Group 29 Progress Report:
IoT Attack Detection Machine Learning Model

Jacob Brodersen, Daniel Maurer, Olivia Reich
{brodersj, maured2, reicho}@mcmaster.ca

1 Introduction

The Internet of Things (IoT) encapsulates a network of numerous interconnected devices and technologies, facilitating real-time communication and data sharing. In this field, a major concern is the identification of malicious network traffic, an issue only growing in scale as time goes on. IoT connected devices are becoming extremely popular with a predicted 29 billion IoT connected devices by 2030 (Katigbak, 2025).However, due to network complexities as well as the diverse technologies and communication channels integrated through the system, there are many possible areas of weakness that leave IoT devices susceptible to cyber-attacks and malicious activity. Almost all IoT systems are at risk of attack, regardless of the active security measures put into place.

As a means of risk mitigation, this project aims to classify network traffic as one of 12 different classes including 9 malicious behaviors and 3 benign behaviors. Such a classifier can be used within Intrusion Detection Systems (IDSs), resulting in more robust and adaptive security for IoT networks (Sharmila and Nagapadma, 2024). Many such IoT network monitoring applications exist, either as software bundled with the purchase of a related IoT device, or as standalone applications. These applications tend to focus on not only security, but other important statistics including device performance, health, and availability (Cooper, 2025). Our solution will focus purely on identifying and intercepting possible attacks.

2 Related Work

We have identified two main papers that aim to complete the same task as our project. Saba et.al focus on applying Convolutional Neural Networks (CNNs) to anomaly-based intrusion detection systems, arguing that deep learning's pattern matching abilities are superior to the more traditional machine learning techniques used to find anomalies in IoT infrastructure (Saba et al., 2024). Sharmila et.al, the creators of the dataset we plan to employ, utilize Vector-quantized Variational Auto-encoders (VQAEs) as their model of choice. The major argument they make against the above approach is incompatibility with the limited processing power of most IoT devices. They propose QAEs as a superior model, also utilizing an anomaly-based detection approach (Sharmila and Nagapadma, 2023). Rahman et.al work on a similar problem in the field of IoT – device classification, a task becoming ever-difficult as IoT device popularity grows and an increasing number of devices are created. The group created an extensive new data set and evaluated various models, with Random Forest performing the best (Rahman et al., 2025). Two other papers that work on similar problems in classifying general internet traffic are discussed as follows. Fesl et.al work on classifying both the protocol and content of VPN traffic, experimenting with Recurrent Neural Networks (RNNs) amongst other methods to classify packets (Fesl and Naas, 2025). They experience some success, but Random Forest still outperforms on their dataset; this may or may not indicate their success with classifying IoT traffic. Lastly, Latif-Martinez et.al produce a novel approach for anomaly detection in general network monitoring with a modified version of Graph Neural Networks (GNNs), which exhibits considerable performance increases on benchmark data sets (Latif-Martínez et al., 2025).

3 Dataset

3.1 Original Dataset

INSERT SECTION

3.2 Preprocessing Implementation

INSERT SECTION

3.3 Preprocessed Dataset

INSERT SECTION

4 Features

Describe any features you used for your model, or how your data was input to your model. Are you doing feature engineering or feature selection? Are you learning embeddings? Is it all part of one neural network? Refer to item 2. This may range from 0.25 pages to 0.5 pages.

INSERT SECTION

5 Implementation

The current model implementation is a standard neural network with no recurrent or convolutional layers. Defined using PyTorch, the model was designed to be as general as possible to facilitate the eventual tuning of hyperparameters beyond learning rate or regularization strength, as we also wish to experiment with network width and layer depth. To create a model instance, the input dimension (e.g. the number of features), output dimension (e.g. number of classes), and dimensions of all hidden layers within a list are taken as inputs. Note that the number of layers is also fully customizable, reflecting how long the list is. A sequential neural network is then created with these parameters, with the activation function between each layer chosen as ReLU due to its simplicity and efficiency. The raw output of the model, given a singular instance's features, is a vector with length reflecting the number of classes, with the most positive argument being the model's prediction.

