Python Dependencies:

Requires: Python3 and following python3 packages Install the following packages using command pip install “*dependency\_package*”

## **numpy, tensorflow, keras, sklearn, joblib, matplotlib**

To run python files type: python3 “*file\_name.py*”

The job of an intrusion detection sounds fairly simple, that is detect any abnormality observed in the network but in order to make a system like that we need a lot of information beforehand, like a good amount of training data that can be used to make the IDS better and constantly updating the system with new threats and vulnerabilities.

**Problem 1.** In order to generate training data, we need to label it and that is an expensive process. Usually human interaction is involved in producing labeled data and human labor is always expensive.

**Problem 2.** Even after managing to get labeled data, there still exists another problem, which is Class Imbalance problem. The amount of data generated by network traffic is huge and out of that data barely 4-5% of the traffic is malicious which gives rise to class imbalance problem.

Now, in order to deal with these problems, we are proposing DAID (Deep Active Intrusion Detection) in which we develop a deep neural network (CNN) which deals with the scarcity of labeled data. We then add an Active Learning Technique that optimizes model performance by picking up most valuable samples from unlabeled dataset and that takes care of class imbalance problem. In order to improve the efficiency of the program we implemented parallelization techniques in python for active learning and hyperparameter tuning.

## III. Data Set

In order to user our approach we need a lot of data which can be easily generated using Wireshark since the data we require is indeed network traffic. But the problem with that will be labeling. Since Wireshark does not label the data by itself, we will manually need to label the data which will not be a reliable process. In network intrusion detection (IDS), anomaly-based approaches in particular suffer from accurate evaluation, comparison, and deployment which originates from the scarcity of adequate datasets. Many such datasets are internal and cannot be shared due to privacy issues, others are heavily anonymized and do not reflect current trends, or they lack certain statistical characteristics. These deficiencies are primarily the reasons why a perfect dataset is yet to exist.

A labeled dataset is of immense importance in the evaluation of various detection mechanisms. Hence, creating a dataset in a controlled and deterministic environment allows for the distinction of anomalous activity from normal traffic; therefore, eliminating the impractical process of manual labeling.

There for we decided to use a well-known dataset called ‘ISCX Dataset’ which is collected by the Canadian Institute for Cybersecurity. It is a benchmark intrusion detection dataset with contains 7 days of synthetically recorded packet details replicating the real time network traffic by labelling the attacks. We used the **ISCXISD 2012[4]** . Figure 1 shows the Distribution of Normal vs Anomaly samples in ISCXISD 2012 Dataset.

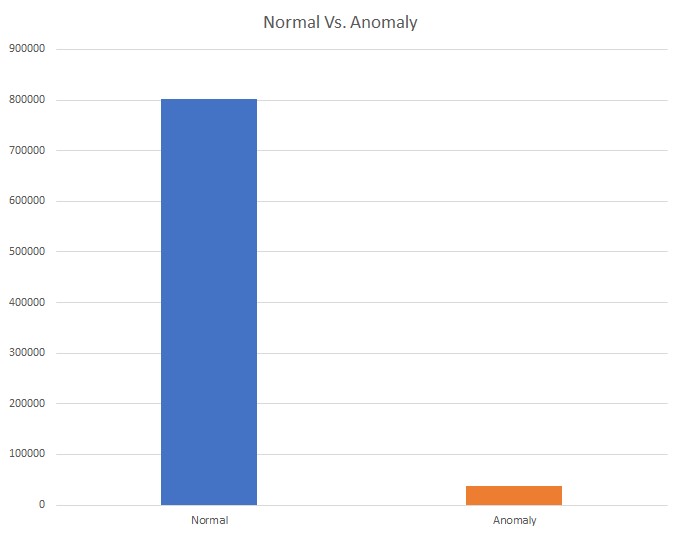
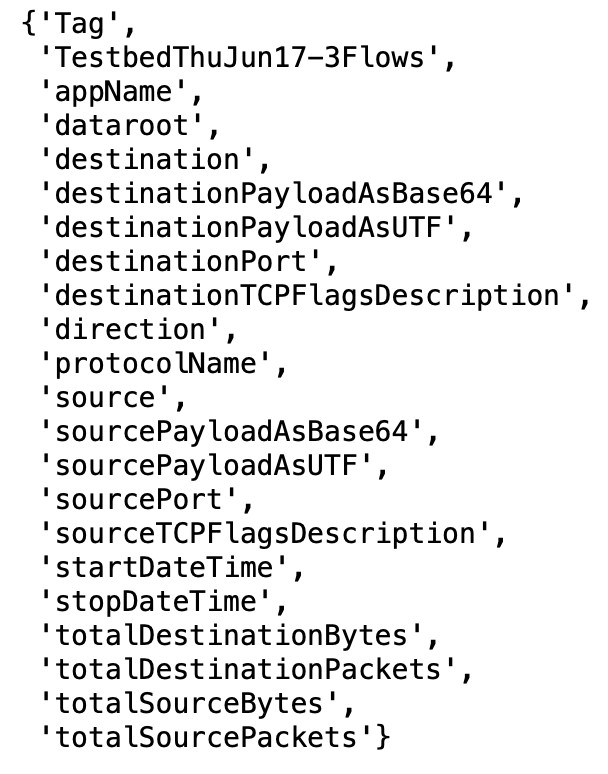


