"ALEXANDRU IOAN CUZA" UNIVERSITY OF IASI

FACULTY OF COMPUTER SCIENCE

BACHELOR THESIS

Improving TrustGAN: Making Deep Neural Networks More Trustworthy

proposed by

Ina Vivdici

Period: july, 2023

Scientific coordinator

Assoc. Professor Liviu Ciortuz

"ALEXANDRU IOAN CUZA" UNIVERSITY OF IASI

FACULTY OF COMPUTER SCIENCE

Improving TrustGAN: Making Deep Neural Networks More Trustworthy

Ina Vivdici

Period: july, 2023

Scientific Coordinator

Assoc. Professor Liviu Ciortuz

Table of Contents

Introduction

It seems like everyone you encounter knows what's AI. ChatGPT changed a lot of the things we do, and that's just the beginning. To me, AI is very inspiring, it gives me hope that we can improve our lives, do the things we love, and be healthier and happier. Machine Learning is a subset of AI, defining how the model learns a task. It is one of the forces driving the AI "movement". Machine Learning is the process of accumulating information about a task and being able to infer using it.

In order to make a difference, and try to help make our lives better I decided to dive into the field of Machine Learning (ML). As it is well known, besides ChatGPT, ML can convert black and white images to colorful ones, generate fake images or videos, and improve the resolution of an image. The second capability is one of the most scandalous ones, there is a lot of debate about its utility. All of these are possible using Generative Adversarial Networks (GANs).

GANs are very used nowadays, thus I decided to see how they can be used. The capabilities I mentioned earlier are the ones GANs are used the most for, however, I wanted to find something even more useful. Upon searching for systems that use GANs, I encountered a system named: TrustGAN. It felt like proof that everything can be used for a greater purpose, not just for fun or harm.

Some people are happy that now you can do less work and let the AI do it for you. On the other hand, there are people that are skeptical and don't rely as much on AI. AI is becoming more present in our lives, and as with any other technology, it poses risks, and TrustGAN can help shrink them. TrustGAN's main purpose is to present the person using an ML model with the model's confidence. This will help to understand how much the model believes the prediction it made. This makes the model more reliable and trustworthy.

As it is with every system and work, you need to understand the fundamentals that they use, such that you will be able to understand how they work and why, this is

the purpose of **Chapter 1**. It presents the 3 main concepts that are needed in order to understand this paper: Convolutional Neural Networks, Generative Adversarial Networks, and Residual Units. They are all presented in a simple manner while ensuring that crucial details were included.

In this paper, I intend to present and improve the TrustGAN pipeline. In order to present the relevance and context of this pipeline I wrote **Chapter 2**. Its purpose is to introduce three articles that target a model's confidence. It presents the main concepts needed to understand the system and how was the model's confidence obtained. One of these three papers represents the basis on which TrustGAN was built, that is (1).

Now, that the context and the fundamental concepts are presented, it is time to get to understand how TrustGAN was created and how it is implemented. In **Chapter 3** there are presented the 2 networks that compose the pipeline, how they interact, and how are they trained. Moreover, it also presents how was the pipeline implemented. The functionalities and project structure is also provided here.

There are no perfect systems, thus I improved the pipeline in the **Chapter 4**. I refactored the code and added new relevant functionalities. I also experimented with the sizes of the networks and the loss the target model uses. On top of this, I integrated TrustGAN with other models too, in order to prove its usability and flexibility.

Now, having the motivation and the paper's table of contents explained, I will continue by explaining the fundamentals. I presented them by providing a clear and concise explanation that captured all the essential aspects without overwhelming them with unnecessary complexity.

Chapter 1

Fundamentals for understanding the paper

Due to the fact that I appreciate works that don't require you to read x other works, I decided to embed in my own work the most important concepts one cannot go further in the paper without them. They will aid the reader in learning a new concept or revising it.

1.1 Convolutional Neural Networks (CNNs)

Convolutional Neural Networks are one of the most used types of networks. They are very useful for image processing and are well adapted to this task. CNNs are used for both of the networks in TrustGAN.

1.1.1 What is Convolution?

In order to illustrate the convolution operation I will use the example referenced in (2) from chapter 9.1. Suppose we have a spaceship and a laser. The laser measures the position of the spaceship at every instant of time. The laser outputs a value $x(t)$ corresponding to the timestamp t. However, our laser is noisy and can't get the exact position of the spaceship every time. Due to this, in order to estimate the position of the spaceship at a certain time, we will average other measurements obtained. Not all measurements have the same importance though, older ones matter less, so we need to have a weighted average. Thus, we will use a function $w(a)$ where a will represent the age of the measurement. Using this function we obtain a smoothed estimate of the

position:

$$s(t) = \int x(a)w(t-a)\,da \tag{1.1}$$

This operation is called **convolution**, but more commonly it is referenced with an asterisk:

$$s(t) = (x * w)(t) \tag{1.2}$$

When using the convolution for the digital representations of numbers we need to discretize it. Thus, in the example with the laser, we will now have regular intervals when it measures the position. We will assume that the laser measures the position once per second, thus t will have integer values. If we now consider that x and w are defined only for t being an integer value we will obtain the 1.1 and 1.2 equations as:

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a) \tag{1.3}$$

Using the convolution terminology, in these equations, x represents the **input** and w represents the **kernel**. The output is also presented as the **feature map**.

1.1.2 Using Convolution in Machine Learning

As anything ever created, until its utility is proven it is rarely used. Thus, I will now introduce the place of the convolution operation in ML and how it is performed.

The Convolution Operation

When it comes to storing data in computers for machine learning applications we often store it in an array. Inputs and kernels are usually multi-dimensional arrays, often referenced as **tensors**, which in ML are usually finite. To illustrate this, let's suppose we have a bidimensional image I as input and that we will use a bidimensional kernel K, thus we will have convolution as being:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{1.4}$$

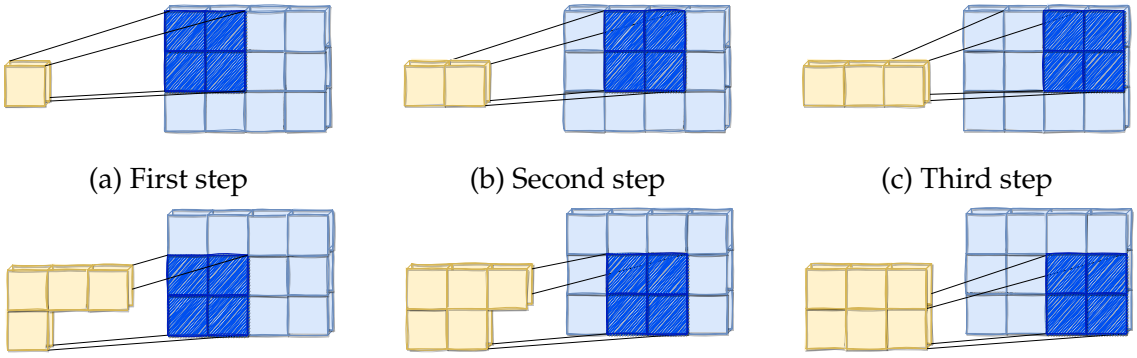
Flipping the kernel relative to the input means that as m increases, the index from the input increases, but the index in the kernel decreases. It is used for obtaining the commutativity property of the convolution. Thus, using this we can rewrite 1.4 equation as:

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n) \tag{1.5}$$

The equation 1.5 is used more often than 1.4 because we have fewer m and n values, thus fewer multiplications. This happens because the kernel is usually smaller than the input. However, commutativity is rarely useful in NNs, thus convolution is often used in another form, called **cross-correlation**:

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i+m, j+n) K(m,n) \tag{1.6}$$

In order to illustrate how this operation is performed in more detail I will use a bidimensional kernel which is 2x2 and a bidimensional input which is 4x3. The kernel "slides" on the input as can be seen in 1.1. The kernel will take every possible position in the input, without going outside the boundaries of the input or itself. As the kernel "shadows" the input, every value of a block of the kernel that shadows another one is multiplied with it, and all these multiplications are summed together, forming the corresponding element of the feature map as can be seen in 1.2.



