DOF1.0 Enhancing Data Generation with GenAI

Mariam Salman Mohammad Zbeeb Mohammad Ghorayeb

April 2024

1 Introduction

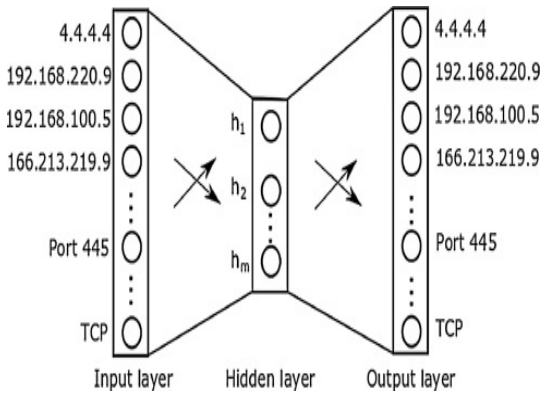
1.1 Problem Statement and Objectives

Generative AI, or GenAI, mimics human creativity by generating text, images, or other data using models trained on vast datasets. It learns patterns from input data to create new content, akin to human creativity. Some datasets are very hard to generate using instrumentation or classical techniques. Our **long-term mission** is to create a rigorous tool capable of generating various complex high-dimensional datasets that are challenging to collect, label, or generate. In the scope of this course contribution, we focus on creating the first module of this tool capable of generating one of the hardest datasets to deal with: malicious networking traffic. For network-based intrusion detection, few publicly available labeled datasets contain realistic user behavior and up-to-date attack scenarios. Available datasets often suffer from being outdated or have other shortcomings. Typically, network traffic is captured in packet-based or flow-based formats. Our objective is to build a generative model, specifically a language model, capable of generating realistic flow-based network traffic. Large training datasets with high variance can enhance the robustness of anomaly-based intrusion detection methods. Thus, our aim is to provide generated data that can be used to improve the training and evaluation of such methods. It's worth mentioning that this is the first language modeling approach applied to this problem. Previous efforts have utilized generative adversarial neural networks, which excel in image generation. However, we believe that this problem is suitable for language modeling.

2 Data

#	Attribute	Type	Example
1	date first seen	timestamp	2018-03-13 12:32:30.383
2	duration	continuous	0.212
3	transport protocol	categorical	TCP
4	source IP address	categorical	192.168.100.5
5	source port	categorical	52128
6	destination IP address	categorical	8.8.8.8
7	destination port	categorical	80
8	bytes	numeric	2391
9	packets	numeric	12
10	TCP flags	binar/categorical	.A..S.

(a) *wireshark*



(b) *GANS*

2.1 Data Acquisition

We embarked on our journey into data-centric engineering by initiating the collection process. We launched into action by setting up machines and orchestrating an attack over IP. This involved one computer attacking another, allowing us to capture the packets by monitoring the incoming traffic using WireShark. As a result of this endeavor, we have acquired 30,000 packets, each adhering to the format depicted in Figure (a).

2.2 Exploratory Data Analysis

Performing basic exploratory data analysis (EDA) such as calculating the variance of each feature or counting the number of unique elements in each column reveals that this data cannot be statistically sampled or simply assembled by hand or through classical techniques. Therefore, we require a methodology to generate more of this data, such as the one I will mention in steps 2 and 3, or one that we will present.

2.3 Generative Adversarial Neural Networks Approach

The only comparable solution to this problem, particularly concerning network data, is through Generative Adversarial Networks (GANs). They typically utilize raw network data without augmenting them into CSVs. However, it's worth noting that GANs are renowned for excelling in image generation, and text generation isn't their primary forte. Language models like GPT, on the other hand, excel in generating coherent and contextually relevant text across various topics. They offer fine-tuned control and interoperability, whereas GANs may face challenges in achieving textual coherence and often require more intricate training for similar results

2.4 Expanding Features

The real data used for training Neural Intrusion Detection Systems (NIDS) typically aren't in the format presented in Figure (a). Instead, they are expanded to contain 80 features each. This expansion process often involves extracting flows from the collected packets using a tool called CICFlowmeter-V4.0 (formerly known as ISCXFlowMeter). CICFlowmeter is an Ethernet traffic bi-flow generator and analyzer designed for anomaly detection purposes. It has been widely utilized in various cybersecurity datasets, including the Android Adware-General Malware dataset (CICAAGM2017), IPS/IDS dataset (CICIDS2017), Android Malware dataset (CICAndMal2017), and Distributed Denial of Service (DDoS) datasets.
With a dataset consisting of 30,000 packets, each containing 80 features extracted using tools like CICFlowmeter-V4.0, our aim is to expand this data to further enhance the training of Neural Intrusion Detection Systems.

