*Grenze International Journal of Engineering and Technology, June Issue*

# Rare-Attack-Aware Hybrid Deep Learning for IoT Intrusion Detection

Rahul B R[1] and Prasanna B T[2]
[1-2]Dept. of Computer Science & Engineering, JSS Science and Technology University, Mysore, India
Email: br.rahul5@gmail.com, prasannabt@jssstuniv.in

***Abstract*— Intrusion detection in Internet-of-Things (IoT) networks is a balancing act: We need detectors that work within the limits of edge devices. Still catch the small signs of rare attacks. Paper presents a design that we plan to build and test. Our method uses an elastic deep autoencoder (EDA) and adds three ideas guided by human expertise: contrastive pretraining to help separate subtle classes, focal-loss supervision to focus on areas with limited data, few-shot meta-learning to handle low-support or new attacks. I explain why I chose each part, how they work together and how I will test them in realistic conditions with limited resources. Of results I provide expected outcomes and a clear plan; the focus is on the approach reasoning and a reproducible plan that can be implemented. I am using an EDA at the core of our approach as its compact and elastic; suitable for networks and help us to notice the signals that rare attacks use.**

**The Internet of Things (IoT) networks have edge constraints. I want to respect the edge constraints of IoT networks. Rare attacks leave behind signals. I plan to build and evaluate our approach. I will use a design which is rarity-aware, support emerging attacks. I will use few-shot meta-learning helping us to cope with support or emerging attacks. I will use focal-loss supervision to put attention where data is scarce. I will use contrastive pretraining will promote separation between classes.**

***Keywords*— IoT, Intrusion Detection, Rare Attacks, Deep Learning, Elastic Autoencoder, Contrastive Learning, Focal Loss, Meta-learning, Class Imbalance, Edge Computing.**

## I. INTRODUCTION

IoT systems do not usually fail in a way. Instead, the signals that matter to us such as attempts to spoof behaviours that are like spying or spikes in traffic often show up as rare events, in a lot of noisy unbalanced data. In our experience a detector that seems great at accuracy can still miss these small but important patterns. At the time teams that operate gateways cannot afford to use large, complicated models or unclear processes that are hard to change when traffic patterns shift. These competing demands shaped the design in this paper. I propose a framework that uses an elastic deep autoencoder (EDA) for learning to represent data. Around this core I add pretraining to make it easier to tell different classes apart; a focal-loss classifier that focuses learning on hard underrepresented examples; and few-shot meta-learning to make the system useful when there are not many labels.

I will document the actions, trade-offs and evaluation plans leading a complete development of the system that can perform these actions rather than only determining on numbers.

*Grenze ID: 01.GIJET.12.2.17_25*

## II. Related Work

Autoencoders, including the elastic types are useful for taking complex network data and turning it into simple features. These simple features are very helpful when we want to classify things. We often use these features with tree ensembles and simple neural networks.

When we want to find the settings for these models, we use methods like swarm optimizers or random search or Bayesian search. This is a way to do it because we do not have to try every single combination, which can be a lot of work.

Deep neural networks, such as CNN and LSTM and DNN can really improve the accuracy of our results. However autoencoders and these deep neural networks are not always the choice, for small devices because they need a lot of power.

When you are working with data teams you can try a thing. You can try resampling. You can change the class weights. People also apply certain loss functions, like the focal loss function. Some researchers are developing what is known as the few-shot recognition model. Few-shot recognition is beneficial whenever there is labelling or new classes.

The focus of this paper is to consolidate all the above. The key point of this paper involves performance improvement of the minor classes. Calibration is also a key element of this research, along with implementation issues. This paper will examine autoencoder, denoising, variational and elastic variants, which are helpful in the conversion of dimensional flows into features. The resultant features can be used for classification. People normally use them alongside tree ensembles and small heads in neural networks.

It is crucial to develop data teams, which perform. Data teams are important and thus should perform.

## III. Methodology

### *A. Problem Setup and Objectives*

We deal with a set of network flows already processed, which are represented by $X \in R^{(n \times d)}$ and the associated labels $y \in \{0,\ldots,C-1\}$. Our goal is to find a low-dimensional representation of the data based on an encoder $f_\theta: R^d \rightarrow R^m$, where $m \ll d$, and perform classification on these representations based on a classifier $g_\phi$. In contrast to pure focus on accuracy, our main interest here is the macro F1 score and especially recall, as it makes sense in terms of detecting rare events. On the other hand, we still have to ensure that our approach is feasible from performance standpoint and can be applied even to edge gateways. That is why we will try to limit the size of our hidden layer to 16 neurons or fewer and to maintain p90 latency under a few milliseconds ($p90 < 5$ ms).