(a) First step (b) Second step (c) Third step

(d) Fourth step (e) Fifth step (f) Sixth step

Figure 1.1: Applications of convolution operation on the input, using the kernel in chronological order. The pale blue figure with blocks represents the input. The shadow of the white block (the dark blue projection on the input) represents the application of the kernel on the input. The floating yellow blocks represent the resulting feature map. The black lines represent the correspondence between the feature map element calculated and the corresponding kernel position on the input.

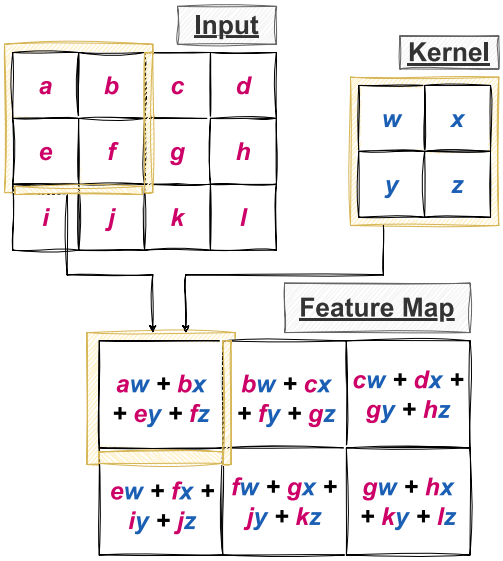


Figure 1.2: How the kernel is applied and the feature map value is calculated in the convolution operation. The input elements bordered with yellow represent which ones are having the kernel applied to. The yellow-bordered feature map element represents the resulting element from applying the kernel to the yellow-bordered elements of the input.

Configuring a CNN

Parsing through a layer from a CNN is usually composed of three main steps:

1. effectuating multiple convolutions in parallel in order to produce a set of linear activations

2. every linear activation is passed through a non-linear activation function (ex. ReLu), also referenced as **detector stage**

3. using the **pooling function** to modify the output for the next layer

The pooling function has the purpose of substituting the result in a section of the network with the summarized statistics of the nearby outputs. One type of this function is the max pooling function, it outputs the maximum value from a rectangular vicinity. Pooling helps with obtaining an approximative invariant in regard to minor changes in the input.

A **stride** represents how many units we move our kernel on the input when applying the convolution operation. In the examples figs. 1.1 and 1.2, the stride was equal to 1. As can be concluded from the calculation of the convolution operation, it shrinks

the size of the next layer. This means that some information from the previous layer will be lost, but this behavior is not always desired. Moreover, not every pixel is used the same number of times. These issues are solved by **padding** the input. Padding means adding zeroes such that they create a border around the input.

What is the Use of Convolution in ML?

In a classical dense neural network a neuron from a layer L influences all the neurons from the layer $L + 1$. However, this is not always the needed behavior. When we use a convolution operation instead of matrix multiplication, that neuron influences only n other neurons, where n is the width of the kernel. Thus, we obtain **sparse interactions**. They have multiple benefits including computational efficiency (fewer calculations for obtaining the values), improved generalization of the task, and feature selection (extracting the most important information).

Moreover, in classical dense NNs, there is one weight per neuron connection, which means that every neuron will learn a different value and representation of the data. However, in CNNs the kernel takes the role of weights in the network. This kernel is applied in the same way in different locations of the input, thus making the network compress the most important features. This process is referred to as **parameter sharing**. Due to this, the network will learn to detect a feature independent of its location in the image. In addition to this, parameter sharing helps with dimensionality reduction by having fewer parameters to learn, which in turn makes the network generalize better and avoids overfitting.

1.2 Generative Adversarial Networks (GANs)

Generative Adversarial Networks are the stars in the ML field when it comes to generating data. They were the obvious choice for the confidence attacker in TrustGAN.

1.2.1 Generative Models vs. Discriminative Models

When it comes to model types in ML there are several ones, but when it comes to estimating probabilities there are two main types: generative and discriminative. Both

of them have the goal of estimating a probability in order to be able to understand how the data is structured.

A discriminative model tries to learn the differences between the types of data, such that it will able to draw the boundaries between them. On the other hand, a generative model tries to understand every little detail and characteristic of the data, having the goal of understanding how the data is embedded into the space. Thus, discriminative models learn only a separator of the dataset classes. On the other hand, the generative models try to model the boundaries of the dataset classes and the distribution of variables in space. Due to this, outliers influence generative models much more than discriminative models.

1.2.2 Why was GAN Created?

As the name implies we need something that will generate data and something that will discriminate, thus distinguish data. The main idea in GANs is that there is a constant game between two adversaries. This idea is based on game theory, more specifically the min-max problem, where each one of the players tries to maximize their gain.

A GAN is a system composed of two neural networks a generator and a discriminator. The goal of the system is that the generator will create realistic data that will be able to fool the discriminator into thinking that it is real, not generated. Thus, the discriminator has the goal of learning to distinguish between fake(generated) data samples and real data samples, while being trained on two datasets: one containing real data samples and the other one containing fake ones (generated by the system).

1.2.3 What is GAN Composed of?

The **discriminator** receives d- dimensional data, with d being the dimension of the dataset, and outputs 0 or 1. Its goal is to classify real data with the label 1 and fake one with 0. The error for this network is used for training the generator. The objective function of the discriminator is:

$$\text{Maximize}_D J_D = \underbrace{\sum_{\bar{X} \in R_m} \log[D(\bar{X})]}_{m \text{ samples of real examples}} + \underbrace{\sum_{\bar{X} \in S_m} \log[1 - D(\bar{X})]}_{m \text{ samples of synthetic examples}} \tag{1.7}$$

, where $D(\bar{X})$ means the value that the discriminator assigns for the data sample $\bar{X}$. As can be observed, the logarithms will both approach 0 when $D(\bar{X})$ approaches the target value: 0 for the fake data and 1 for the real data.

The **generator** has the purpose of generating data such that the discriminator will consider the fake data as being real. It gets as input N_m samples of d-dimensional noise data, and in turn, it outputs synthetic data. Its objective function is directly correlated with the discriminator one:

$$\text{Minimize}_G J_G = \underbrace{\sum_{\bar{X} \in S_m} \log[1 - D(\bar{X})]}_{m \text{ samples of synthetic examples}} = \sum_{\bar{Z} \in N_m} \log[1 - D(G(\bar{Z}))] \tag{1.8}$$

As can be seen here, the right-hand side of the equation expressed the sum for the fake data samples from 1.7. $\bar{Z}$ represents the input sample for the generator and $D(G(\bar{Z}))$ represents the value the discriminator outputs for the generated sample for the noise input $\bar{Z}$ for the generator.

1.2.4 How is Training Performed?

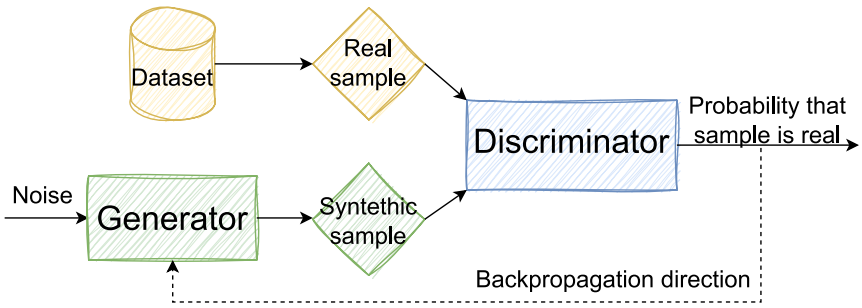


Figure 1.3: Diagram representing how the GAN is trained. The yellow color represents the flow for the real data and the green for the fake data. The dotted line is the back-propagation direction for the fake data after the discriminator outputs a probability associated with it. The rectangles represent neural networks and rhombuses represent data samples. The cylinder represents the training dataset with real data. The lines represent the flow of the data.