2.5 Mapping into Text Domain

We are developing a language model that operates on text rather than numerical data, aiming to predict the next word given a sequence such as "I like food like"; our innovative strategy involves representing our dataset using a finite set of alphabets, dividing the data into 49 symbols, each representing 1 percent of the data's range, and augmenting it by mapping each element to the corresponding domain interval, ultimately resulting in a dataset of 30,000 examples, each akin to a sentence, with an added start symbol denoting the beginning of a packet.

3 Modeling

3.1 Statistics Based Model

The initial strategy entails assessing the correlation between pairs of alphabets by calculating the likelihood of their occurrence in sequence. This enables the construction of sequences by sampling from the relevant symbol pool for each feature. This generative model, rooted in likelihood, assigns probabilities to examples. However, it overlooks data dependencies, prompting a shift towards auto-regressive models. These models disaggregate the probability distribution into a product over individual features, allowing for the modeling of the probability of the next symbol based on preceding ones.

3.2 Deep Learning

3.2.1 Multi Layer Perceptron

In this phase, our objective is to implement a likelihood model capable of predicting the next symbol given a current symbol. We aim to adopt an architecture where, at each layer, numerical data seamlessly flows and weights adjust to predict the alphabet with the highest likelihood to follow the given sequence. However, a significant challenge lies in developing an auto-regressive model rather than a simplistic one, as merely modeling the probability between pairs of characters would yield minimal improvement over the Bi-gram Model.

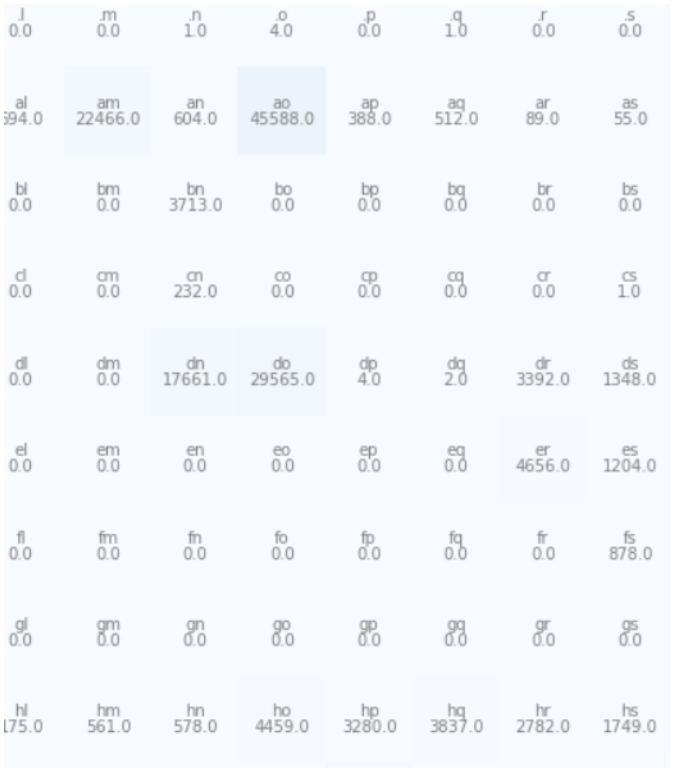


Figure 2: A close look at the Bi-gram model representation (this is not the complete image)

3.2.2 From Characters to Tokens

Ultimately, deep learning boils down to manipulating matrices through operations like w*x, where we squash, add, and introduce biases to create prediction trends. To facilitate feeding data through the neural network, we must once again convert our symbols into numerical representations. Additionally, the model needs to learn dependencies in the data, recognizing that certain characters are interchangeable while others are not, and understanding when it's permissible to replace one symbol with another. So we need to augment our **alphabets** to **vectors** in R^m where m is a hyper parameter.

1. Associate with each word in the vocabulary a distributed word feature vector (a real-valued vector in $\mathbb{R}^m$),
2. Express the joint probability function of word sequences in terms of the feature vectors of these words in the sequence,
3. Learn simultaneously the word feature vectors and the parameters of that probability function.