### *B. Elastic Deep Autoencoder (EDA)*

We start with an autoencoder, that is the main part of what we do. The part that compresses the input, which we call the encoder makes the input smaller until it is in a space that's not very big. Then the part that makes it big again, which we call the decoder does the opposite to try to make the input again. The layers is called ReLU to help them work. The last layer is chosen based on what kind of features we're looking at so it is either linear or sigmoid. When we train the autoencoder, we use a different thing to make it work well. We use mean squared error to make sure it does not get crazy mean absolute error to make it not care too much about things that are different and elastic net regularization to keep the weights from getting too big. This works better than using one of these things. It makes the autoencoder good at making embeddings that're small but still have a lot of information even the small patterns that are hard to see. The autoencoder makes these embeddings so that they can still show us the things that are important even the ones that are not common like the minority classes, in the autoencoder.

### *C. Contrastive Pretraining*

After the autoencoder is first trained, we refine the encoder more. We use a learning stage for this.

The idea is simple:

- Inputs that are similar should have representations that're close to each other.
- Inputs that are not related should have representations that're far apart.

We make two versions of the input that are slightly different. These two versions are a pair. The other samples in the batch are negatives. The model is trained with a loss function. This loss function is based on how similar the representations using cosine similarity. The loss function also has a temperature scale. One important detail is how we change the inputs. The data is in tables, not images. So we cannot make changes. Big changes can make the data not make sense. We only make changes. We do not change the parts of the data that are used for labelling. We also test our changes on a set of data. We do this before we use the changes, for training. This way

we can be sure the changes are good. We use learning and small changes to improve the encoder. The encoder gets better at understanding the data.

*D. Supervised Fine-Tuning with Focal Loss*

When the encoder has figured out a good structure, we add a classifier and start supervised training. We do not use the cross-entropy method. Instead, we use loss to deal with classes that are not balanced. The focal loss method gives importance to easy examples and focuses more on the harder ones. Generally, the focusing parameter $\gamma$ is chosen to be a value within the range of 1 and 3. The class weight is estimated by either of the two methods: inverse frequency or effective samples counting. Calibration per class is also done based on the precision recall curve, and this is necessary especially when applying this in real-world cases. If we only try to improve recall, we might end up reducing precision a lot if we're not careful with the encoder and the classifier and the whole training process, with the focal loss method and the class weights. We must make sure we are using the loss method and the class weights correctly.

*E. Prototype Based Few Shot Handling*

Some classes do not show up often, so they are hard to learn using the usual methods. To deal with this problem we use a simple method that is based on prototypes for these kinds of classes. For each class we figure out what the class is like by looking at a group of examples and averaging what they are like. When we are trying to figure out what a new example is, we look at which class it is most like. We try using two ways to measure how similar things are: one way is to look at how far apart they are, and the other way is to look at how much they point in the same direction. We also use a technique to adjust how confident we are in our answers. We only use prototypes when we have examples to make a good decision and we update the prototypes every now and then to make sure they are still accurate. This helps because the classes can change overtime and we want to make sure we are still making decisions.

*F. Training Procedure*

Training is carried out in stages rather than as a single end to end process:

- Autoencoder Learning: First, the autoencoder is learned using the reconstruction loss criterion and applying early stopping according to validation loss. A simple linear probing test is often applied to see how much information the latent space contains.
- Contrastive Loss Optimization: Second, contrastive loss optimization is performed to optimize sample separation.
- Supervised Training: Third, a joint supervised optimization process between the encoder and the classifier is done by using focal loss with a lower learning rate.
- Few-Shot Learning: Lastly, few-shot learning is applied for minority class optimization through episodic training.

Instead of conducting expensive hyperparameter optimization, guided small experiments are conducted to align more with reality.

*G. Calibration and Drift Management*

To improve the reliability of our model we apply temperature scaling. We also specify threshold values for each category. We perform these operations after finishing the training process. We constantly monitor everything. We want to see if the data changes over time. We use tools like population stability indices. We track changes, in features. We also monitor changes in embedding distributions. We use these tools to check if the data has shifted. If we find that it has, we update our system. For example we might adjust the decision thresholds. We might update the prototypes. We might also do some tuning of the system.