As can be seen in 1.3 the discriminator gets two types of data: synthetic(green rhombus) and real (yellow rhombus). Both are used for backpropagating the loss for the discriminator, however, only the synthetic one is used for the generator. The training starts with giving the generator m noise samples and it outputting synthetic sam-

ples. Then, these samples are collected with real m data samples, both of them creating a mini-batch of $2m$ size. Then, stochastic gradient ascent is executed on the discriminator for this mini-batch. This is the first loss computed.

Afterward, the generator is given another m noise samples, and as shown in 1.3 the generated samples are then fed to the discriminator. Then, we use the outputs of the discriminator and execute stochastic gradient descent on the generator. This is done by backpropagating from the output of the discriminator to the generator, but without modifying the discriminator, the direction being shown in 1.3.

The first steps where the discriminator is trained are usually repeated k times, where k is usually a number smaller than 5, and the last steps are repeated only once. If one wants to train a GAN they need to take care of updating the discriminator every time the generator changes, such that the generator will be pushed to create diverse data. For more information (3) can be consulted.

1.3 Residual Unit

Deep neural networks risk the degradation of training performances as they get bigger. In (4) the authors elaborated the concept of deep residual networks which aid this issue. A residual deep neural network has at its basis **residual units**. Residual units can be used with a variety of network architectures, used most often with CNNs because the residual units are especially useful for image recognition. (4)

One of the main ideas is making the network learn the differences between the input and the output. Figure 1.4 represents how the network learns a residual representation. A residual representation here represents the question mark, which means the difference between the output and the input. The input is on the left, represented by the low-quality image, and the output is on the right, represented by the high-quality image. As can be seen in 1.4 the network doesn't need to learn how to obtain the result, but only what to add to the input in order to obtain a result.

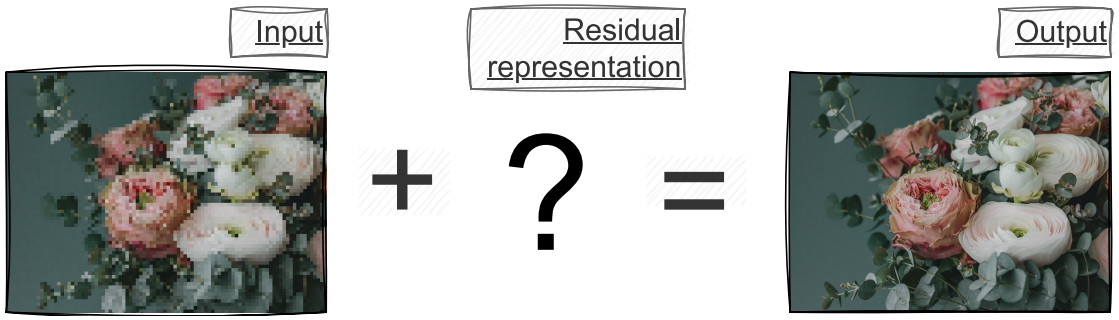


Figure 1.4: Image representing how the network improves the quality of an image using residual units. The low-quality image is the input and the high-quality one is the output. The question mark represents what should the network learn in order to obtain the resulting image.

In order to obtain such a representation we need to structure a residual unit such that it will add the input to the output of a layer. As can be seen in 1.5 there are 2 weight layers in a residual unit. The first one gets the input from the previous layer (here it gets x, the input because it is the first layer). The second layer gets the output of the first one after applying the ReLu activation function. Lastly, the output of the second layer is added to x, and after applying the activation function we get the output of the unit.

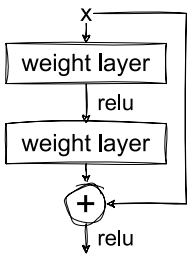


Figure 1.5: Image representing the structure of a residual unit. x represents the input the unit is given. relu is the activation function ReLu. The unit has 2 layers, named weight layers. The arrow from x to the plus means that x is added to the output of the last layer. relu on an arrow means that the activation function is applied on that step. (4)

The arrow connecting x to the last layer of the residual unit represents a **skip connection**. This connection can improve backpropagation. It can make it faster and even help with the vanishing gradient problem, due to the fact that we can compute fewer gradients using the skip connections. The gradient in this case can be computed by skipping a layer and calculating the gradient by using the connection.

Now that we have the fundamentals at our feet, we are equipped to read and understand the next chapter. Thus, let's dive into State of the Art and find out what other systems tried to improve a model's confidence and how did they do it.

Chapter 2

State of the Art

Not all the models need to have reliable confidence in Out-of-Distribution samples, but the ones that do have a strong need for it. Imagine a war machine that needs to identify the opponent's machines, you would most definitely want the model to not make a decision when it sees something similar to the target and tell you that it's uncertain. Considering this crucial use, among the literature, there are various approaches for improving the confidence of an ML model. Thus, I will list here three of them and explain the idea and accomplishments briefly.

2.1 Using Dropout as a Bayesian Approximation

Dropout is a commonly used regularization technique in neural networks. Its main purpose is to prevent overfitting. The main idea is to "drop out" a certain proportion of neurons (excluding the input and output layer), usually done in the training stage of a model. This is accomplished by using the dropout rate, which indicates the probability with which we want to set a neuron's output to zero. This helps the network to develop a broad and diversified interpretation of the data rather than relying on a single neuron or a group of neurons.

Dropout is a close approximation to Bayesian model averaging, which is the act of averaging over several models with varying weights to account for model uncertainty. Dropout adds stochasticity to the model by dropping neurons at each iteration, resulting in distinct sets of weights being utilized for prediction. This is equal to picking a model from a weighted distribution. If more information is needed (5) can be consulted.

The article (6) resides on the fact that a lot of NNs already use dropout, thus in this case, in order to use this strategy, the specialist wouldn't need to change the target model. The authors succeeded to model T forward passes with mathematical formulas, rather than actually doing the passes. This information is further used in order to calculate its mean and variance. The mean represents the output of the network and the variance is the confidence of the network. This is highly intuitive because the mean would represent the average output and the variance would represent how much the output changes. If the output changes a lot, that means the network is unstable and unsure of the predictions it makes.

2.2 Using True Class Probability Instead of Maximum Class Probability

Usually, in Neural Networks (NNs) we consider the output of the NN to be the neuron with the maximum softmax output, namely Maximum Class Probability (MCP). The biggest issue with MCP in the context of estimating the confidence of a model is that the confidence for the correct and erroneous predictions overlap, thus having little to no difference in the confidence (7). The paper (7) introduces True Class Probability (TCP) which has these confidence values clearly separated.

TCP represents the probability with which the model assigns the correct label to the given input. However, this probability can't be directly estimated, thus the paper tries to solve this issue. To the model, we add a ConvNet and a ConfidNet. ConvNet has the purpose of extracting the most important features from the input. These features will then be fed to the classifier.

The classifier is the model we want to improve our confidence for. It learns to predict the conditional probability $P(y|x, w)$, which is the probability of the network to predict the label y given the input x and the weights w. Then, ConfidNet adds an additional output called the "confidence output". The "confidence output" is calculated using a sigmoid calibration function, which maps the raw output of the classifier to a probability that can be interpreted as a confidence rating.

ConfidNet learns to estimate the True Class Probability (TCP) by multiplying the predicted probability of the classifier's output by the estimated confidence score of the model's confidence output. The TCP is then used as an estimate of the probability that

the input belongs to the correct class label. A predicted TCP value is considered correct if it is bigger than 0.5 of the true class probability.

2.3 Using a Generative Adversarial Network

Learning the confidence of a model is also about learning if an input is in- or out-of-distribution. Getting in-distribution (ID) data is obvious and intuitive - get it from the dataset. How about out-of-distribution (OoD)? Generative Adversarial Networks are used a lot nowadays: from converting images from black and white to color to generating artworks. In the paper (1) they were chosen due to their versatility and the ability to generate a lot for various data accustomed to any needs.

A natural question arises though: "Why wouldn't it be efficient to just show the model we want to improve, i.e. the target model, some random noise, in order to make it differentiate between ID and OoD?". The authors conducted multiple experiments. They created a binary classification task, where the class is drawn from a Gaussian distribution, and the data is bounded to the $[-50, 50]$ interval.