The feature vector represents different aspects of the word: each word is associated with a point in a vector space. The number of features (e.g. m=5, 10 , 20 in the experiments) is much smaller than the size of the vocabulary (e.g. 50). The probability function is expressed as a product of conditional probabilities of the next word given the previous ones, (e.g. using a multilayer neural network to predict the next word given the previous ones, in the experiments). This function has parameters that can be iteratively tuned in order to maximize the log-likelihood of the training data

3.2.3 Implementation of MLP

To gain insight into the hyperparameters and parameters involved in our computation, we seek motivation from the paper "A Neural Probabilistic Language Model" by Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. The training of this neural network evolves at this stage the tuning of 2 things the weights of the neural nets them selves W and the embedding table C. We desire here two things, the first is to calibrate the weights to output the most likely alphabet to come after and we need to let the embedding C to learn when and what to insert and interchange some of the alphabets at the index (i) of others.

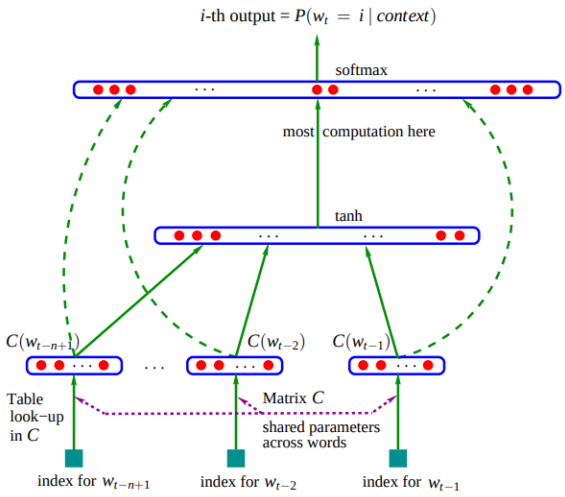


Figure 3: Neural architecture: $f(i, w_{t-1}, \ldots, w_{t-n+1}) = g(i, C(w_{t-1}), \ldots, C(w_{t-n+1}))$ where g is the neural network and $C(i)$ is the i-th word feature vector.

The Forward Pass:

1. Perform forward computation for the word features layer:

$$x^{(k)} \leftarrow C(w_{t-k}),$$
$$x = (x^{(1)}, x^{(2)}, \ldots, x^{(n-1)}).$$

2. Perform forward computation for the hidden layer:

$$o \leftarrow d + Hx,$$
$$a \leftarrow \tanh(o).$$

3. Performing the forward computation for output units in the i-th block involves activating the activations using sigmoid at the output layer. From here, we can sample a character to be the next in the sequence. This sampled character is then fed back into the neural network to predict its next character. The sampling process is driven by training the neural network to increase the probability of the correct label compared to others, encouraging the sampling process at each step.

Backward Gradient Computation

(a) Perform backward gradient computation for output units in the i-th block:

- Clear gradient vectors $\frac{\partial L}{\partial a}$ and $\frac{\partial L}{\partial x}$.
- Loop over j in the i-th block:
  1. Compute $\frac{\partial L}{\partial y_j} \leftarrow \frac{1}{j}\mathbf{1}_{j==w_t - p_j}$.
  2. Update biases: $b_j \leftarrow b_j + \epsilon\frac{\partial L}{\partial y_j}$.
  3. Update $\frac{\partial L}{\partial a} \leftarrow \frac{\partial L}{\partial a} + \frac{\partial L}{\partial y_j}U_j$.

(b) Back-propagate through and update hidden layer weights:

- Loop over k between 1 and h:
  1. Compute $\frac{\partial L}{\partial o_k} \leftarrow (1 - a_k^2)\frac{\partial L}{\partial a_k}$.
  2. Update $\frac{\partial L}{\partial x} \leftarrow \frac{\partial L}{\partial x} + H_0 \frac{\partial L}{\partial o}$.
  3. Update $d \leftarrow d + \epsilon \frac{\partial L}{\partial o}$.
  4. Update $H \leftarrow H + \epsilon \frac{\partial L}{\partial o} x_0$.