The chosen loss function for the model is Cross Entropy loss, selected due to its ability to penalize incorrect predictions stronger in comparison to a loss function such as MSE. This loss is evaluated on each iteration's batch, with this loss used to calculate the gradient that updates the model's parameters. The full cross-entropy loss across all instances is also calculated every few iterations but does not play a role in updating parameters. Within PyTorch, this function implicitly calculates the softmax on input vectors, hence why this layer is left out of the model. With similar reasoning, to compare model outputs to target labels when calculating the F1-score, both the model output and target label vectors are run through the argmax function to convert to a single integer reflecting the classes. Similarly, the F1-score is also calculated every few iterations, but its value does not influence updates to model parameters.

To optimize, we implemented mini-batch stochastic gradient descent, both to reduce the computational load of training the model on ~120,000 instances at once and to take advantage of its ability to escape local minimums. Batch size is also configurable, allowing it to be tuned as a hyperparameter. To ensure randomness in batch selection, the training data is shuffled once per epoch, ensuring that possible bias from ordering of instances is reduced. The SGD optimizer is currently utilized due to both its relative simplicity and the scale of the dataset being relatively small. However, we may investigate using the Adam optimizer in future iterations, leveraging its ability to increase model convergence rates by incorporating momentum and adaptive learn rates. Currently, no methods of regularization (L1, L2, or dropout) are implemented as the model does not appear to rapidly overfit, however the model has been designed such that this functionality can be easily implemented in future iterations.

In total, the list of model hyperparameters available to be experimented with are as follows: learning rate, number of layers, depth of each layer, batch size, and number of iterations. Within future development of this project, we may look into implementing additional forms of hyperparameter tuning with both methods of regularization and early stopping. If that is the case, regularization strength, dropout probability, and patience will be additional hyperparameters.

For each specified hyperparameters, a specific range of hyperparameter values was chosen as possible options. The itertools import from Python was then used to create a Cartesian product that contained all possible combinations of the possible hyperparameter values. A random sample of 50 combinations was then chosen in order to conduct random search. Due to the large amount of data and high computational demands of training the model, random search was chosen over grid search to avoid excessively long and demanding computations during hyperparameter tuning.

Our primary baseline is a simple linear regression model, with an identical loss function and optimization method as described above. Outperforming this baseline would demonstrate that the added complexity of multiple non-linear layers is necessary for accurately solving the problem.

6 Results and Evaluation

6.1 Method of Evaluation

Currently, the data is partitioned such that approximately 80% is used for training, 10% for hyperparameter tuning via validation, and the remaining 10% is reserved as a held-out test set. This test set is not part of the training and instead is used to simulate our model's effectiveness on real-world data, as this data has not been exposed during the training process and only tests on our complete model after hyperparameter tuning. Due to the large amount of data within this dataset, the need for k-fold validation wasn't as imperative, as 98,000 data points proved to be enough while executing model training. Similarly, ~12,000 - 13,000 data points are sufficient for both hyperparameter tuning as well as verification that the model executes to an acceptable degree on real-world data.

During hyperparameter tuning, there were 3 batch options, 4 learning rate options, 4 iteration count options and 20 hidden structure options chosen at random, each containing 2–4 layers, with layer sizes selected from six possible depth configurations. The best results from all options within batch size, learning rate and iteration count were showcased on a graph. Due to the expansive list of possible hidden structures, it is not evaluated on its own graph but simply noted as one of the parameters used to generate the best results in the other hyperparameter graphs in the legend.

F-scores was compared against the results from the baseline model. As mentioned above, the baseline model is a simple linear regression model. Hyperparameters are set to reasonable values that are estimated to provide stable training given the attributes of the dataset. The set values for the baseline model have been provided below.

batch size: 64
learning rate: 0.1
max iteration count: 10000

6.2 Results

After hyperparameter tuning, the following parameter values were deemed as optimal:

number of layers: 4
depth of first layer: 64
depth of second layer: 64
depth of third layer: 32
depth of fourth layer: 16
batch size: 128
learning rate: 0.1
max iteration count: 50000

After applying these hyperparameters and training the model, the following results were concluded:

f-score on validation data: 0.9453
f-score on baseline model: 0.8956
f-score on testing data: 0.9477

The graphs generated for the above results can be found in Figures 1 through 12 in the appendix.

One notable consideration when viewing the graphs generated by the program is how final values are shown when there are different iteration counts. Certain lines have shorter iteration counters and therefore stop earlier in the graph. To provide better visualization in regard to how the final values compare to each other, the final value that is noted at the last iteration is extended at a flat rate to the iteration value of 50000.

6.3 Evaluation of Results

When comparing validation and testing f-scores, the difference between the two is less than 1%. This suggests that the model is generalizing well, as there are no signs of overfitting the data.

Overall, both validation and testing show improvement over the baseline f-score by approximately 5%. This indicates that the hyperparameter tuning helps improve the model. However, the high baseline value may indicate dataset issues that should be further explored to rule out. Some of these issues may include class imbalance, low variability among samples, or an insufficiently diverse data distribution. This will likely be further explored in future development.

7 Feedback and Plans

Initially, we were planning on implementing a convolutional neural network (CNN) as the authors of the dataset used or a recurrent neural network (RNN) due to the sequential nature of our data. However, our TA suggested that before we try those, we should try a basic neural network as they can also be successful for our type of task. In the future, we plan to try and implement our initial idea of an RNN and CNN neural network to see how they compare to the basic neural network we have implemented.

Another piece of feedback we received was based on our feature engineering process. We were unsure of how to find unnecessary features in our dataset. Our TA said that we should first try not removing any features as neural networks can often handle the features themselves. However, if we did need to remove features, checking the correlation between features would be useful. For now, as suggested, we have just used all of the features (excluding the four we discussed in Features). In the future, we plan to use the correlation between features to remove some features and test whether only keeping one of highly correlated features has any positive impact on the model performance. Another method we may try in the future that the TA recommended was plotting the linear regression of each feature vs. the target and using the best ones to narrow down the feature set.

A third piece of feedback we received was related to how to determine the number and size of layers we should have in our neural network. The TA suggested experimenting with different sizes and number of layers using hyperparameter searching to find what works best. We have implemented this as described in Results and Evaluation.

After initial evaluation of our model and baseline, we have seen very promising performance. One main concern we have, however, is bias in the dataset making it easier to predict correctly. Namely, one target class makes up 76.9% of the dataset, and as such, if the model is in doubt, it can choose this class and be right much more frequently than if the classes were uniformly distributed. As a next step, we plan to remove instances of the dataset to get a more uniform distribution of target classes to see how that affects the models' performance.

Team Contributions

7.1 Jacob Brodersen - Preprocessing

Key Contribution Area: Preprocessing

Contribution Summary: In terms of the code, I worked on the preprocessing, profiling of the dataset, and workflow / script layout. For the report, I worked on Dataset, Features, and Feedback and Plans.

7.2 Daniel Maurer

Key Contribution Area: Model & Training Loop Definition

Contribution Summary: For the code, I created the model definition, as well as the training loop. For the report, I wrote the Model Implementation, Introduction, and Related Work.

7.3 Olivia Reich

Key Contribution Area: Model & Training Loop Definition

Contribution Summary: For the code, I implemented training with hyperparameter tuning, testing and graph visualization . For the report, I wrote a paragraph in the Model Implementation regarding justification for the chosen hyperparameter tuning method of random search as well as Evaluation & Results, the Appendix, and LaTeX formatting.

References

S. Cooper. 2025. The best iot device monitoring tools. Accessed 7 October 2025.

J. Fesl and M. Naas. 2025. A comprehensive machine learning-based approach for virtual private network traffic detection, classification and hiding. *Computer Networks*, 270.

R. Katigbak. 2025. The economy of things: The next value lever for telcos. Accessed 7 October 2025.

H. Latif-Martínez, J. Suárez-Varela, A. Cabellos-Aparicio, and P. Barlet-Ros. 2025. Gat-ad: Graph attention networks for contextual anomaly detection in network monitoring. *Computers & Industrial Engineering*, 200.

M. M. Rahman, F. Bouhafs, S. A. Hoseini, and F. den Hartog. 2025. Unsw homenet: A network traffic flow dataset for ai-based smart home device classification. *Computers & Industrial Engineering*, 204.

T. Saba, A. Rehman, T. Sadad, H. Kolivand, and S. A. Bahaj. 2024. Anomaly-based intrusion detection system for iot networks through deep learning model. *Journal of Network and Computer Applications*.

B. Sharmila and R. Nagapadma. 2023. Quantized autoencoder (qae) intrusion detection system for anomaly detection in resource-constrained iot devices using rt-iot2022 dataset. *Cybersecurity*, 6:1–15.

B. S. Sharmila and R. Nagapadma. 2024. Rt-iot2022 dataset. Accessed 27 September 2025.

8 Appendix

Figures 1 through 4 present top-performing model configurations identified at each tested batch size: 64, 128, and 256.

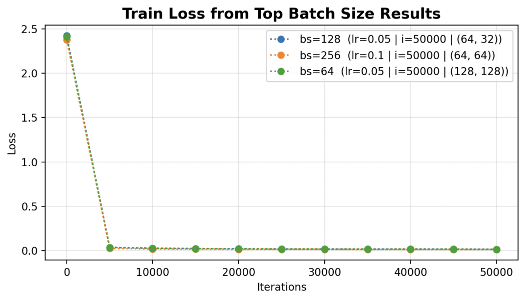

Figure 1: Training loss for top-performing model configurations across batch sizes.

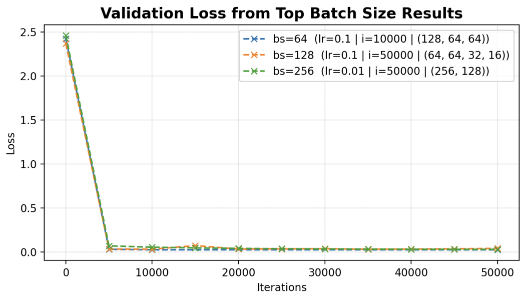

Figure 2: Value loss for top-performing model configurations across batch sizes.

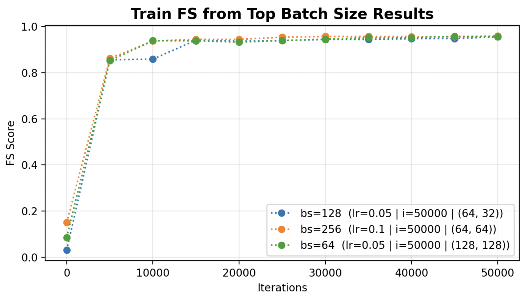

Figure 3: Training FS score for top-performing model configurations across batch sizes.

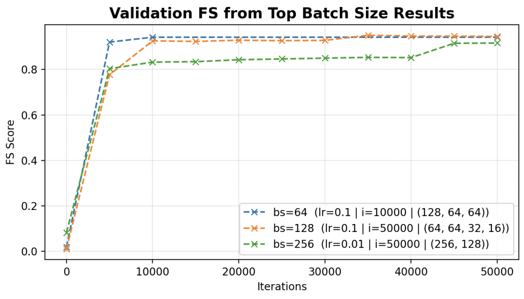

Figure 4: Validation FS score for top-performing model configurations across batch sizes.

Figures 5 through 8 present top-performing model configurations identified at each learning rate: 1e-1, 5e-2, 1e-2, 5e-3

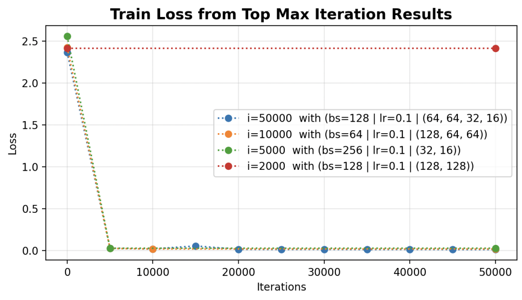

Figure 5: Training loss for top-performing model configurations across learning rates.

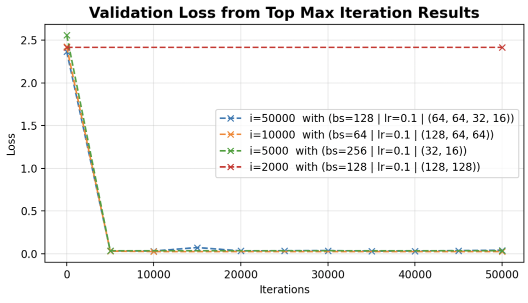

Figure 6: Value loss for top-performing model configurations across learning rates.

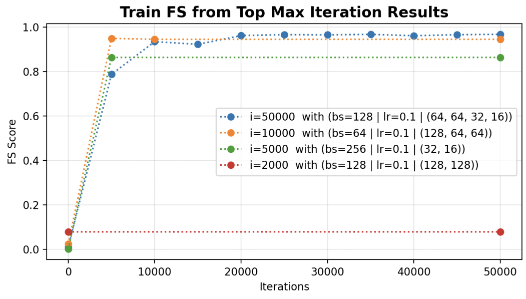

Figure 7: Training FS score for top-performing model configurations across learning rates.

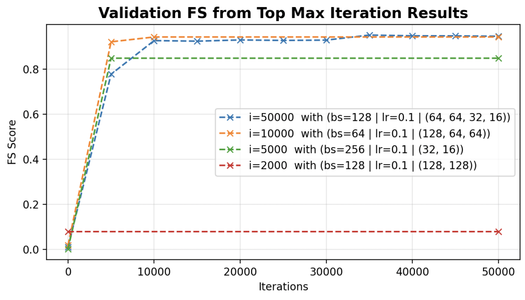

Figure 8: Validation FS score for top-performing model configurations across learning rates.

Figure 9 through 12 present top-performing model configurations identified at each maximum iteration count: 2000, 5000, 10000, 50000

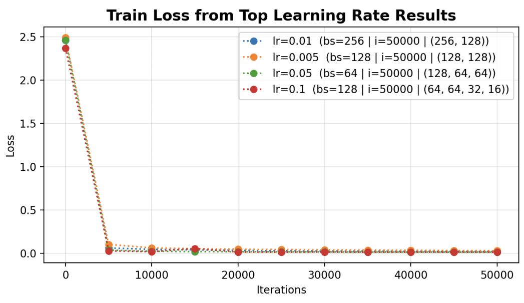

Figure 9: Training loss for top-performing model configurations across max iteration counts.

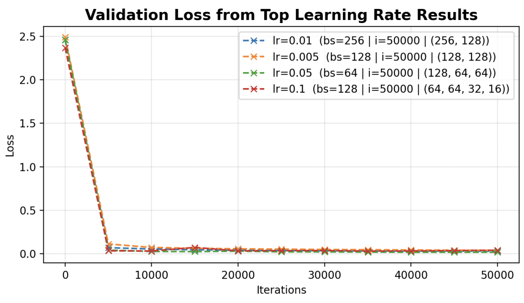

Figure 10: Value loss for top-performing model configurations across max iteration counts.

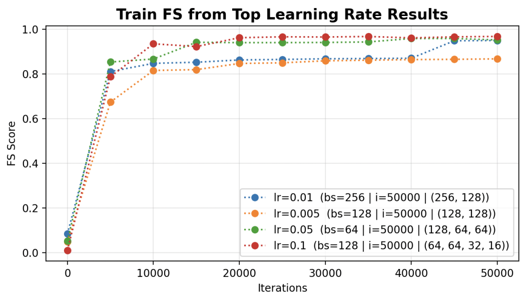

Figure 11: Training FS score for top-performing model configurations across max iteration counts.

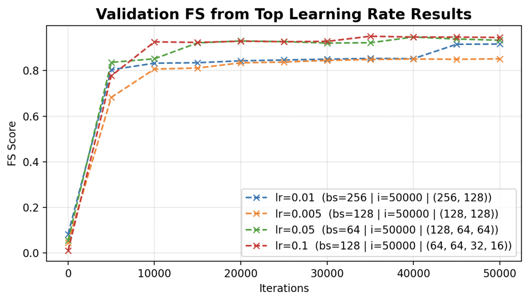

Figure 12: Validation FS score for top-performing model configurations across max iteration counts.