Fig1. Distribution of Normal vs Anomaly samples in ISCXISD 2012 Dataset

a) Features in Dataset

A network data contains a lot of information in it. The parameters present in our data set are as follows:

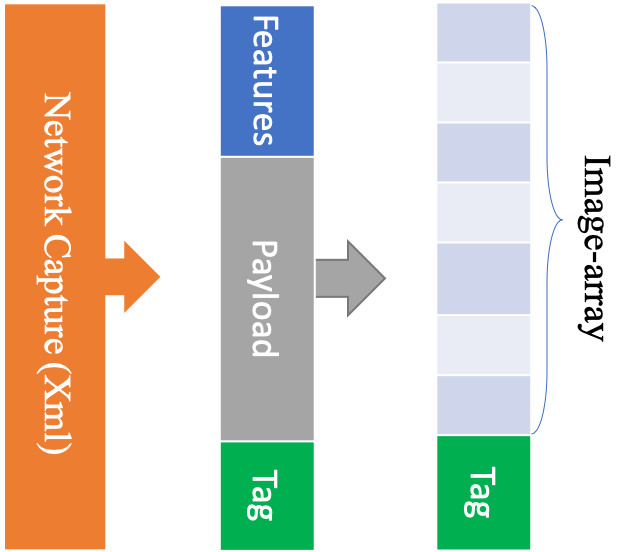


Out of these parameters we decided to choose only one parameter as the feature of our data set because of the limitation on the availability of resources. We decide to make payload as the feature because payload contains the most amount of information in it and computational resources were very limited for us.

b) Dataset Processing

### *Python code file: data\_extract.py*

In order to make payload useful for our model, we had to process it from its raw form. *Figure 2* demonstrates the conversion of raw data.



*Figure 2*

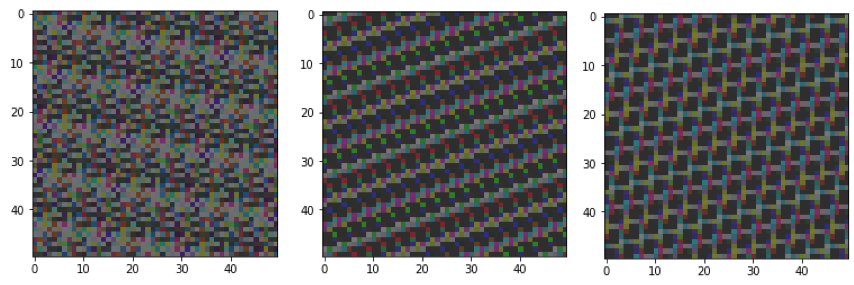
Figure 2

1. Extract the payload from the entire dataset.
2. Convert a payload to a 7500-vector array. (50x50x3)
3. Test-Train Split = 20-80%
4. Labeled-Unlabeled Split = 50-50% of Train Dataset

c) Visualization

The processed data looks like the *Figure 3*. Which shows image arrays of 3 random samples respectively

### Python file: visualization.py



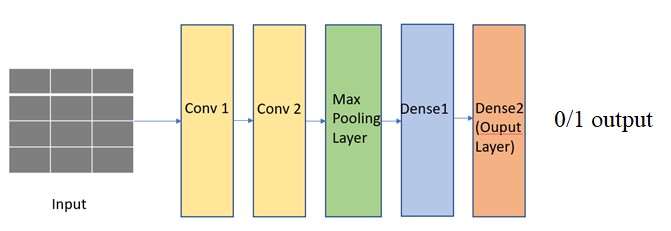
*Figure 3*

The processed data is then fed to a Convolutional Neural Network.

## IV. Models (Convolutional Neural Networks)

**Not so deep, a 4 hidden layer network:**

This is a fresh 4-layer sequential convolutional network with 2 convolution layers with 32 and 64 neurons respectively and 3 x 3 kernel size. We used MaxPooling of 2 x 2 size and 1 stride. On top of that we put a fully connected layer with 128 neurons and an output layer wit sigmoid activation function classifying samples as Normal or Anomaly.



### Fig 7

V. Hyperparameter Tuning and Parallelization:

**Python file for sequential code NotSoDeep: cpdp\_notsodeep\_sequential.py Python file for parallel code NotSoDeep: cpdp\_notsodeep\_parallel.py** Before training the network we need to choose a set of optimal parameters for learning algorithm. There are many parameters that we need to tune. For example lets consider the following parameters with just three options for each.

•Learning Rate [ 0.001, 0.01, 0.1]

•Batch\_size [ 32, 64, 128]

•Optimizer [RMSprop, Adam, SGD]

•Number of Epochs: [ 20, 40, 60]

If we try to search for perfect set from the above examples, we will need to do an exhaustive search with 3 \*3\*3\*3 = 81 combination times full training time. It can take hours to do the exhaustive search sequentially. But since the training for each set is independent of the other we can parallelize the search.

a) Parallelization:

For parallelizing the hyperparameter tuning part we used GridSearchCV from the scikit learn package of python:

### grid = GridSearchCV(estimator=model, param\_grid=param\_grid, n\_jobs=4, cv=2)

where param grid is the list of parameters, n\_jobs = number of concurrent workers and cv = number of cross folds for each set of parameters.

GridSearch CV uses joblib backend which is as follows:

### joblib.parallel\_backend((*backend*, *n\_jobs=-1*[)¶](https://joblib.readthedocs.io/en/latest/parallel.html)

where backend is the kind of backend which can be ‘Multiprocessing and Multithreading’ and n\_jobs = concurrent workers.

We are using multithreading for this project

VI Results without active learning:

### Python file VGG16: cpdp\_final\_model\_vgg16.py Python file NotSoDeep: cpdp\_final\_model\_notsodeep.py

We tried to compare the time taken in hyperparameter tuning when done sequentially and when done in parallel(multithreading).

a) Comparison Results:

For following hyperparameter set (just 6 combinations):

Used Combination:

### Epochs : [20], batch\_size[64,128], Optimizer: [RMSProp], Lr: [0.001, 0.01, 0.1]

|  |  |
| --- | --- |
|  | **TimeTime Taken in NotSoDeepModel** |
| Serial | 9412.142543077469 seconds = ~156 minutes |
| Parallel | 8151.576468465333 seconds = ~129 minutes |

Table 1

The comparison shows that parallelization of exhaustive search of hyperparameters did improve the computational speed.

The next set of results in Table 3 shows the best set of parameters finally used in the models in further experiments as obtained using parallel GridSearch.

b) Best set of Parameters:

|  |  |  |
| --- | --- | --- |
|  | **Options** | **NotSoDeep** |
| Optimizer | [RMSprop, Adadelta,  Adam] | RMSprop |
| LR | [0.001, 0.01, 0.1] | 0.01 |
| Batch\_size | [32, 64, 128] | 64 |
| Epochs | [20, 30] | 20 |

#### Table 2

c) Test Accuracy Results:

**Not so deep CNN:**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Predicted Normal** | **Predicted Anomaly** | Accuracy |
| Actually Normal | 12999 (TN) | 1 (FP) | 87.80 |
| Actually Anomaly | 1813 (FN) | 59 (TP) |

#### Table 5

The next step is to increase the number of true positives and decrease the number of predicted normal. For that Active Learning is used, explained in next section.

## VIII: Active Learning Algorithm

***Python code file: cpdp\_notsodeep.py, least\_confidence\_bias.py, generic\_active\_learner.py***

### Python file: cpdp\_notsodeep.py

Active Learning for class imbalance sensitive problems is a relatively new field in terms of literature & many of the techniques are model specific, for eg. SVM[5]. For our problem, we implemented a binary active learning method called Least Confidence Bias, which is a popular active learner for binary class imbalanced problems. Least Confidence Bias algorithm’s main function biases the classification entropy function of active learning towards picking points which the model is unsure about in classification, but also is a bit more probable to be in the minority class. For example, between 2 unlabeled points such that P(y=0|x) = P(y=1|x) = 0.5 and P(y=0|x) = 0.45 P(y=1|x) = 0.55, the active learner will choose the second as it is more probable to be in a minority class than the first one (we consider y = 1 to be the minority class).

The algorithm for LCB is simple; given an set of unlabeled points X, compute their predicted probabilities P(y=0|x) and P(y=1|x). Then compute the class imbalance in the training set as a fraction as pp (= # of samples with y=1/total # of samples). Finally, we associate a value q with every unlabeled sample x as follows:

If P(y=1|x) < mean( 0.5, pp), Q(x) = P(y=1|x)/ mean( 0.5, pp)

Else Q(x) =(1 - P(y=1|x))/(1 - mean( 0.5, pp))

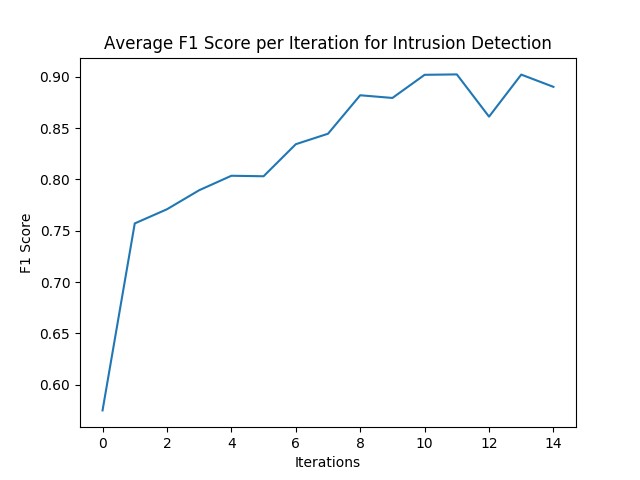
We then simply query x with the largest values of Q(x). The biasing effect of LCB is dependent on the class imbalance of the training set. The behavior of Q value with P(y=1|x) for different imbalance distributions are given below.

As we did see, for every unlabeled point x, we require P(y=1|x), pp to compute Q. None of these depend on the value of any other unlabeled point. This means we can easily parallelize the Q computation by spawning a large number of threads, each given the value of P(y=1|x) and pp, to calculate their corresponding Q(x). In python3, this is done through a library called joblib, which was extremely easy to use and work with in terms of parallelizing tasks.

## X. Results: with Active Learning

### *Python code file: cpdp\_notsodeep.py (for getting results) plotter.py (for visualization)*

Active learning is an iterative procedure, where we pick some particular number of points from unlabeled set and query their label from an annotator. Since doing that for this project was infeasible ( almost no expert annotators for the data + too small timeframe to get annotations). Hence, like most active learning papers, we ran a simulation; we split our labeled dataset into 3 parts: 5000 used as training, 53000 used as “unlabeled set”, and 7100 used for testing. The model will be given only the X features of the unlabeled set and when it queries, the corresponding labels of these features will be given, which along with the queried X values, will be appended to the training set and the model will be retrained. We did active learning for 15 iterations, and for each iteration we picked the top 300 unlabeled points for query. After each iteration, we checked the F1 Score of our model. This experiment is repeated 5 times, and the average values for each iteration is taken. We chose to perform the experiment on the notsodeepmodel. The result, along with the initial and final confusion matrix, is given below:



|  |  |  |
| --- | --- | --- |
| **Confusion Matrix (Initial)** | **Predicted Normal** | **Predicted Anomaly** |
| Actually Normal | 12999 (TN) | 1 (FP) |
| Actually Anomaly | 1813 (FN) | 59 (TP) |

|  |  |  |
| --- | --- | --- |
| **Confusion Matrix (Final)** | **Predicted Normal** | **Predicted Anomaly** |
| Actually Normal | 12993 (TN) | 1 (FP) |
| Actually Anomaly | 316 (FN) | 1556 (TP) |

Furthermore, we also checked the total time taken by sequential active learner and compared it to the parallelized version. The results of that experiment is given below as well:

|  |  |
| --- | --- |
|  | **Time Taken in NotSoDeepModel** |
| Serial | 13.78 minutes |
| Parallel | 13.36 minutes |

The time taken by both serial and parallel version of our active learner code appears to be the same. This while a bit disappointing, may be explained by the fact that training neural networks itself is a very heavy task, and takes up a large amount of memory and processing power, thereby bottlenecking the whole application. In such a case, the speedup of using a parallel algorithm to compute q over for loop over 50000 points will be negligible.