(c) Update word feature vectors for the input words:

- Loop over k between 1 and $n - 1$:

$$C(w_{t-k}) \leftarrow C(w_{t-k}) + \epsilon \frac{\partial L}{\partial x^{(k)}}$$

  where $\frac{\partial L}{\partial x^{(k)}}$ is the k-th block (of length m) of the vector $\frac{\partial L}{\partial x}$.

Conclusion: At this stage, the key processes are forwarding, sampling, and updating the weights and embeddings.

4 Advanced Initialization of the Weights

4.1 Eliminating Initial Loss

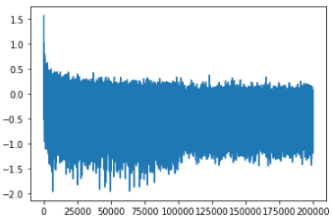
Language models often exhibit high initial loss due to confidently assigning wrong probabilities to incorrect alphabets during the early stages of training. This leads to the neural network spending considerable time reducing this easy part of the loss before converging to an optimal point in the cost function, resulting in resource and time inefficiencies. To address this, we scaled the probability of weights at the output level during initialization by a small value (epsilon). This scaling results in a more uniform or roughly uniform distribution of probabilities, allowing the neural network to assign similar or roughly similar probabilities to each alphabet during the first pass. Let W_{out} be the weight matrix connecting the hidden layer to the output layer, and let U_{out} be the bias vector at the output layer. We initialize these parameters as follows:

$$W_{\text{out}} \sim \mathcal{U}(-\epsilon, \epsilon),$$
$$U_{\text{out}} \sim \mathcal{U}(-\epsilon, \epsilon),$$

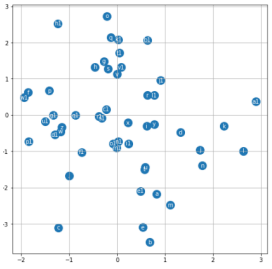
where $\mathcal{U}(-\epsilon, \epsilon)$ denotes a uniform distribution with support in the range $(-\epsilon, \epsilon)$. This initialization strategy ensures that the weights and biases at the output layer are scaled by ϵ, resulting in a roughly uniform distribution of probabilities for the output activations. In mathematical notation:

$$W_{\text{out}}(i, j) \sim \mathcal{U}(-\epsilon, \epsilon), \quad \forall i, j,$$
$$U_{\text{out}}(i) \sim \mathcal{U}(-\epsilon, \epsilon), \quad \forall i.$$

Note that we can take out the bias to be 0 at the initialization level.



(a) Showing that the neural network successfully took out the easy part of the cost function and started fluctuating to land on a minima directly

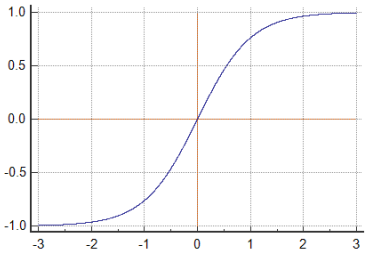


(b) Showing how the neural network learned that some of the symbols can be used interchangeable

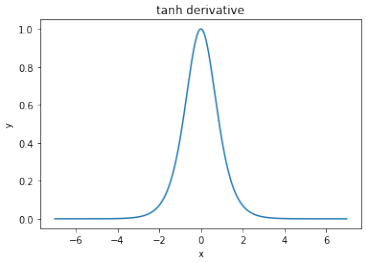
4.2 Fixing the Saturated Tanh

General practice for setting the weights in a neural network is to set them to be close to zero without being too small. Good practice is to start your weights in the range of [-y, y] where y = $\frac{1}{\sqrt{n}}$ (n is the number of inputs to a given neuron which is called the fan in).

Professional practice is to initialize the weights based on the activation function of the next layer.



(a) The Tanh activation function



(b) The gradients of the Tanh

During the forward pass and at the beginning of training, the activations passing through the tanh layer tend to be extreme. According to the mapping of the hyperbolic tangent function, extreme values tend to lie on the tails at either one or negative one.

During the backward pass, when the neurons with tanh activation function come to update their weights, they often encounter a zero gradient, as depicted in Figure (b). Consequently, in the update step $\frac{\partial L}{\partial a} \leftarrow \frac{\partial L}{\partial a} + \frac{\partial L}{\partial y_j} U_j$, the neuron behaves in a shut-off mode, since the gradient is zero and the weights are changed by a magnitude of zero.