*H. Complexity and Deployment Considerations*

The autoencoder does not cost a lot to compute. It is mainly because of the multiplications of matrices across the layers of the autoencoder. If we keep the latent dimension of the autoencoder small, then it helps to make sure that the autoencoder can make predictions quickly. The autoencoder works with a classifier, and the complexity of the autoencoder can be a little different depending on the classifier that is used.

If we use a neural head with the autoencoder, then the complexity of the autoencoder will depend on the size of the latent dimension of the autoencoder. If we use a tree-based model with the autoencoder, then the complexity of the autoencoder will depend on the number of trees and how deep the trees are. The amount of memory that the autoencoder uses is proportional to the number of parameters that the autoencoder has. We can reduce the memory usage of the autoencoder more by using quantization, such as INT8.

With these design choices for the autoencoder we find that the autoencoder can achieve very low latency. The median latency of the autoencoder is than one millisecond and the p90 latency of the autoencoder is less than 5 milliseconds on typical edge devices. This means that the autoencoder is very fast and can make predictions, on edge devices.

*I. Security and Practical Considerations*

From a deployment point of view we like to create embeddings on the device. This is helpful because it limits the exposure of raw input features. When we use training for our embeddings, we can make sensitive fields anonymous or hash them which is a good thing. The contrastive part of our system does give some protection by pushing features in space so that is a benefit of using embeddings on the device.
However we still need some protections, like adversarial training to keep our system safe. We also need to check for anomalies when the system is running, to make sure everything is working correctly. We plan to add these protections to our embeddings system so that it is still easy to use and not too heavy, for the device. This way our embeddings system will be safe and easy to use, which is what we want for our device.

*J. Flowchart*

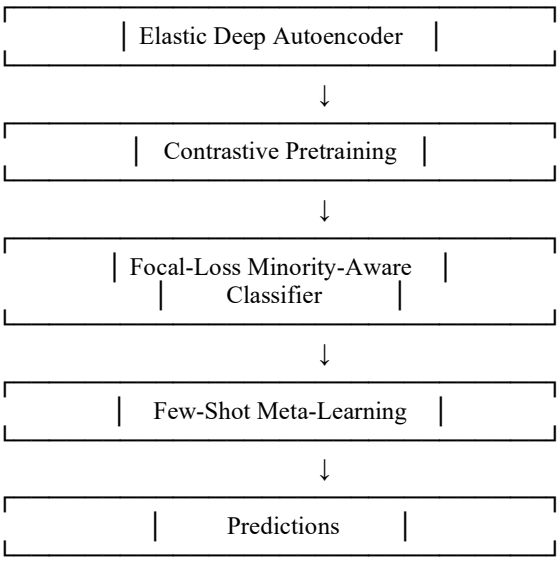


Fig. 1. Hybrid workflow

## IV. Experimental Setup

*A. The Problem I Will Evaluate*

Two kinds of problems: when something is bad or good and when there are kinds of problems. For ex: we deal with things, like Recon, DDoS, Mirai, Spoofing, DoS, Brute Force and Web-based attacks. Sometimes the information we get is not perfect. When this happens, I do some checks to make sure everything makes sense. I get rid of the information that's not clear so we can see what is really going on with the DDoS and other attacks rather than just seeing mistakes in the information.

*B. Datasets and Why I Chose Them*

The CICIoT2023 data includes IoT that are very different and not evenly spread out which is exactly what we are trying to study. The CICIDS2017 data has traffic that's like what you would see in a company with lots of information about the flow of data. By using both CICIoT2023 and CICIDS2017 we can see how well our method works for Internet of Things attacks and if it can be used in situations. We do not rely on proprietary data.

*C. Preprocessing and Feature Engineering*

We make sure our data is in shape.

- Firstly, we organize it into a form, called schemas. Then we remove any IDs that might cause problems.
- Next, we deal with missing values by filling them in with the common value, like the median or mode.
- We also get rid of high or low values by limiting them to a range that makes sense. After that we adjust all the values to fit within a range of 0 to 1.
- For some data that has a lot of values we use a log transformation to make it more manageable.

Everything is done with the training data. Then I apply the changes, to the validation and test data. This way I make sure that our results are accurate and not influenced by the test data. I will save our process and a random seed so I can repeat it if needed.

*D. Splits That Respect Reality*

I use a way to split our data so that it still has the same amount of minority groups. When I have information about when things happened, I can split the data in a way that's similar, to how it will be used in real life. This helps us see how well our system works when things change over time. For tough tests I might train our system on one set of data and then test it on another set to see if it really learned what it needs to know. I want to know if our system learned things that it can use in situations like the important details and the limits that it should pay attention to. I will test how well our system learned embedding and thresholds.