They tried 3 different approaches: generating numerous OoD, random noise, and data very close to ID, but still OoD points. The third one improved confidence the most, thus this was the obvious choice. This results comes naturally because it is more challenging to distinguish between OoD and ID samples, when they are similar, which leads to a more accurate estimation of the confidence.

In order for the target model to learn to estimate its confidence correctly it will be attacked by the generator. In this way, the target model will learn the differences in ID and OoD samples and change its confidence considering them. Thus, the target model will get higher confidence for ID samples and a much lower one for OoD samples.

By fine-tuning the losses, layers, and weights the authors succeeded to get better results than most of the other papers did. They forced the Generator (G) to minimize the distance between ID and the examples it generates while generating the samples at the boundary ("in the lowest density") (1) of ID. The Discriminator (D) learned to distinguish between ID and OoD samples, thus helping the generator to create samples that were more similar to the in-distribution samples.

This article is a very interesting application of GANs. It presents a similar use to data augmentation but with the goal of improving trustworthiness. The idea presented in this article seemed very promising to Hélion du Mas des Bourboux, thus he decided

on improving it further and created his own model - TrustGAN (8). So, let's find out how Hélion du Mas des Bourboux created another system for confidence estimation using GANs.

Chapter 3

TrustGAN Presentation

There are several problems related to the state of the art that TrustGAN wants to solve. Firstly, dropout is not always used in a neural network, thus first technique presented in the state of the art wouldn't be a viable one in all cases. Secondly, we would need an additional NN for the second and third techniques when deploying our model, which changes its structure and nature.

3.1 What is TrustGAN?

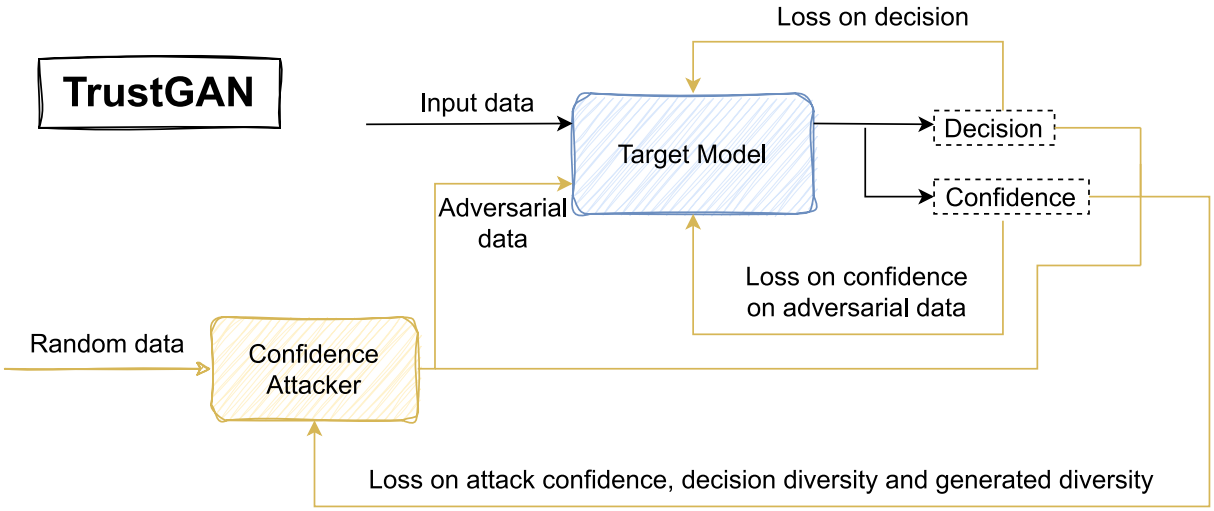


Figure 3.1: TrustGAN training process. The target neural network in blue is improved by the confidence attacker in yellow. Yellow lines represent inputs used only during training, black lines represent inputs used in training and inference. (8)

TrustGAN is a generative adversarial network (8). Its main task is to make the confidence of the target model on OoD and ID data more reliable. As can be seen in

the figure 3.1 the pipeline is composed of a confidence attacker and a target model.

3.1.1 Target Model

The target model represents the model that we want to improve the confidence for. In this current pipeline, it also represents the discriminator. It gets input from two different sources: input data (from the training dataset) and adversarial data (data generated by the confidence attacker). It outputs a decision, which is further used to estimate the confidence. The confidence is estimated using maximum class probability (MCP):

$$\text{MCP} = \max_{0 \leq i < n} \hat{s}_i \tag{3.1}$$

, where n is the number of classes, and $\hat{s}_i$ is the probability (or the score) estimated by the target model corresponding to the class i. Using this definition the probability is in the range $[\frac{1}{n}, 1]$. Starting value means a random prediction and ending value is a 100% confidence. In order to have a confidence score ranging from 0 to 1 $\hat{s}_i$ was re-normalized :

$$\hat{C}_i = \frac{\hat{s}_i - \frac{1}{n}}{1 - \frac{1}{n}} \tag{3.2}$$

The target model has two main tasks that it needs to achieve:

0. produce accurate predictions for the tasks it was designed for, corresponding to input data

1. present high confidence for ID samples and low confidence for OoD samples, corresponding to adversarial data and under-represented samples in the dataset

In order to efficiently train the target model and achieve the above-mentioned goals, that compete with each other, losses are needed. Two of them are used: loss on decision and loss on confidence. In this work the losses are represented as follows: the first index of a loss signifies the network that it targets and the second one - is the index of the loss for that network. Loss on decision is the given loss for task 0: L_{00}. Loss on confidence is the designed loss used by the trustGAN for task 1: L_{01}. L_{01} is given by the soft-cross entropy loss:

$$L_{01} = \frac{1}{\log n} \frac{1}{n} \sum_{i=0}^{n-1} [-y_i \log \hat{s}_i] \tag{3.3}$$

Here, y_i is the target for the i-th sample, i.e. $\frac{1}{n}$. $\hat{s}_i$ is the estimated probability of the target network, i.e. after applying softmax of the output neurons $\hat{l}_i$. $\log n$ serves the purpose of normalizing the loss.

In order to be able to successfully integrate the target model in the TrustGAN pipeline it needs to provide access to how the model is created and trained. This is needed in order to train the network for confidence estimating and/or for the main task it was designed for.

3.1.2 Confidence Attacker

The confidence attacker represents the generator from a GAN. It gets as input random data and outputs adversarial samples to feed the target model. Its goal is represented by three main tasks which don't compete with each other:

0. make the target model confident in whatever decision it takes
1. produce distinct adversarial data samples from distinct random inputs
2. induce distinct decisions of the target model from distinct random inputs

For the first task, the loss is obtained by identifying the class with maximum confidence. It pushes the attacker to generate samples that induce a higher confidence score:

$$\begin{aligned} L_{10} &= -\frac{1}{\log n} \max_{0 \le i < n} \log \hat{s}_i \\ &= -\frac{1}{\log n} (-\max_{0 \le i < n} (\hat{l}_i) + \log \sum_{i=0}^{n-1} \exp(\hat{l}_i)) \end{aligned} \tag{3.4}$$

For the second task, the loss is:

$$\begin{aligned} L_{11} = -\frac{1}{\sum_{j=0}^{N-1} \sum_{k=0}^{j-1} \overline{|r_j - r_k|^m}} \cdot \\ \sum_{j=0}^{N-1} \sum_{k=0}^{j-1} \frac{\overline{|r_j - r_k|^m}}{1 + \overline{|a_j - a_k|^m}} \end{aligned} \tag{3.5}$$

In this equation, j and k are two different samples of a set with N samples. The set can be represented by a batch or sub-batch. a_j represents the adversarial sample produced by the attacker from the random numbers r_j for the sample j. The random

numbers are sampled from a uniform distribution with values from 0 to 1 exclusively. m gives the order of the norm, here $m = 2$ is used. The loss gets closer to 0 as the generated samples get more different.

In a similar manner, the loss for the third task compares the target model decisions:

$$L_{12} = -\frac{1}{\sum_{j=0}^{N-1}\sum_{k=0}^{j-1}\overline{|r_j - r_k|^m}} \cdot \sum_{j=0}^{N-1}\sum_{k=0}^{j-1}\frac{\overline{|r_j - r_k|^m}}{1 + \sum_{o=0}^{n-1} -\hat{s}_{jo}\log\hat{s}_{ko}} \tag{3.6}$$

In this equation, $\hat{s}_{jo}$ represents the score of the target model on the adversarial sample j of data a_j for the class o. This loss gets closer to 0 if the generated scores are very different.

In order to train the network and unify all the tasks into one, these 3 losses are combined into 1:

$$L_{13} = \frac{1}{3}(L_{10} + L_{11} + L_{12}) \tag{3.7}$$

this loss is presented in figure 3.1 as "Loss on attack confidence, decision diversity, and generated diversity".

3.2 Using the Pipeline

As the theoretical fundaments of the system were explained it is now needed to understand how it works when integrated together. Moreover, it is crucial to present how it can be used.

3.2.1 Training pipeline

The training pipeline is composed of three major steps per batch:

1. train the confidence attacker using random numbers as inputs and the loss L_{13}
2. train the target model on the adversarial data generated by the confidence attacker using L_{01}
3. train the target model on the input data using L_{00}

After every epoch the state of the target model and confidence attacker are saved, this facilitates a similar result as experience replay in GAN. The performance metrics are also stored at every epoch, such that it would be easy to visualize how the model is performing. Moreover, at every epoch, two adversarial samples are memorized: a random one and the best one.

There are multiple parameters that can be configured in order to optimize the training pipeline:

- the number of training epochs, default value: 100
- the number of repetitions of the first step, default value: 1
- the number of repetitions of the second step, default value: 1
- the number of repetitions of the third step, default value: 1
- the number of epochs in which the target model will train alone default value: 0
- random proportion of times steps 1 and 2 are skipped, default value: 0
- the validation interval when k-fold validation is performed, default value: 25
- the number of k folds performed, default value: 5
- the batch size, default value: 512
- the target model loss, default value: 'cross-entropy'
- the target model on gan, default value: 'cross-entropy'
- the number of residual units the target model has, default value: 7
- the number of residual units the confidence attacker has, default value: 5

3.2.2 Inference

In order to perform inference, we drop all the yellow lines and the confidence attacker. This current pipeline will be a classical one, containing a model with its input and output. The confidence will be derived from the decision, as stated in the section 3.1.1. Due to the fact that the confidence will have a value ranging from 0 to 1, one can choose a threshold t for any decision taken. The model will thus raise a flag when the confidence is smaller than the threshold, making it more reliable and trustworthy.

3.3 Implementation

The pipeline was used in combination with the MNIST dataset in order to prove its utility and efficiency. This dataset is one of the most used and classical ones for experiments, thus it was chosen to showcase the pipeline.

3.3.1 Neural Networks Structure

The target model is a 2D convolutional neural network with residual connections. This structure is inspired by (9). It is flexible because it works with any image's width and height, it only depends on the channel's number. It was chosen because it makes the model easier to test, making finding OoD images easier, without having restrictions on the size.

The confidence attacker has a similar structure as the target model with some differences. Firstly, it outputs images, rather than numbers corresponding to class classification. Secondly, it has fewer residual units. This was done in order to stop the attacker from making the target model more focused on enduring the attacks rather than learning the task at hand. Thirdly, it uses LeakyReLu, not ReLu, as the target, because this helps the GAN to perform better, for the exact reason batch norm was used instead of weight norm. Lastly, the last layer has hyperbolic tangent as an activation function (tanh), when the target has a linear one.

3.3.2 Project Structure and Functionalities

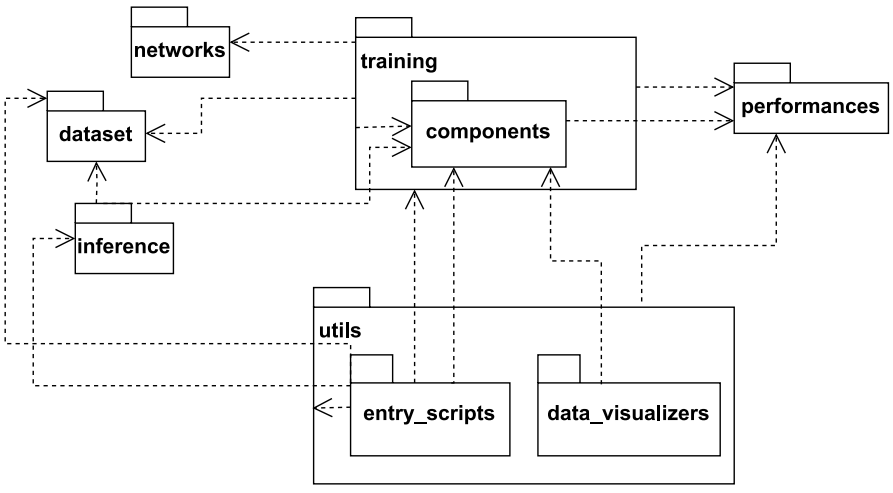


Figure 3.2: UML package diagram for the refactored code containing all the packages and how they interact with each other.

The project has 9 packages represented in the figure 4.1b. The arrows represent which package uses a specific package, with the arrow representing the package that is used. The arrows inside a bigger package going into or outside a smaller package mean that the smaller or bigger package uses the other one.

The package **entry_scripts**, located in the utils package, is concerned with providing entry scripts for using the system's classes for training models, downloading data, and plotting results. It has three scripts dedicated to each task. All of them have different optional and compulsory parameters that indicate how the scripts compile. They help the user interact with the system without diving into the code base. The package **data_visualizers**, located in the utils package, is concerned with displaying the dataset samples and the training results of the network, using Jupyter notebooks.

The **inference** package is used for inferring on a trained model. It returns the true labels, predictions, and confidence scores associated with the predictions. The **networks** package contains all the models created for this system. All of them extend torch.nn.Module, making them have the forward function and a good network structure to be memorized. In my training pipeline, I used Gan and Net classes. Any network that wants to be used as a target model needs to have these parameters for the constructor: nr_classes, device, nr_channels, is_batch_norm, is_weight_norm, dim, residual_units_number.

The **dataset** package handles all the interactions with the dataset. It saves, loads, changes, and extracts the data. It also takes care of saving the data in the needed format and modifying the labels and the dataset as needed by the model. The **utils** package has the purpose of a helper package containing all the additional functionalities other than training, validation, testing, loading, and parsing data. It plots the images that represent good or bad predictions of the network and the confidence attacker generated images. It also plots matplotlib graphs representing the training and validation results. Moreover, it logs and saves data related to the training process like the models and the performances.

The **performance** package takes care of the performance calculations and logging. It has the functions needed for computing the losses. Moreover, it calculates all the losses and accuracies for every model and dataset (training and validation). Then, all this data is saved in a .npy file which can be then plotted using utils. The **training** package contains the logic and functionality of training the models. It helps with recovering from NaN (Not a Number) in the models. Moreover, it trains both of the

models using the components package located in it.

The **components** package, located in the training package, is designated for containing all the elements for the TrainingPipeline class, using the Builder design pattern. As this pattern suggests, the classes in this package need to contain all the data TrainingPipeline needs. Thus, every class contains the data relevant to its name and handles it, as needed. For example, NetworksData creates the networks with the given parameters, configuring their sizes, losses, and types.

Having the system explained, we can now continue with the contributions I have for this pipeline. There are multiple ones, so let's not waste time on a lot of introductory sentences.

Chapter 4

My Contribution

As it is well known, anything ever created can be and should be improved if it is worth using. As I dived in the TrustGAN pipeline there were several improvements I came up with. I describe and list them below, explaining why they were needed and what did I do to achieve better results.

4.1 Code Improvements and New Functionalities

As good as any system can be, it can be made even better. Adding useful functionalities and refactoring the code is the starting point. Thus, this is what I started with.

4.1.1 K-fold Cross Validation

The validation's accuracy in the initial system was calculated by evaluating the performance on the validation set every epoch. This is not among the best quantitative measures of how a model is performing, thus I changed how the performances are estimated. My choice was k-fold cross-validation. Initially, the dataset is split into k parts, with one part of it being dedicated to validation and the rest of them to training. Then, the training and validation process is performed as usual. This process is repeated then k times and the final accuracy for training and validation for the model will be the average of all of the accuracies throughout the k models. Due to the fact that it calculates a mean, thus averaging how a model performs on different chunks of the dataset, its results are more reliable.

I chose to validate at 25 epochs and at the beginning and end of the training be-

cause it would've been very time and resource-consuming to do it frequently. Also, comparing the accuracy on the test set with the validation set time to it is more appropriate for every step in this case. I chose k to be 5, due to similar reasons. 5 is a good measure for averaging the performance considering how time-consuming training is and that my resources are limited.

In order to add this functionality, there were several functionalities that needed to be changed or added. Firstly, I need to change how the data is saved, removing the split of the dataset. Then, I needed to load the data such that it will make 5 different pairs of validation and training sets. These are stored as arrays with loaders for every model respectively. Moreover, I needed to change how the performances are stored, storing the mean only. The last change needed was for plotting, considering that validation and training don't have the same number of calculated accuracies how they are both plotted needed to be customized.

4.1.2 Some Functionalities

Due to the fact that I implemented k-fold cross-validation, I needed to not only save one model but all of them. Also, I needed to save every list of performances for them and a list containing its averages, such that it will be easy to plot and compare them between models. Moreover, I added the functionality of setting k=1 in k-fold, such that one would be able to train only one model.

I also added more parameters to the training script such that it would be easier to customize the behavior of the model like k-fold, validation interval, the path to the performances, number of residual units for the target model, and gan. I implemented measuring the time an epoch executes and getting an average based on the models created during the cross-validation. I also did the same thing for the GPU memory usage, taking the maximum memory usage during the execution. These are saved into a .npy file and the execution time can be plotted.

Moreover, I added Tensorboard plotting functionality. Now, a npy file containing performances can be plotted in Tensorboard. Due to the fact that I could execute only a limited number of epochs at a time and I needed to choose a model from which to continue training. To make this easier, I added plotting the models' performances and average performances together. It is also possible to plot them separately. To make the comparison of different hyperparameters in training I also added plotting of average

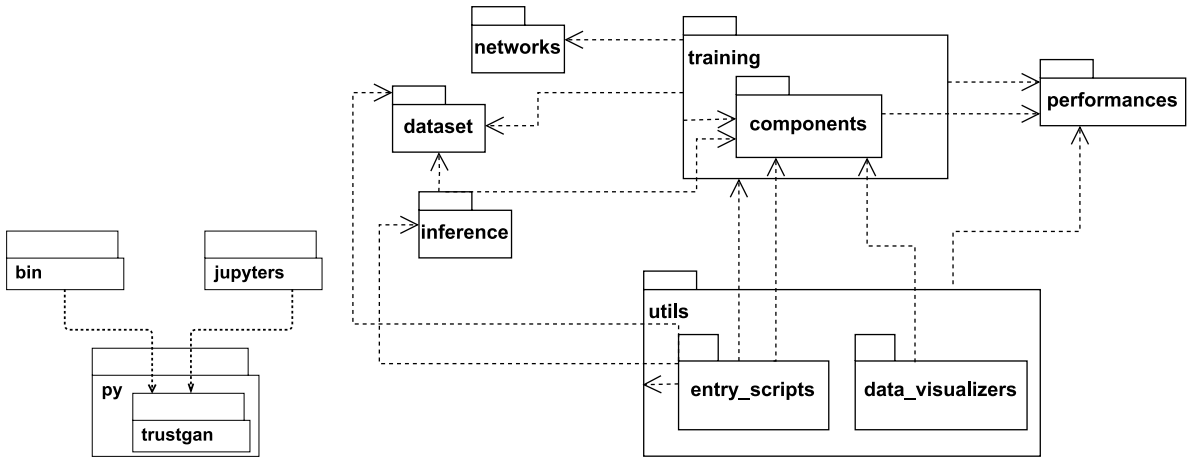
performances together.

On top of this, I added plotting in Tensorboard the execution time for every model, averaging 100 epochs, and displaying the time for every epoch. In addition to this, I also added plotting the maximum GPU memory usage for every model using matplotlib.

Besides this, in order to be able to change the loss for the target model, I also added the functions for hinge loss, squared hinge loss, cubed hinge loss, and Cauchy-Schwartz divergence which can be configured with a parameter when running the training script. In addition to this, there was no script for performing inference after training the model, thus I decided to add it to entry_scripts as well. It has the main purpose of obtaining the confidence score for right and wrong predictions. It can be performed on the train, test, or validation set, which can be configured when executing the script.

One of the functionalities that helps make the system even more flexible is adding the dynamic finding of a constructor for a specific class name and giving its parameters dynamically. This will help to make the parameters a target model receives more flexible. This was done by passing a JSON string, which will be unpacked as a dictionary to the training script and then unpacked again with ** it when calling the class constructor.

4.1.3 Code Refactor



(a) Original code (b) Refactored code

Figure 4.1: UML package diagrams for the original and refactored code, representing the packages and the relations between them.

The initial structuring of the code was very centralized and poorly separated. The training contained the most logic of the system, even the performance processing and predictions. Moreover, classes weren't split into files and there were no packages in which the code logic was divided into. On top of that, the naming of the variables, functions, and classes wasn't self-explanatory and clear.

Due to these facts, when I tried to understand how this system was implemented I decided to dedicate my time to refactoring the code. Clean code is one of the most important assets of a project. Every programmer knows how important it is to understand the code written without reading a lot of documentation or comments. I decided that if someone wanted to improve this system and add new functionalities to it, this would be harder than necessary. Thus, I present my contribution to the code base of the system.

As can be seen in 4.1, originally there were 4 packages, but now there are 9. bin has become entry_scripts, which explains better what it contains. In a similar manner, jupyter is now data_visualizers. The package containing all the functionalities was initially placed in the package py/trustgan. There was no division in packages for the code base. Due to this, I divided the py/trustgan package into 6 main packages: dataset, networks, performances, inference, training, and utils. Every one of them deals exactly only with the thing their names imply, suggesting a good separation of concerns. Moreover, entry_scripts and data_visualizers are placed now in the utils package suggesting their use and functionality in regard to the system.

The figures 4.2 and 4.3 contain all the classes that contain the TrustGAN system for the original and refactored version respectively. In 4.3 the data members and functions aren't represented in order to have better visibility in regard to the interaction between the classes. In the 4.3 entry_scripts and data_visualizers are excluded because they don't have any classes, and don't present the system itself, but how to interact with it.

As can be seen in figure 4.3, the refactored code contains more classes and packages. This is made in order to separate the functionalities better, without having high model coupling. With the refactored code it is easier to find where to make changes or understand what functionality goes where. I reduced model decoupling by making static functions when needed or using just some data members of training, and only when crucially needed to use an instance of the training object.

All the figs. Annex 1 to Annex 6 represent the division of the refactored in pack-

Figure 4.2: UML class diagram for the original code.

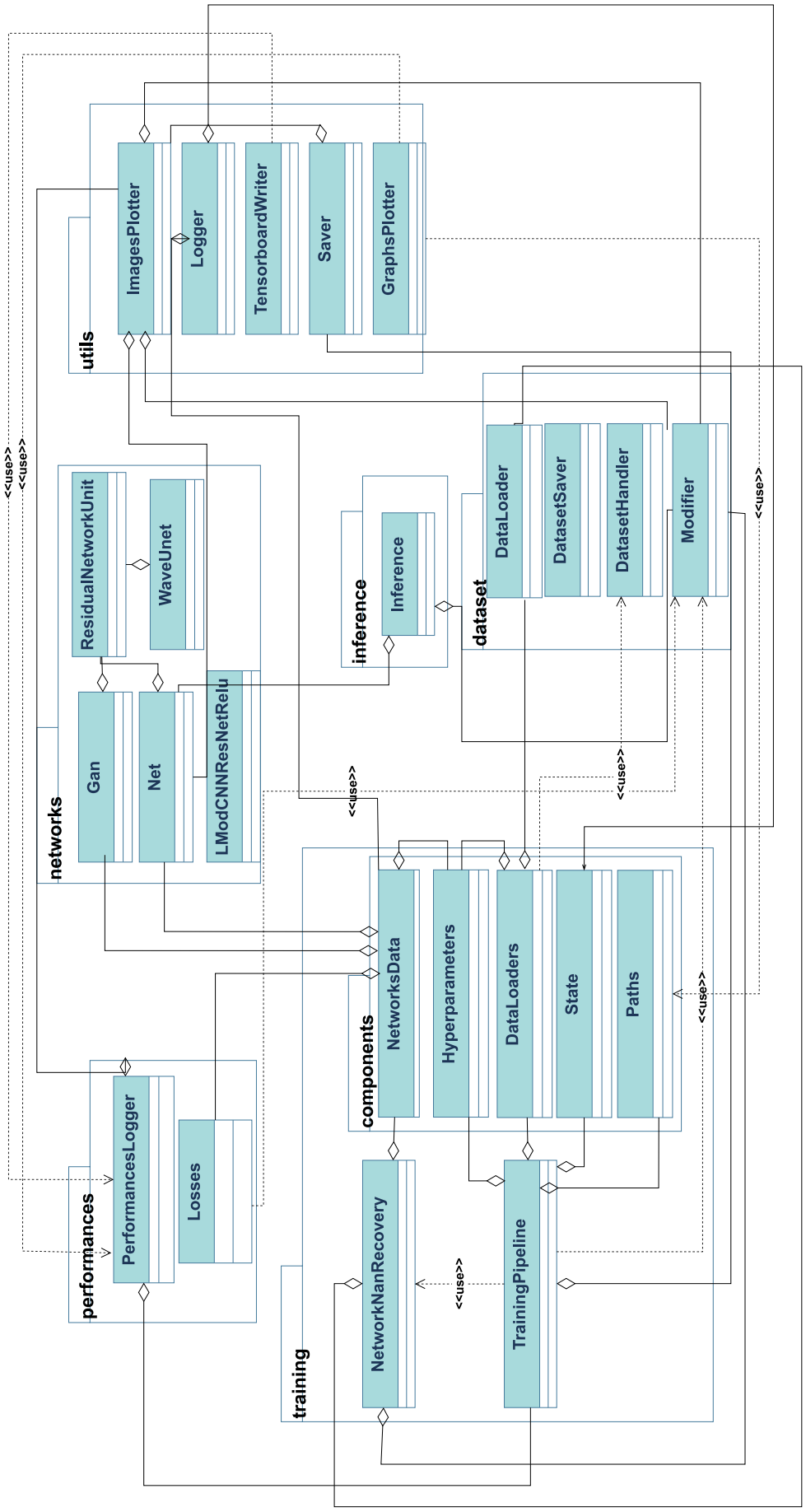


Figure 4.3: UML class diagram for the refactored code.

ages, the classes they contain, and what data members and functions they have. As can be seen in 4.3 the names are more suggestive and clear, and the functions are better separated into classes (this can be better seen at figs. Annex 1 to Annex 6 in the annexes). Moreover, it is clear how many functionalities the code has and how each component interacts with the other, compared to how 4.2 had, where most of the logic was in one class and the functions were complex and did a lot of tasks.

To exemplify, initially, all the code from performances, inference, training, and utils were in the Training class. The Training class, now, named TrainingPipeline, had 16 parameters, and now it has only three. I used the Builder design pattern in order to separate the data members and parameters into separate classes. The package training/data contains some of the data members' training needs. Also, I created new functions, instead of only one big function, for example, the train function in training_pipeline had 80 lines of code, and now it has 8, thus making it simpler to understand. Moreover, I changed some function names, variables, and class names, for example, the net is now named target_model, ResNetUnit is ResidualNetworkUnit.

4.2 Varying the Size of the Networks

As the execution time of this system takes 10 hours for 5 folds, I considered trying to make the networks smaller. I hoped I could get similar performances but much less execution time, and I succeeded.

4.2.1 Explaining the Process

It is worth noting that for achieving the results below and in the section 4.3 I used Kaggle code in order to compile the code. The models were trained on GPU P100, which has 16GB of memory. The CPU provided has 13 GB. All of these mentioned specifications are free to use, with GPU having the limitation of 30 hours of usage per week and the code can execute continuously for no more than 12 hours. Moreover, all the network training was done with the target model that didn't learn anything beforehand.

The size of a network defines how much time it will require to be trained and how fast will it learn. Finding the optimal size is not a trivial task, because just adding more layers than needed may make the results worse, as can be seen in (4). The size

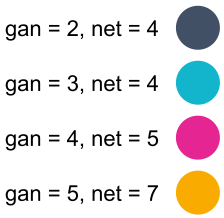
can also induce overfitting or underfitting, if not properly suited for the task. On top of this, it defines how well will it be able to model the task at hand using the losses and the neurons. If the network is too small it won't be able to efficiently model the task at hand or will learn it in an extensive number of epochs. If it is too big, however, it can take a lot of time to train or go too fast into overfitting, not being able to generalize the task properly.

The target model initially had 7 residual units, and with one residual unit containing 2 layers, this means 14 layers. On the other hand, the confidence attacker had 5 residual units, i.e. 10 layers. Due to the fact that the main task for the target model is classifying digits using the MNIST dataset, according to other articles like (10), very good performance can be achieved using smaller networks. Moreover, using Kaggle's free GPU resources, training 100 epochs takes 10 hours approximately. These two factors indicated that experimenting with the size of the networks is a good starting point for improving the training results.

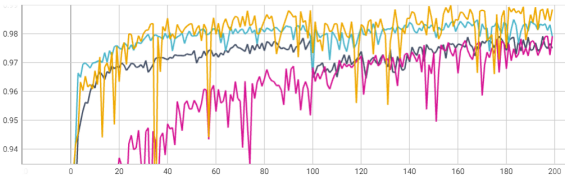
Due to the fact that the task at hand for the confidence attacker is to generate an image meeting certain criteria, I chose for it 2 residual units, considering 1 to not be sufficient enough. Then, due to the fact that the target model must be bigger than the target model, such that the second one won't get to be too good at generating images and attacking, I chose to have 4 residual units for the target model. Then, I chose to experiment and add one more residual unit to verify the aforementioned theory. Lastly, I chose to have 3 units for the confidence attacker and 5 for the target model, due to similar reasons as with 2 and 4 residual units.

4.2.2 Performance Comparison

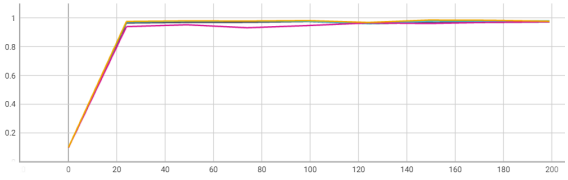
Figure 4.4 represents the performances for every configuration of models executed: a confidence network (gan) with 2 residual units and a target model (net) with 4 residual units and so on. All of them are color-coded as presented in the legend. As can be seen in 4.4b, the **best accuracy at training for the target model** is reached for the yellow model in training. Nonetheless, there is still not a crucial difference between the biggest and smallest network. At the last epoch, the smallest network configuration (gan=2, net=4) has 0.975 accuracy and the biggest one (gan=5, net=7) has 9.884. Moreover, the smallest configuration is more stable, having fewer variations. However, at validation, there is no significant difference between a bigger and a smaller pipeline.



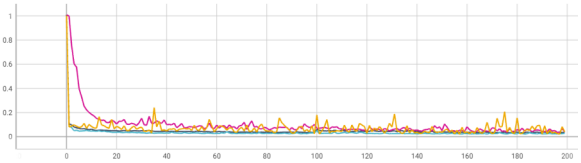
(a) Legend representing the models each color represents



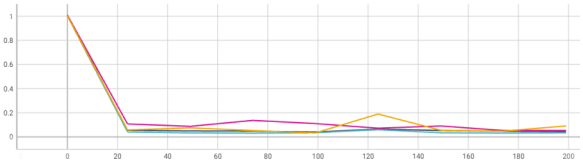
(b) Training target model accuracy



(c) Validation target model accuracy

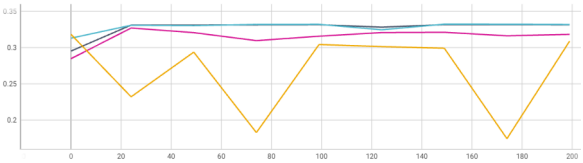


(d) Training target model loss

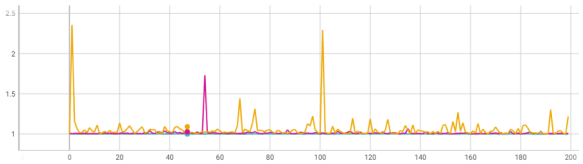


(e) Validation target model loss

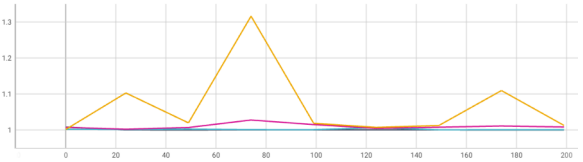
(f) Training confidence attacker loss



(g) Validation confidence attacker loss



(h) Training target model on GAN loss



(i) Validation target model on GAN loss

Figure 4.4: Performance metrics for 4 models with different residual units number for the training and validation set. GAN is the confidence attacker and net is the target model. The numbers in the legend represent the number of residual units for every network.

The **target model loss on training and validation** doesn't differ very much from one configuration to another, thus motivating us to try a smaller network further. As for the **confidence attacker loss on training**, the bigger network succeeds to achieve a smaller loss, at least periodically. The **confidence attacker loss on validation set** has the biggest network as also the one with the most variations and also the smallest loss achieved. However, the target model on GAN loss is more important than the GAN, because making a GAN learn the task well is not our ultimate goal, but making the target model distinguish between OoD and ID.

Interestingly, the **loss for the target model on GAN** is approximately the mirrored version of losses for the GAN. This could indicate that the target model is performing much better at the task than the confidence attacker. Considering that the biggest network here doesn't have the smallest loss, the idea that the networks should be smaller is still present.

Taking everything into consideration, we could consider that it is worth it to try different losses for the smallest model in order to try to make its performances closer to the bigger ones. However, it is important to also note if the execution time and GPU usage get smaller and by how much, considering that this is also a goal of ours - making them smaller. If we could achieve this, the system would need fewer resources for training.

4.2.3 Resource Usage Comparison

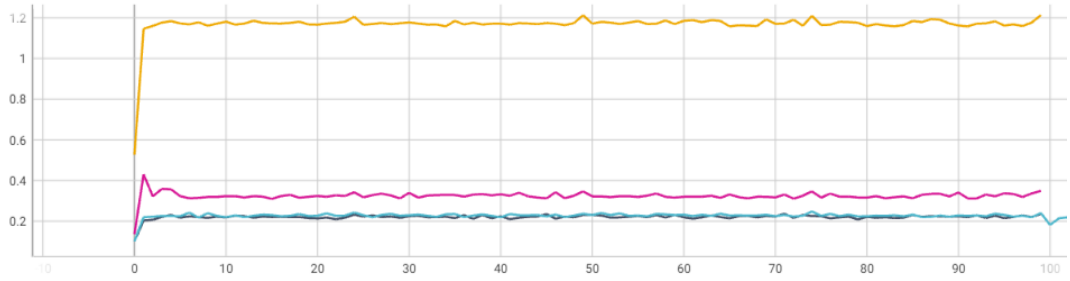
The resources used by the models that were tracked were the maximum GPU usage and the time in minutes an epoch executes. The first graph 4.5a was created by taking the epoch modulo 100 and averaging the existing execution time. The second graph 4.5b was created using the torch.cuda utility, *max_memory_allocated(0)*. The coloring of the graph is the same as for the 4.4.

As can be seen in 4.5a the **execution time** for an epoch for the biggest model is approximately 5 times more than for the smallest ones. This would signify faster training, we could perform more epochs in less time and get even better results as seen in the previous section.

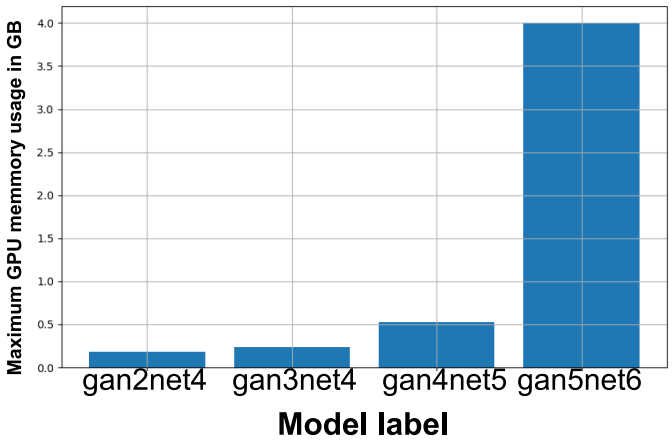
Moreover, the **GPU memory usage** in GB presented in 4.5b is more than 4 times bigger for the biggest model. This means that not only the smallest model can achieve better results faster, but also with less memory. Considering everything said, it is worth

trying to improve the smallest model.

Moreover, this is the reason I did not try the configuration gan=3 and net=4, because it is clear that the smaller configurations behave better than it. It is not crucially different when it comes to performance for the bigger models, but it is faster and requires less memory. Thus, making it desirable. So, the next step is trying to change the loss for the target model with the smallest network configuration.



(a) The time in minutes the epoch executes



(b) The maximum allocated GPU memory in GB

Figure 4.5: Resource usage for 4 models with different residual units number. The first graph represents the mean from the 200 epochs in 100 chunks of the time it takes to execute the epoch. The GAN is the confidence attacker and net is the target model. The numbers in the legend represent the number of residual units for every network.

4.3 Experimenting with the Losses for the Target Model

As varying the sizes of the networks proved to be successful, I wanted to see how the losses impact the small networks. As the paper (11) suggested, cross-entropy shouldn't be the only loss tried on a network, thus I experimented with 4 other losses.

4.3.1 Explaining the Process

The loss defines how well a model learns, it defines how it distinguishes between a well-classified item and a bad one. It also defines how should the model perform backpropagation, i.e. how should the neurons' values change. Choosing a loss is a very important step in creating a model. It defines how fast will a model train, how will it overcome plateaus, and how far will the accuracy get (11).

There are numerous loss functions that can be chosen for training a model. However, trying a lot of different loss functions in order to find the best one for the model is not a common practice. One of the most popular losses is cross-entropy loss, however, it is not universal, nor the best option for every model (11). However, this is the loss the author chose for the target model. Thus, I chose to try various losses, especially the ones that performed well for a deep neural network for the MNIST dataset.

The first loss I decided to implement is the hinge loss. Its popularity is increasing lately and it shows very good results on various datasets. In (11) hinge loss obtained a good accuracy, thus I chose it. Moreover, I chose to experiment with squared hinge loss and cubed hinge loss, because they proved even better than classical hinge loss. On top of these, Cauchy-Schwarz Divergence also proved to be successful on the MNIST dataset, so I chose to try it too.

4.3.2 Describing the Chosen Loss Functions

Cross Entropy Loss quantifies the discrepancy between the target variable's actual probability distribution and the projected probability distribution. Large discrepancies between projected and actual probability are penalized. Additionally, it forces the model to give the true class high probabilities. In probabilistic models, the **Cauchy-Schwarz Divergence** is used to assess how differently two probability distributions behave. The Cauchy-Schwarz inequality, which asserts that the inner product of two vectors is constrained by the product of their norms, serves as the foundation for this statement. This loss motivates the model to reduce the difference in probability distributions between the two.

Hinge Loss was primarily developed for binary classification using Support Vector Machines (SVMs). It determines the difference between the genuine class's expected score and the other classes' scores. The loss is greater than zero if this margin falls below a predetermined level. Instances having a margin greater than the criterion

are more likely to be appropriately classified as a result. An alternative to hinge loss is **Squared Hinge Loss**. