTest+.ARFF**: The full NSL-KDD test set with binary labels in ARFF format
- **KDDTest-21.ARFF**: A subset of the KDDTest+.arff file which does not include records with difficulty level of 21 out of 21

To avoid redundancy, we use only the `KDDTrain+.ARFF` and `KDDTest+.ARFF` files, which contain a total of about 148,500 entries.

Table I contains a summary of the most important attributes of the dataset.

Summary	
train rows	125973
test rows	22544
total rows	148517
columns	42
duplicates	629
null values	None

TABLE I: Dataset Attributes

B. Features

As for the features of this dataset, they can be broken down into 4 types (excluding the target column):

- 4 Multi-Class
- 6 Binary
- 16 Discrete
- 15 Continuous

Tables II and III contain a description of the categorical features of the dataset, while Tables IV and V describe the numerical features. Note that this definition slightly differs from the one provided by [5], since here categorical and binary features are listed regardless of their format (text or numeric).

C. Categories Distributions

The NSL-KDD dataset contains a few binary and multi-class categorical features, which are used to label each connection with its protocol, service and other useful characteristics, e.g. if the user tried to gain super-user access or if he spawned a root shell during its connection.

Multi-Class Features	
Feature	*Distinct Values*
protocol_type	3
service	70
flag	11
su_attempted	3

TABLE II: Multi-Class Features in the NSL-KDD dataset

Binary Features	
Feature	*Number of '0's*
land	99.98%
logged_in	59.72%
root_shell	99.85%
num_outbound_cmds	100.00%
is_host_login	99.99%
is_guest_login	98.77%

TABLE III: Binary Features in the NSL-KDD dataset

Discrete Features		
Feature	*Mean*	*Stddev*
wrong_fragment	0.02	0.24
urgent	0.00	0.02
count	83.34	116.76
srv_count	28.25	75.37
dst_host_count	183.93	98.53
dst_host_srv_count	119.46	111.23
duration	276.78	2.46k
src_bytes	40.22k	5.4M
dst_bytes	17.08k	3.7M
hot	0.19	2.01
num_failed_logins	0.00	0.07
num_compromised	0.26	22.23
num_root	0.27	22.69
num_file_creations	0.01	0.52
num_shells	0.00	0.03
num_access_files	0.00	0.10

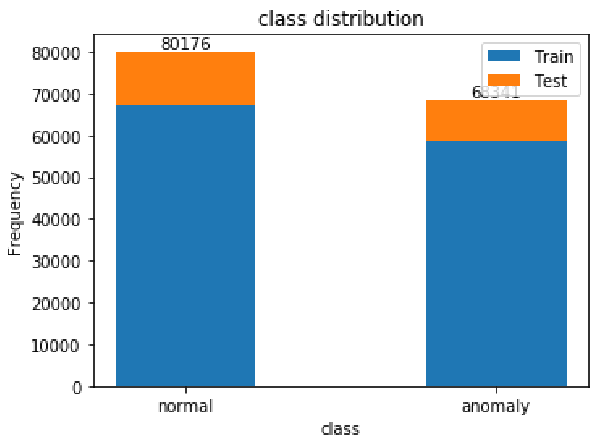
TABLE IV: Discrete Features in the NSL-KDD dataset

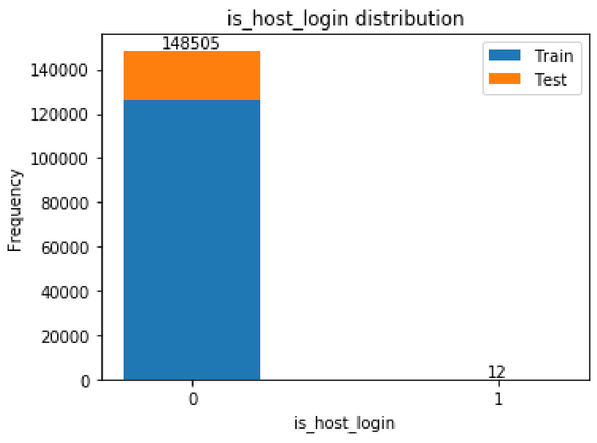
Continuous Features		
Feature	*Mean*	*Stddev*
serror_rate	0.26	0.43
srv_serror_rate	0.26	0.43
rerror_rate	0.14	0.34
srv_rerror_rate	0.14	0.34
same_srv_rate	0.67	0.44
diff_srv_rate	0.07	0.19
srv_diff_host_rate	0.10	0.26
dst_host_same_srv_rate	0.53	0.45
dst_host_diff_srv_rate	0.08	0.19
dst_host_same_src_port_rate	0.15	0.31
dst_host_srv_diff_host_rate	0.03	0.11
dst_host_serror_rate	0.26	0.43
dst_host_srv_serror_rate	0.25	0.43
dst_host_rerror_rate	0.14	0.32
dst_host_srv_rerror_rate	0.14	0.34

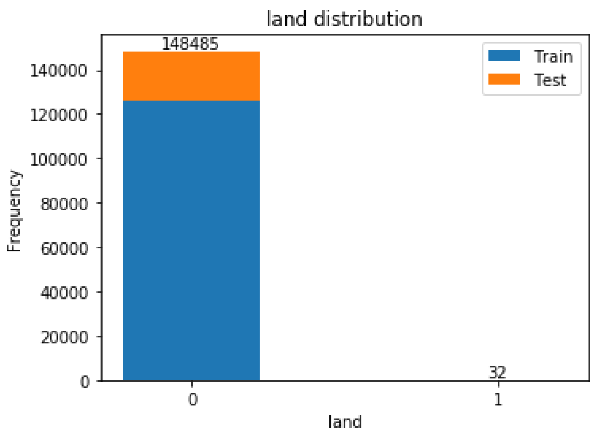
TABLE V: Continuous Features in the NSL-KDD dataset

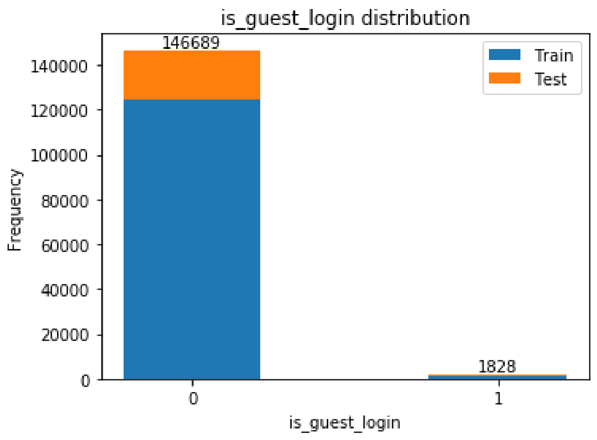
The figures contained in the next page describe, for each categorical feature, how many occurrences of each label are present in both the train and the test set. Note that *class*, *protocol_type*, *service* and *flag* features contain textual values, while all other categories are expressed by a number.

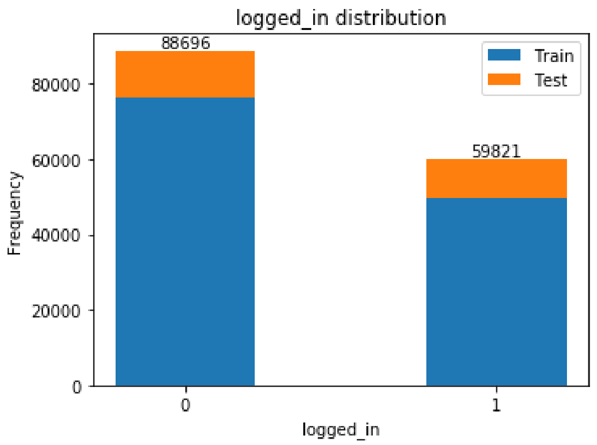
As we can see, the target variable (*class*) has a nearly even distribution of its values (*normal* and *anomalous)* throughout the dataset, meaning that there are a lot of anomalous examples that can be used for training. As opposed to this, we can see that some of the categories are not evenly distributed, having more than 99% of the samples belonging to just one class. This kind of analysis can be useful during the feature selection step.

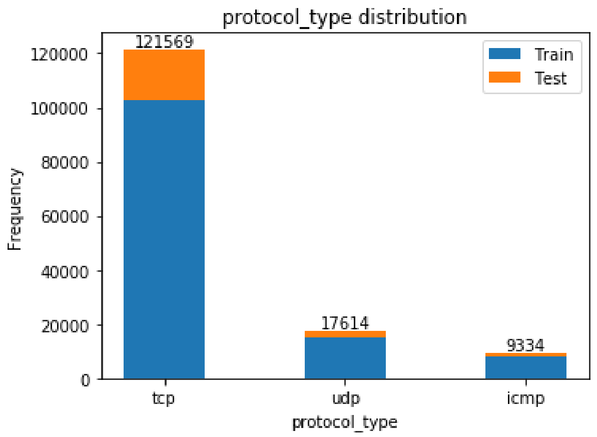

class distribution
Train
Test
80176
68341
Frequency
normal
anomaly
class

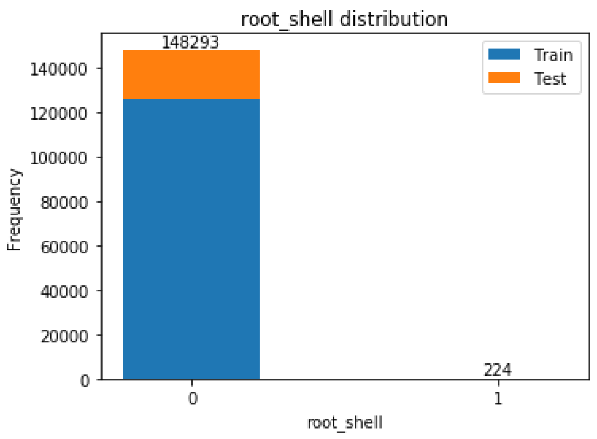

is_host_login distribution
Train
Test
148505
12
Frequency
0
1
is_host_login

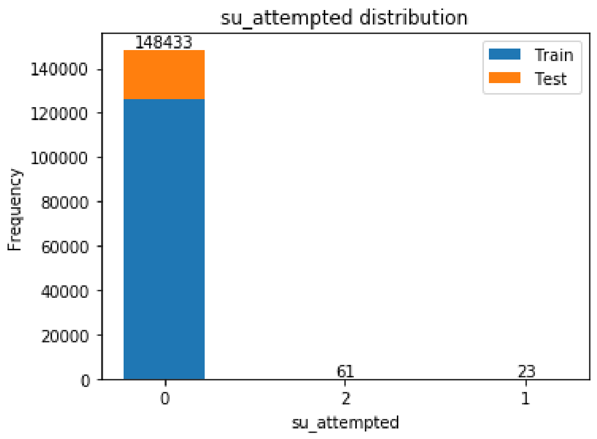

land distribution
Train
Test
148485
32
Frequency
0
1
land

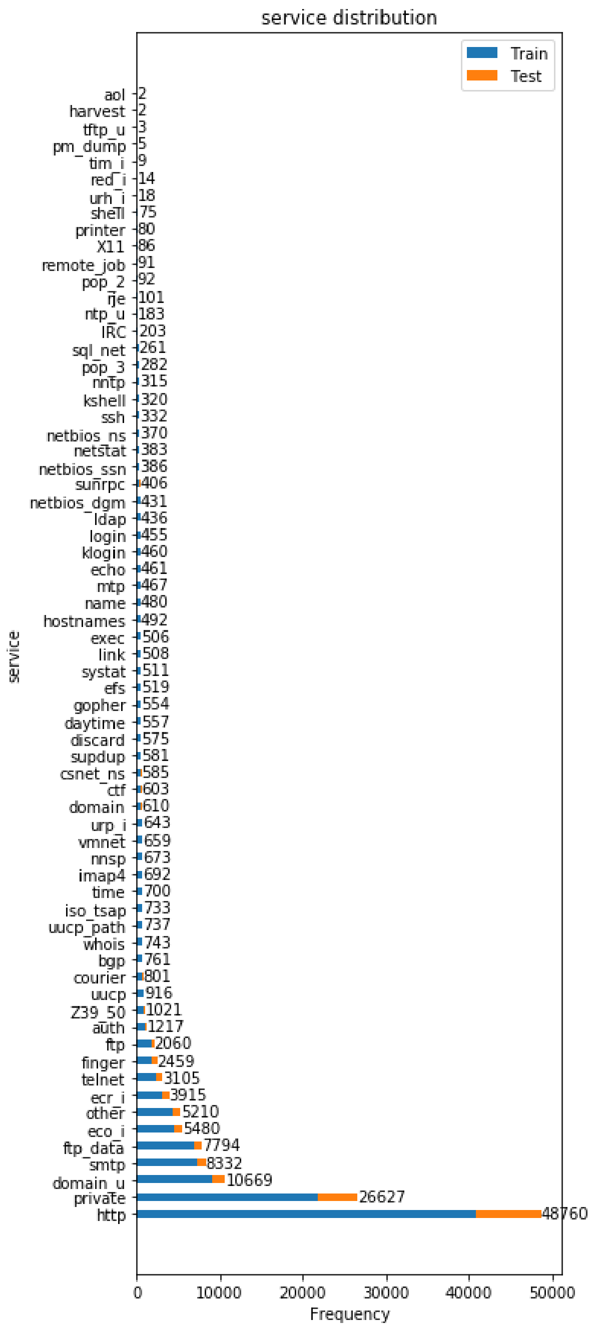

is_guest_login distribution
Train
Test
146689
1828
Frequency
0
1
is_guest_login

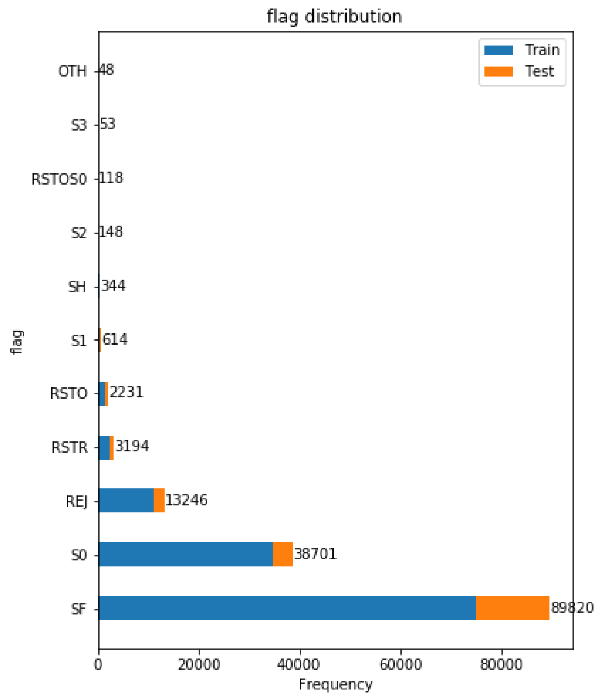

logged_in distribution
Train
Test
88696
59821
Frequency
0
1
logged_in


protocol_type distribution
Train
Test
121569
17614
9334
Frequency
tcp
udp
icmp
protocol_type


root_shell distribution
Train
Test
148293
224
Frequency
0
1
root_shell


su_attempted distribution
Train
Test
148433
61
23
Frequency
0
2
1
su_attempted





D. Numerical Data Distribution

Numerical features have very different meanings in this dataset, and consequently different ranges. Continuous features are used for rates (e.g. error rates) and discrete features give information about the number of bytes in the packet, the connection duration, the number of reconnections and many other variables.

Figure 1 represents the normalized dataset: each column's value has been normalized between 0 and 1 in order to visualize how the different values of each feature are distributed. Note that this normalization takes into account both the train and the test set, hence it can be used only for data analysis. Section III-C describes how the train and test set have been normalized for learning.

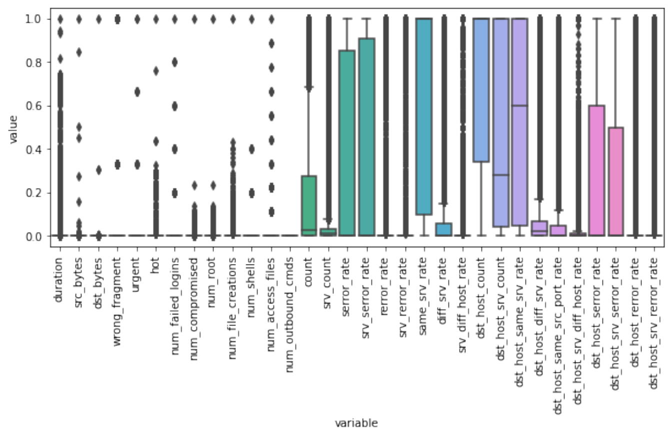


Fig. 1: Distribution of discrete and continuous values in the normalized dataset.

E. Correlation Matrix

As a final step of the data analysis, we ran a correlation analysis for all the features of the dataset. Figure 2 illustrates the correlation matrix. Focusing on the *class* column, we can see how each feature is correlated with the target variable, either directly or inversely. This analysis can be extremely useful when performing feature selection to estimate how many important features there are in the dataset.

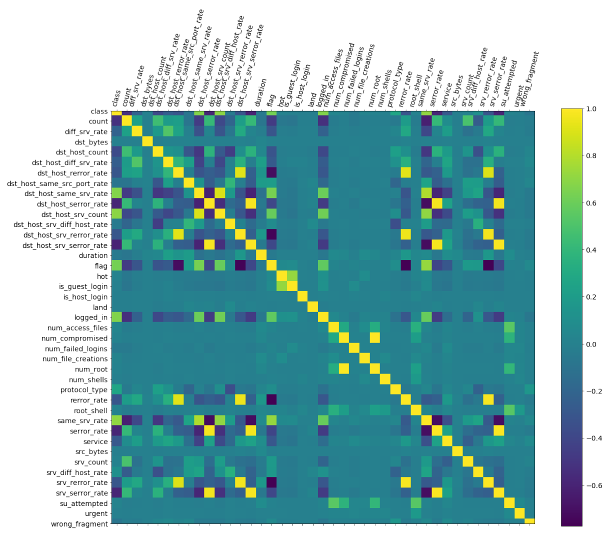


Fig. 2: Correlation matrix between each feature of the dataset. A brighter color means higher correlation.

III. Data Preparation

In this Section we will describe all the techniques used for cleaning and preparing the data for the learning phase.

A. Data Cleaning

Since the NSL-KDD dataset is already an enhanced version of the older KDD '99 CUP dataset, little additional cleaning had to be performed: the set had already been cleaned from redundant data and null values [5]. Also, the ratio between normal and anomalous entries is good for machine learning purposes.

The only step taken at this stage, a part from loading the `.arff` files and decoding strings using UTF-8, was to change all the entries with *http_XXX* as their `service` values into *http*, where *XXX* denotes the port number.

This decision was taken since the port of an http connection, which is specified in the protocol string only for some entries, is very unlike to be correlated with an anomalous behaviour in a general case. Also, leaving this distinction could lead to new http connections on previously unseen ports to not be recognized correctly by the algorithm.

This cleaning has later been proven to be an effective solution for partially reducing overfitting.

B. Categorical Columns Encoding

After cleaning the dataset, the second step was to convert all categorical columns into one-hot encoded columns. This means that, for each distinct value of each categorical column, a new column is generated, contaning '1' for the rows belong to that category and '0' elsewhere.

This kind of encoding is the most popular choice when it comes to preparing data for ANN learning [8]. This is done to prevent the algorithm from interpreting categories as numbers, which can lead to problems like the algorithm considering a category the mean of other two categories etc.

The affected columns in particular are:

- service
- protocol_type
- su_attempted
- flag
- class (target variable)

The dataset resulting from this encoding contains **124** columns, of which 2 are the target columns.

Note that, a part from the target column, all other binary columns where not affected by the encoding. This is to avoid redundant columns, since binary features would have two corresponding encoded columns in which one can be directly inferred by observing the other.

C. Normalization

Having encoded the categorical columns, now the numerical columns have to be treated. In particular, it was decided to apply MinMax [9] normalization to both discrete and continuous columns. The normalization process has been accomplished as follows:

- Fit the model onto the train set
- Transform the train set
- Transform the test set (with the same normalization model)

Figure 3 describes the distribution of the numerical values in the train set after normalization, while Figure 4 illustrates the same transformation applied to the test set. As we can see, all values in the train set are correctly normalized between 1 and 0, while the test set contains some values that are greater than the maximum found on the train set, hence their normalized values is greater than 1.

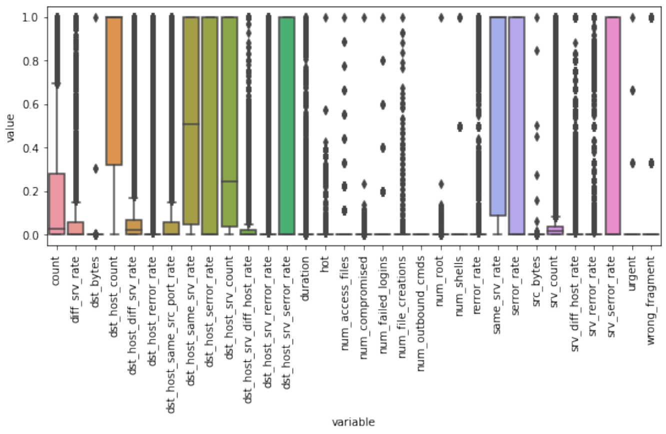


Fig. 3: Distribution of numerical values in the normalized train set.

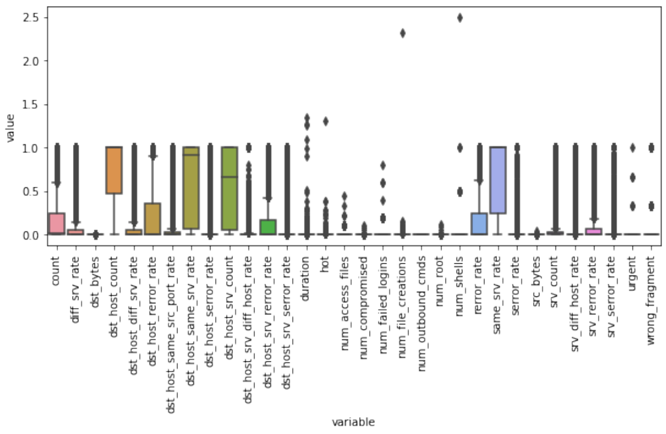


Fig. 4: Distribution of numerical values in the normalized test set.

IV. Feature Selection

Since the one-hot encoding increased considerably the number of features, some feature selection is needed to reduce the problem's complexity and reduce overfitting. If not correctly performed, feature selection can introduce some penalties in terms of the accuracy of the final output. A further discussion and comparision of the different models can be found in Section VI.

For this work, two different feature selection techniques have been chosen: ExtraTreeClassification and Univariate Selection.

A. *ExtraTree Classifier*

ExtraTreeClassifier is a classification algorithm provided by python's `scikit-learn` library [10], which employs the random forest method to build a classifier and evaluate the feature importance.

Figure 5 shows the ranking of the 15 best features obtained when running the classifier on the train set.

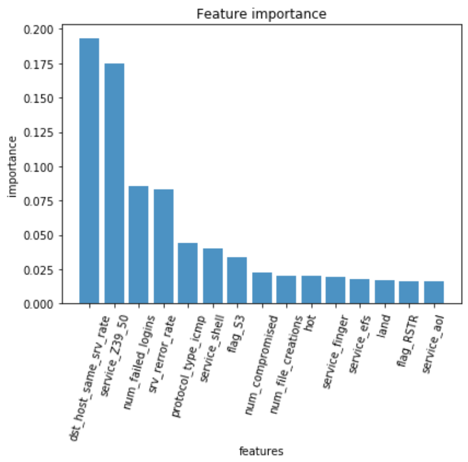


Fig. 5: Top 15 feature by importance as per ExtraTree Classifier ranking

B. *Univariate Selection*

Univariate feature selection is another method provided by the `scikit-learn` collection: it examines each feature individually to determine the strength of the relationship of the feature with the response variable [11].

Figure 6 shows the best 15 features found with Univariate Selection.

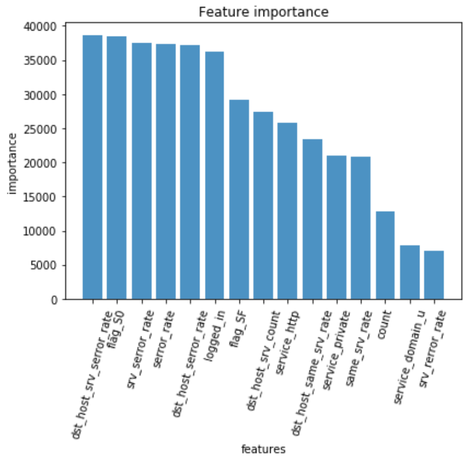


Fig. 6: Top 15 feature by importance as per Univariate Selection ranking

V. The Model

This Section describes how the Neural Network models where built and how the learning hyperparameters where set.

A. *Project Environment*

This project has been carried out using python3.6. To build the ANN model, the Keras library has been employed, with TensorFlow as a backend. The whole project has been developed on a Jupyter Notebooks project hosted on Google Colab [12].

B. Building the Neural Network

The approach for constructing the neural network for this work has been to try different combinations of layers and see how each model performs in terms of accuracy and loss, comparing these two metrics for the train and test set.

All neural networks have been implemented in Keras [13] using the Sequential (i.e. feed-forward) model and Dense (i.e. fully interconnected) layers. Following the work of [14], the tested models consists of both *shallow* networks and *deep* networks with 2 hidden layers.

C. Hyperparameters

The hyperparameters regarding the optimization and learning of each model have been mostly kept at the default values provided by Keras [15]. These are some of the defaults that Keras provides:

- Learning Rate: 0.001
- Dropout: None
- Optimization Algorithm: Adam

D. Epochs

Each model was trained with a fixed number of epochs (**150**) and batch size = **10** for comparison purposes. In future developments, fixed epochs could be substituted by early stopping to speed up the learning phase.

E. Activation Function

Finally, the chosen activation function for all layers is the **sigmoid** function, which has proven to outperform other tested solutions, like using *relu* for the input layer or *tanh* for the hidden layers.

VI. Validation

Many models have been tested in order to compare different solutions and find the best fit for this task. This section provides a comparison between the performance of each neural network model.

Each model comes with a description of its shape (i.e. number of layers, units per layer etc.), the training history (loss and accuracy in each epoch) and the final results on the test set (confusion matrix).

A. Train-Test Splitting

The NSL-KDD dataset is already divided into a train set containing 125973 instances ($\approx 85\%$) and test set containing 22544 instances ($\approx 15\%$). We also use 15% of the train set as *dev* set, to keep track of the accuracy and loss of each model during the training phase.

The initial batch of models have been tested on the provided test set, according to the initial division. Nevertheless, during the the development of this work it was been noted that reshuffling the train and test set significantly improves the ANN performance: these results are given in Section VI-F
.

The data normalization has not been performed on the entire dataset, but separately for the train and the test set using a normalization algorithm that was fitted on the train set. This ensures that the training of the model is not affected by the values that are in the test set.

B. Evaluation Metrics

The most common evaluation metrics for IDS, as reported by [6], are *Attack Detection Rate (DR)* and *False Alarm Rate (FAR)*.

$$DR = \frac{TP}{TP + FN}$$

$$FAR = \frac{FP}{FP + TN}$$

We can calculate these metrics from the confusion matrix, which is represented by the following terms.

- True Positive (TP): the instance is correctly predicted as an attack.
- True Negative (TN): correctly predicted as a non-attack or normal instance.
- False Positive (FP): a normal instance is wrongly predicted as attacks.
- False Negative (FN): an actual attack is wrongly predicted as non-attacks or normal instance.

False positives where a normal network activity is classified as an attack can waste the valuable time of security administrators. False negatives on the other hand have the worst impact on organizations, since an attack is not detected at all.

C. All features

Six different models where tested using all 122 features as inputs: 4 with only one hidden layer and 2 with 2 hidden layers. Table VI reports the evaluation of each model.

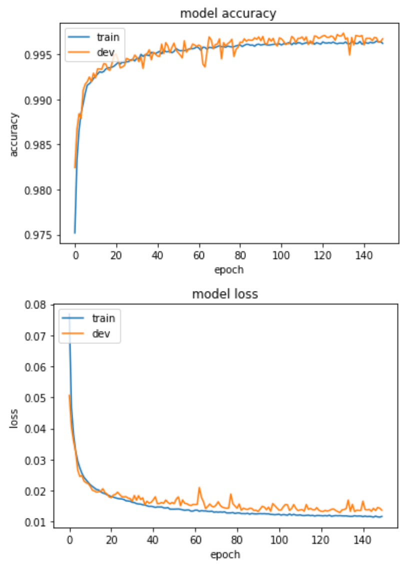


Fig. 7: Learning Curve models of the best model trained on the full data set

Model			**Evaluation**	
Inputs	*Selection Algorithm*	*Hidden Units*	*DR*	*FAR*
122	None	80	71.4%	0.3%
122	None	80, 30	68.1%	0.3%
122	None	30	71.92%	8.2%
122	None	50, 10	72.61%	7.6%
122	None	20	67.71%	7.9%
122	None	10	66.89%	7.7%

TABLE VI: Comparison between models trained on the full data set

D. ExtraTree selected features

Four different models where tested using ExtraTree Classifier as feature selection algorithm: 2 with the top 80 features and 2 with the top 15 features in the importance ranking. Table VII reports the evaluation of each model.

Model			**Evaluation**	
Inputs	*Selection Algorithm*	*Hidden Units*	*DR*	*FAR*
80	ExtraTree	10	61.1%	7.8%
80	ExtraTree	30, 10	69.29%	7.6%
40	ExtraTree	10	59.7%	3%
40	ExtraTree	30, 10	59.5%	3%
15	ExtraTree	8	67.2%	6.86%
15	ExtraTree	4	64.6%	7.3%

TABLE VII: Comparison between models trained on Extra-Tree selecte features

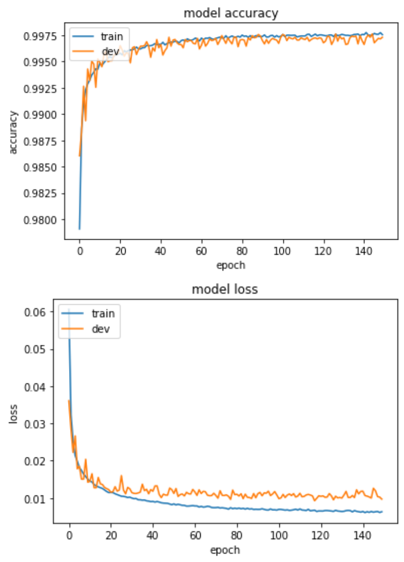


Fig. 8: Learning Curve models of the best model trained on Extra Tree selected features

E. Univariate selected features

Four different models where tested using Univariate Selection as feature selection algorithm: 2 with the top 80 features and 2 with the top 15 features in the importance ranking. Table VIII reports the evaluation of each model.

Model			**Evaluation**	
Inputs	*Selection Algorithm*	*Hidden Units*	*DR*	*FAR*
80	Univariate	10	66.56%	7.4%
80	Univariate	30, 10	70%	7.4%
40	Univariate	10	69%	7.8%
40	Univariate	30, 10	73.7%	3.4%
15	Univariate	8	57%	7.4%
15	Univariate	4	63.7%	3.1%

TABLE VIII: Comparison between models trained on Univariate selected features

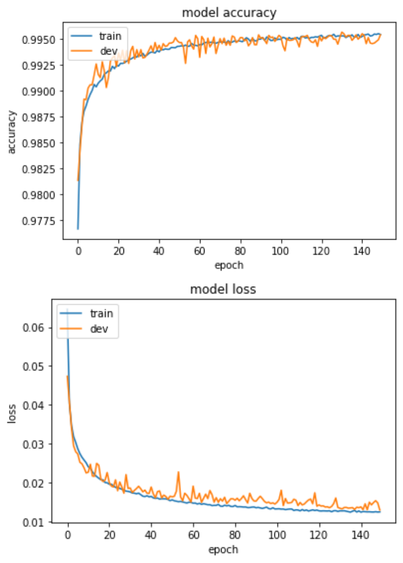


Fig. 9: Learning Curve models of the best model trained on Univariate selected features

F. Re-shuffled Train Set

Given that the results of the preceding models are far from being enough to deploy an IDS in a real network, another approach has been tried. The train and test set have been merged together, split again in train and test set and re-normalized using a normalization based on the train set alone. This approach gave much better results, meaning that probably the test set has many outliers that have to be taken into account when designing the ANN. Figure IX displays the results of the same models but with the shuffled dataset.

Model			**Evaluation**	
Inputs	*Selection Algorithm*	*Hidden Units*	*DR*	*FAR*
122	None	10	99.18%	1.2%
122	None	50, 10	99%	0.6%
122	None	20	98.15%	0.3%
122	None	30	99.3%	0.8%
122	None	80, 30	99.3%	0.5%
80	Univariate	10	98.6%	1.8%
80	Univariate	30, 10	99%	1.1%
40	Univariate	10	98.18%	1.8%
40	Univariate	30, 10	98.7%	1.3%
15	Univariate	8	90.9%	3.2%
15	Univariate	4	91.6%	3.7%

TABLE IX: DR and FAR with new train/test set splitting

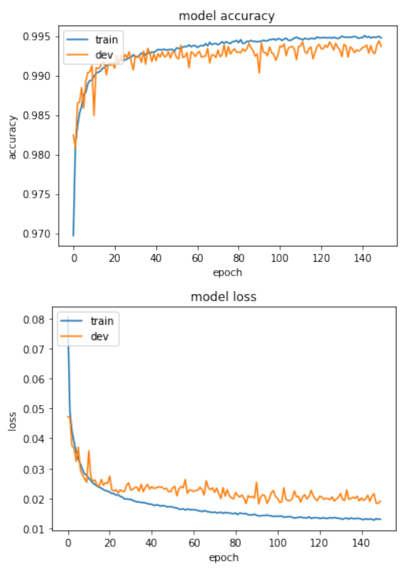


Fig. 10: Learning Curve models of the best model trained on the shuffled data set

VII. Deployment in a Network

This Section describes which steps should be made in order to deploy one or more of the produced models in a real network environment.

A. Sensors

In order to deploy an IDS in a real network, the first components we need are sensors that are able to record traffic and produce the tuples that will be then fed into the aforementioned model.

Sensor placement is a key factor in an IDS to be able to protect the network from invasion. For this task, the network architecture should be studied in detail, to observe all possible points of entrance. Depending on how large and complex is our network, we can choose to deploy the whole system in a single device, as suggested by [16], or implement multiple sensors as done in [17].

In both cases, the most sensitive points where to place the sensors are typically before and after each router or DMZ of the network. Figure 11 describes this situation.

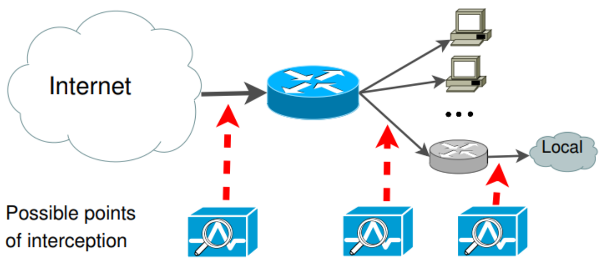


Fig. 11: Possible points of interception for a NIDS

Generally, IDS sensors have two network interfaces — one for monitoring traffic and one for management. The traffic-monitoring interface is unbound from any protocol, which means that the interface has no IP address and other entities can't communicate with it. This guarantees that no attack surface is exposed on the network by the sensor itself.

B. Model Choice

After deploying the sensors, we have to choose which model to deploy. The model choice can depend on evaluation metrics like Detection Rate or False Alam Rate, but should also take into account the model complexity. A higher model complexity can in fact imply a higher cost for data acquisition and higher delays for the system's response. This kind of analysis shall be done taking into consideration also the time of response of each model, which is out of the scope of this work.

For reducing the complexity we can employ one of the Feature Selection methods described in Section IV.

C. Model Deployment

Once the IDS is deployed in the network, we want it to work with streams of incoming data in a fully automated way. This implies several steps, which are described in Figure 12.

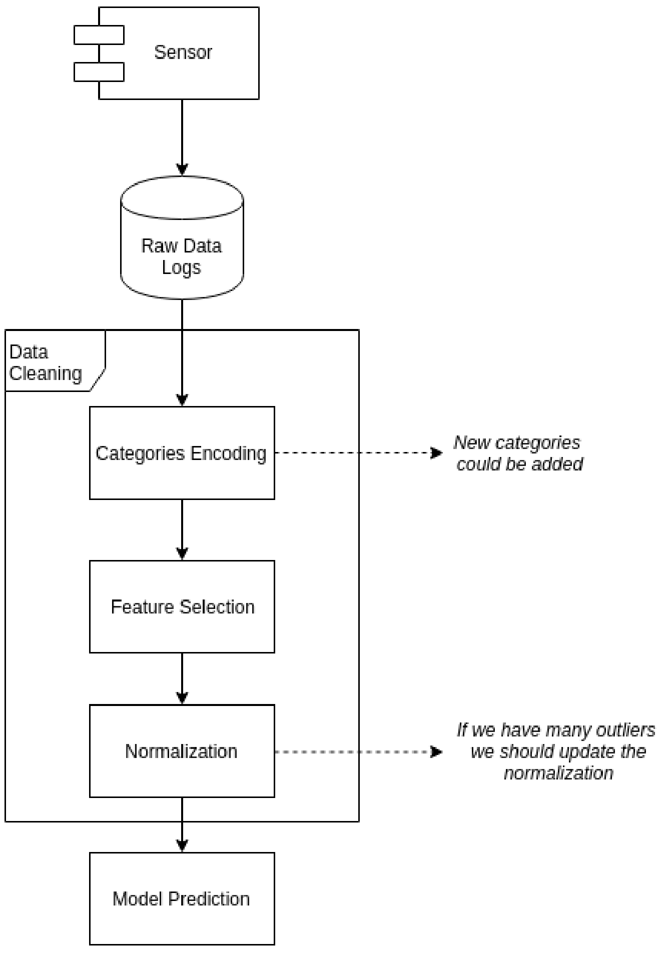


Fig. 12: Steps of the IDS once deployed

There are many possible solutions, to achieve this automation, which range from fully personalized solutions to fully COTS products. One simple possibility, since the models of this work have been trained using TensorFlow, is to use TensorFlow *Extend* [18], a collection of tools used for deploying ANN models in the wild. Figure 13 describes the components of the TFX framework, which range from data cleaning to Keras model validation.

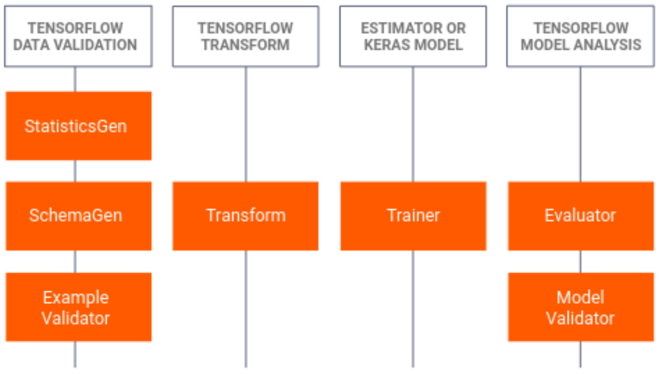


Fig. 13: Image taken from https://www.tensorflow.org/tfx/guide

On-Line deployment can be achieved by using *TensorFlow Serving* [19], which is a framework built to enable fast TensorFlow models' deployment over REST APIs. TF Serving also has the possibility to deploy an ML model in a Docker image and use Kubernetes to manage a cluster of these images running together. This offers a good solution in terms of scalability of the IDS, which is able to easily keep up with a possible growth of the network that it's protecting.

D. Model Adaptation

Once it is online, the model should be then trained and updated with real data coming from the network. This is a time-consuming task, which requires the network to be in a temporary 'safe' state in which the model can learn which is the normal behaviour of the system. After the training time, the IDS is ready to be used in the network environment.

To extend the training time, a human supervisor can be assigned to checking the entries that are signaled as *anomalous* and relabelling them if necessary. The same model can also be periodically retrained with a larger dataset or with only the latest entries.

VIII. Future Work

This work represents only a preliminary analysis of how a NIDS can be modelled and deployed using ANN and TensorFlow.

Many improvements can be made to the models in order to get better performances, such as implementing early stopping instead of a fixed amount of epochs and adapt the keras model to work with TPUs (Tensor Processing Units) which can significantly improve the training time needed.

Also, an in-depth analysis of a target network should be performed in order to deploy the IDS in a real network. Problems as the efficiency, delay and the samples collection in the new network environment are specific of each network, and have to be taken into account if we want to carry out the IDS deployment.

References

[1] B. Mukherjee, L. T. Heberlein, and K. N. Levitt. Network intrusion detection. *IEEE Network*, 8(3):26–41, May 1994.

[2] Aleksandar Lazarevic, Levent Ertöz, Vipin Kumar, Aysel Ozgur, and Jaideep Srivastava. A comparative study of anomaly detection schemes in network intrusion detection. In *SDM*, 2003.

[3] B. Subba, S. Biswas, and S. Karmakar. A neural network based system for intrusion detection and attack classification. In *2016 Twenty Second National Conference on Communication (NCC)*, pages 1–6, March 2016.

[4] Ekambaram Kesavulu Reddy. Neural networks for intrusion detection and its applications. *Lecture Notes in Engineering and Computer Science*, 2:1210–1214, 07 2013.

[5] L. Dhanabal and S. P. Shantharajah. A study on nsl-kdd dataset for intrusion detection system based on classification algorithms. 2015.

[6] R. Thomas and D. Pavithran. A survey of intrusion detection models based on nsl-kdd data set. In *2018 Fifth HCT Information Technology Trends (ITT)*, pages 286–291, Nov 2018.

[7] NSL-KDD Dataset. https://www.unb.ca/cic/datasets/nsl.html.

[8] Kedar Potdar, Taher Pardawala, and Chinmay Pai. A comparative study of categorical variable encoding techniques for neural network classifiers. *International Journal of Computer Applications*, 175:7–9, 10 2017.

[9] MinMaxScaler implementation. https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html.

[10] ExtraTree Classifier Reference. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.ExtraTreesClassifier.html.

[11] Univariate Selection Reference. https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SelectKBest.html#sklearn.feature_selection.SelectKBest.

[12] Google Colab project. https://colab.research.google.com/drive/1ogkvVnKJUkEbUgkLLX6Nx7CsuAGiLSFf.

[13] Keras Library documentation. https://keras.io/.

[14] D. E. Kim and M. Gofman. Comparison of shallow and deep neural networks for network intrusion detection. In *2018 IEEE 8th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 204–208, Jan 2018.

[15] Tensorflow ADAM implementation. https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adam.

[16] S. Kim, N. Nwanze, W. Edmonds, B. Johnson, and P. Field. On network intrusion detection for deployment in the wild. In *2012 IEEE Network Operations and Management Symposium*, pages 253–260, April 2012.

[17] H. Chen, J. A. Clark, S. A. Shaikh, H. Chivers, and P. Nobles. Optimising ids sensor placement. In *2010 International Conference on Availability, Reliability and Security*, pages 315–320, Feb 2010.

[18] TensorFlow Extend Guide. https://www.tensorflow.org/tfx.

[19] TensorFlow Serving Guide. https://www.tensorflow.org/tfx/guide/serving.