To address this issue, managing the standard distribution of the activations entering the tanh activated layer to have a gain of $\frac{5}{3}$ over $\sqrt{\text{fan\_in}}$ ensures that the flow is not extreme and allows the neurons to learn normally.

5 Batch Normalization

The primary motivation for incorporating the batch normalization method in our contribution stems from the recognition that, ultimately, we are dealing with probabilities at the output layer. Therefore, the objective is to ensure that the flow of activations is normalized throughout the network. This normalization not only accelerates the training process but also mitigates overall loss. Most importantly, it alleviates the need for meticulous initialization strategies beyond addressing concerns such as tanh saturation, as discussed in Section 4.1.1. Since the full whitening of each layer's inputs is costly and not everywhere differentiable, we make two necessary simplifications. The first is that instead of whitening the features in layer inputs and outputs jointly, we will normalize each scalar feature independently, by making it have the mean of zero and the variance of 1. For a layer with d-dimensional input $x = (x^{(1)}, \ldots, x^{(d)})$, we will normalize each dimension $x^{(k)}$ as follows:

$$x^{(k)}_{\text{norm}} = \frac{x^{(k)} - \text{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

Algorithm for Batch Normalizing Transform:

Input: Values of x over a mini-batch: $\mathbf{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = BN_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathbf{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{//mini-batch mean}$$

$$\sigma^2_{\mathbf{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathbf{B}})^2 \qquad \text{//mini-batch variance}$$

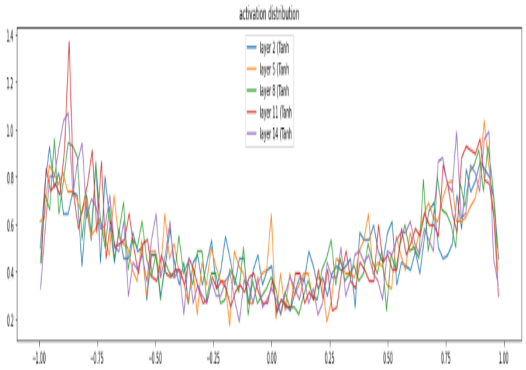
$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathbf{B}}}{\sqrt{\sigma^2_{\mathbf{B}} + \epsilon}} \qquad \text{//normalize}$$

$$y_i \leftarrow \gamma\hat{x}_i + \beta = BN_{\gamma,\beta}(x_i) \qquad \text{//scale and shift}$$

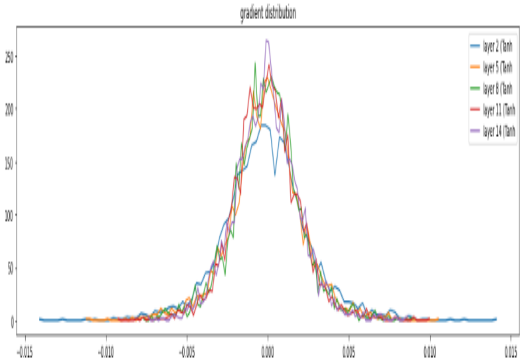
Batch Normalizing Transform, applied to activation x over a mini-batch. The scale γ and shift β parameters endow the neural network with the autonomy to calibrate the normalization process, allowing it to meticulously tailor the distribution of agitations to optimize the flow of gradients during training.

Gains from Batch Normalization:

- **Stabilizes Learning:** Batch normalization helped us stabilize the learning process by reducing internal covariate shift. This leads to faster convergence during training.
- **Regularization Effect:** Batch normalization acts as a form of regularization, reducing the need for other regularization techniques like dropout or even putting the regularization term. This can help prevent overfitting.
- **Improved Gradient Flow:** By normalizing the agitations, batch normalization improves the flow of our gradients through the network, making optimization more efficient.
- **Enables Higher Learning Rates:** With batch normalization, networks can often use higher learning rates without risk of divergence. This can speed up training significantly.
- **Reduced Sensitivity to Initialization:** Batch normalization makes networks less sensitive to initialization choices, which simplifies the process of training instead of worrying more about things like in 4.1.1.



(a) Graph showing how well the activations are distributed after introducing the things we mentioned in 4 and 5. We added layers for visualization effects in plotting.



(b) Graph showing how well the gradients are distributed after introducing the things we mentioned in 4 and 5. We added layers for visualization effects in plotting.

6 The WAVENET Architecture.

The motivation for WaveNet stems from the recognition of sequential dependencies within data. Each symbol's position in a sequence is influenced by preceding symbols. WaveNet, a deep neural network for generating raw audio waveforms, addresses this by employing a fully probabilistic and autoregressive approach. Despite considering all previous samples, WaveNet is efficient, making it suitable for training on datasets with high sample rates. This architecture captures intricate dependencies within the data, particularly valuable for tasks involving sequential information.
In our context, the main ingredient of WaveNet are causal convolutions. By using causal convolutions, we make sure the model cannot violate the ordering in which we model the data: the prediction $p(x_{t+1}|x_1, \ldots, x_t)$ emitted by the model at timestep t cannot depend on any of the future timesteps $x_{t+1}, x_{t+2}, \ldots, x_T$ However the model is very sensetive to make sure that the prediction depends on the previous predictions.
Let $x = (x_1, x_2, \ldots, x_T)$ denote a sequence of data points, where x_t represents the data at time step t. A causal convolutional layer operates on this sequence as follows:

$$y_t = f(x_{t-k}, x_{t-k+1}, \ldots, x_t) = \sum_{i=0}^{k} w_i \cdot x_{t-i}$$

where:

- y_t is the output at time step t,
- f is the convolutional operation,
- k is the kernel size (width of the convolutional filter),
- w_i are the weights of the filter,
- x_{t-i} are the input data points.

In the context of WaveNet or other sequence modeling tasks, the crucial aspect is the causal nature of this operation. This means that y_t only depends on the past or present data $x_{t-k}, x_{t-k+1}, \ldots, x_t$, but not on any future data points $x_{t+1}, x_{t+2}, \ldots, x_T$. Mathematically, we can express this causality constraint as:

$$y_t = f(x_{t-k}, x_{t-k+1}, \ldots, x_t) = \sum_{i=0}^{k} w_i \cdot x_{t-i}, \quad \text{for } t \geq k$$

This characteristic ensures that the prediction yt at time step t solely relies on the data available up to that point, without considering any future data. Such temporal consistency is critical for tasks like time-series forecasting or sequential data modeling, where maintaining the data's temporal order is paramount.

Moving forward, our focus lies in pairing two sets of model inputs simultaneously to enhance dependency modeling within the data. This method allows for modeling pairs of elements from the first layer to the second, gradually progressing to bigrams, fourgrams, and beyond. This incremental approach facilitates the gradual adjustment of model parameters, fostering a more nuanced understanding of data dependencies. Instead of abruptly adjusting weights in a single step, this gradual process aids in learning how dependencies in the data manifest.

Within the literature context, the incorporation of convolutions primarily serves efficiency purposes. Rather than directly forwarding the input into the network, linear filters are applied over the input space, leveraging CUDA-integrated kernels for implementation. However, we won't delve into this aspect further as current computational resources aren't a limiting factor.

While alternative techniques like ResNets and transformers exist for addressing dependency handling, we're currently satisfied with the results and output structure produced by our approach.

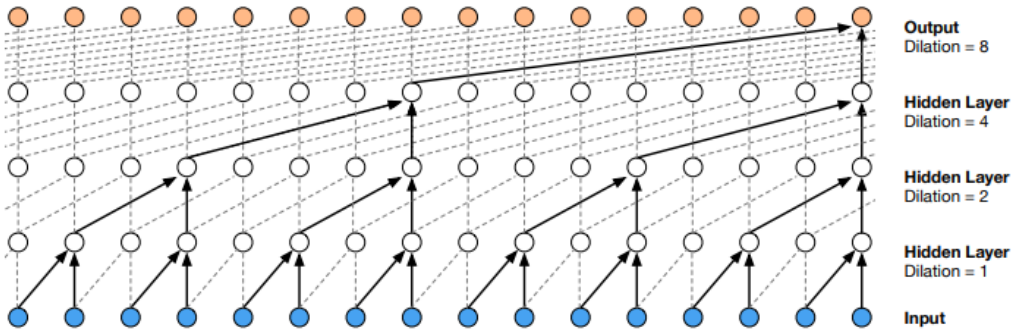


Figure 7: Visualization of a stack of causal convolutional layers.

7 Performance Log

Model	**Test Loss**
Statistical Modeling	1.7
MLP with precise initialization	1.1
Batch Normalized layers	0.9
Wavenet	0.5

Table 1: Test loss on 1000 passes.

8 Testing Generated Data

We are confident that the generated data can significantly improve the performance of the classification models, particularly in the context of classifying networking packets. Before delving into the testing results, it's important to establish some key arguments:

8.1 Mitigating Overfitting

Overfitting occurs when a model(classfier in our case) becomes overly complex for smaller datasets, typically characterized by having more parameters than available data. However, the generated data we employ mitigates this risk in the following ways:

- The generated data substantially increases the size of the dataset, providing a more robust foundation for training the model.
- By ensuring that the generated data conforms to the same distribution as the original dataset, we prevent the model from memorizing specific examples and instead encourage it to learn generalizable patterns.

Therefore, the inclusion of generated data does not lead to overfitting but rather enhances the model's ability to generalize to unseen examples.

8.2 Addressing Underfitting

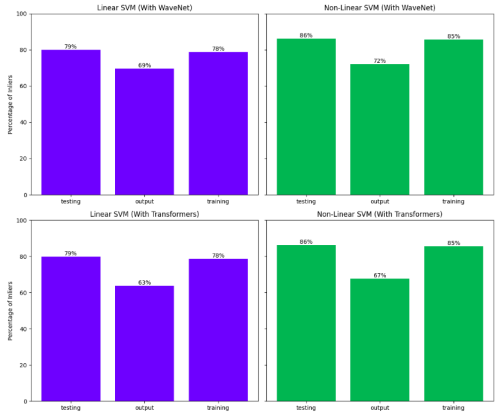
Underfitting occurs when a classifier fails to capture the underlying patterns in the data, resulting in poor performance. The generated data effectively addresses underfitting by:

- Allowing the model to train on a broader range of examples, thereby improving its ability to recognize diverse features.

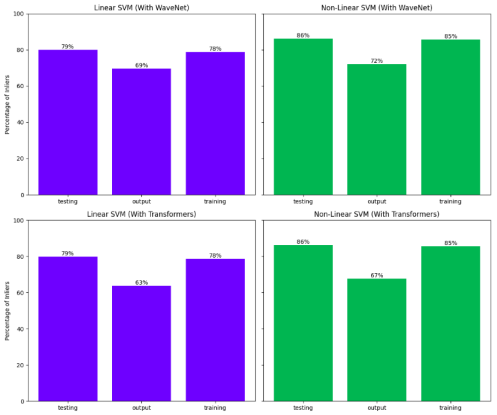
- Providing additional training data that closely resembles the distribution of the original dataset, facilitating a more comprehensive understanding of the underlying data patterns.

8.3 Talking Numbers

Using two testing models, one employing a one-class SVM with a linear kernel and the other with non-linear polynomial kernels, we have obtained the following results: **a ranging percentage of the generated data, from 75 to 80 percent , lies within the distribution of the original data that our model is responsible for generating more of**.



(a) overview1.



(b) overview2

9 Deployment

Listing 1: My Python Code

```
from DOF_API.Network_traffic_generator import Network_traffic_generator
a = Network_traffic_generator()
a.make_more(1000)
```

10 Conclusion

Our team has developed the first domain specific generative model capable of creating structured data files for realistic network traffic. This innovation provides a strategic tool for enhancing intrusion detection systems, offering organizations a significant edge in cybersecurity preparedness. The methodology can be scaled to be adapted to any numerical structured data and we hope to do do in the future.
The model is available as an API and now contain one generator which is network traffic generator that we discussed in this work.
We think its a rigorous tool worth to spend some more time on and adapt some more generators in it.

Acknowledgements

We would like to express our gratitude to everyone who contributed to this project specially to Professor Nassim Daher and Professor Samir Mustapha.

References

[1] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *Journal of Machine Learning Research*, 3:1137–1155, 2003. Submitted 4/02; Published 2/03.

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *arXiv preprint arXiv:1502.01852*, 2015. Submitted on 6 Feb 2015.

[3] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015. Submitted on 11 Feb 2015 (v1), last revised 2 Mar 2015 (this version, v3).

[4] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016.

According to [4], WaveNet is a generative model for raw audio. Additionally, [3] proposed batch normalization as a method to accelerate deep network training. Furthermore, [2] introduced a technique for surpassing human-level performance on ImageNet classification. Finally, [1] presented a neural probabilistic language model.