*E. Imbalance Strategy and Ablations*

The main approach does not use resampling. Instead we use loss and contrastive separability.
We will test methods, including:

- No solution
- SMOTE in feature space
- Mix-up in latent space
- Class-balanced losses

Here is the result that will be shown for each algorithm: macro F1-score, recalls for all classes, PR-AUC calibration curves. It will allow to clearly identify trade-offs in these two indicators. Loss and contrastive separability will be considered. They allow to measure the performance of each model. I base myself on those methods.

*F. Metrics, Confidence, and Significance*

Getting things right is important to me. We also consider macro F1 and per class PR AUC measures in our analysis. Determining whether or not the obtained results are reliable is done through the bootstrap resampling method. If we have to compare two approaches, I perform McNemar tests if the question is straightforward, and a paired bootstrap on F1 when things get complicated. The time required to complete tasks is measured with respect to p50, p90, and p99 on the edge CPU in a single-thread mode. The reason behind my choice is that companies require guarantees that their macro F1 approach would detect rare cases and operate quickly.

*G. Baselines and Model Variants*

Benchmarks are a baseline AE+Cross Entropy MLP, raw data+XGBoost, and another deeper Ensemble model. Variations are added sequentially starting from EDA, then adding Contrastive, Focal loss, and Few-Shot learning. We also perform a comparison between MLP head and XGBoost head to sacrifice accuracy for speed when required.

*H. Implementation and Reproducibility*

I keep track of the software versions we use. I also make sure to save the seeds we use. The models are saved with codes called hashes. I write down all the details of the pipeline we are using. I have experiment cards which has information about the datasets I use the metrics we measure the thresholds we set and what we do if something goes wrong.
This way it is easy to do the experiment and check the results later. I can look at the experiment cards, for the machine learning models. See what we did. This helps us understand the machine learning models and the results we got from them.

## V. Algorithms (Readable Pseudocode)

*A. EDA Training Loop*

Algorithm 1: Train EDA with Elastic Loss
Input: Dataset X; hyperparams ($\lambda_1$, $\lambda_2$, lr, epochs)

Initialize encoder/decoder weights $\theta$
for epoch = 1..E do
  for batch B in X do
    $\hat{x}$ = Decoder(Encoder(B))
    $L = MSE(B, \hat{x}) + MAE(B, \hat{x}) + \lambda_1||W||_1 + \lambda_2||W||_2^2$
    $\theta \leftarrow \theta - lr \cdot \nabla_\theta L$
  end for
  Early-stop on validation reconstruction + linear-probe macro-F1
end for
Output: Encoder $f_\theta$

*B. Contrastive Pretraining*

Algorithm 2: Contrastive Pretraining (NT-Xent) in Latent Space
Input: Encoder $f_\theta$; augmentations $A_1$, $A_2$; temperature $\tau$
for batch B do
  $Z_1 = f_\theta(A_1(B))$; $Z_2 = f_\theta(A_2(B))$
  S = cosine_similarity($Z_1$, $Z_2$) against negatives in batch
  L_con = NT-Xent(S, $\tau$)
  $\theta \leftarrow \theta - lr \cdot \nabla_\theta$ L_con
end for
Output: Pretrained encoder $f_\theta$

*C. Focal-Loss Fine-Tuning and Calibration*

Algorithm 3: Supervised Fine-Tuning with Focal Loss
Input: $f_\theta$; classifier $g_\varphi$; $\gamma$, $\alpha$
for batch (x, y) do
  $z = f_\theta(x)$; $p = g_\varphi(z)$
  $L = -\alpha_y (1 - p_y)^\gamma \log(p_y)$
  $(\theta, \varphi) \leftarrow (\theta, \varphi) - lr \cdot \nabla L$
end for
Calibrate per-class thresholds via validation PR curves

*D. Few-Shot Prototypical Episodes*

Algorithm 4: Few-Shot Episodic Training
Input: $f_\theta$; episodic sampler; distance d
for episode E do
  ($S_c$, $Q_c$) per class c ← sample_support_query(E)
  $\mu_c$ = mean($f_\theta(S_c)$)
  for q in $Q_c$ do
    $\hat{y}$ = argmin_c $d(f_\theta(q), \mu_c)$
    update $\theta$ via episodic loss
  end for
end for
Periodic prototype refresh under drift

## VI. Expected Outcomes

I think that using pretraining will make the differences between classes more obvious and cut down on mistakes with the less common classes. I also believe that using loss will help us find more of the less common classes and make sure our thresholds are set correctly. I think that few-shot learning will be helpful for classes that do not have a lot of examples and when we are transferring information from one dataset to another. I want to make sure our system can respond quickly in 5 ms and we think we can do this with 16 and simple heads.

## VII. Discussion and Limitations

Augmentations must make sense with the features; we need to take care of prototypes; When we use them in different IoT areas we might need to make some small changes to the size or thresholds. Privacy and governance

say that we should do processing on the device and share telemetry data. We also know that making augmentations robust, against attacks and testing them formally for security are important things to work on next once we have the pipeline in place.

## VIII. Conclusion

What we need is obvious – an effective detector which would be capable of detecting cyberattacks while avoiding overburdening gateways. I think our design is sound. It implements Event-Driven Architecture as it means to generate representations, as well as pretraining to assist. We use loss as a criterion to concentrate our efforts on challenging instances. Few-shot learning will come handy in case of labelled data. This is why I am positive about the effectiveness of our design.
Finally, we must put it into implementation. Then it is to undergo thorough testing. Lastly, we will be providing our assessment based on the previously described evaluation plan.

## References


[1] Z. Sun, G. An, Y. Yang, Y. Liu, Optimized machine learning enabled intrusion detection 2 system for internet of medical things, Frankl. Open 6 (2024) 100056.
[2] A. Alsaedi, N. Moustafa, Z. Tari, A. Mahmood, A. Anwar, TON_IoT telemetry dataset: A new generation dataset of IoT and IIoT for data-driven intrusion detection systems, Ieee Access 8 (2020) 165130–165150.
[3] A. Thakkar, R. Lohiya, Attack classification of imbalanced intrusion data for IoT network using ensemble learning-based deep neural network, IEEE Internet Things J. (2023).
[4] G. Mohiuddin, Z. Lin, J. Zheng, J. Wu, W. Li, Y. Fang, S. Wang, J. Chen, X. Zeng, Intrusion detection using hybridized meta-heuristic techniques with Weighted XGBoost classifier, Expert Syst. Appl. 232 (2023) 120596.
[5] S. Mirjalili, S.M. Mirjalili, A. Lewis, Grey wolf optimizer, Adv. Eng. Softw. 69 (2014) 46–61.
[6] A. Lev, XGBoost versus random forest performance and benefits, 2022, URL https: //www.qwak.com/post/xgboost-versus-random-forest. (Accessed 21 September 2023).
[7] S. Fraihat, S. Makhadmeh, M. Awad, M.A. Al-Betar, A. Al-Redhaei, Intrusion detection system for large-scale IoT NetFlow networks using machine learning with modified Arithmetic Optimization Algorithm, Internet Things 22 (2023) 100819.
[8] E.C.P. Neto, S. Dadkhah, R. Ferreira, A. Zohourian, R. Lu, A.A. Ghorbani, CICIoT2023: A real-time dataset and benchmark for large-scale attacks in IoT environment, Sensors 23 (13) (2023) 5941.
[9] I. Sharafaldin, A.H. Lashkari, A.A. Ghorbani, et al., Toward generating a new intrusion detection dataset and intrusion traffic characterization, ICISSp 1 (2018) 108–116.
[10] A. Fernández, S. Garcia, F. Herrera, N.V. Chawla, SMOTE for learning from imbalanced data: progress and challenges, marking the 15-year anniversary, J. Artif. Intell. Res. 61 (2018) 863–905.
[11] H. Bandyopadhyay, Autoencoders in deep learning: Tutorial & use cases, 2023, URL https://www.v7labs.com/blog/autoencoders-guide. (Accessed 14 November 2023).
[12] Y. Huang, Y. Wang, P. Wang, Y. Lai, An XGBOOST predictive model of void ratio in sandy soils with shear-wave velocity as major input, Transp. Geotech. 42 (2023) 101100.
[13] Sayantini, Autoencoders tutorial: A beginner's guide to autoencoders, 2023, URL https://www.edureka.co/blog/autoencoders-tutorial/.
[14] X. Kan, Y. Fan, J. Zheng, C.-h. Chi, W. Song, A. Kudreyko, Data adjusting strategy and optimized XGBoost algorithm for novel insider threat detection model, J. Franklin Inst. 360 (16) (2023) 11414–11443.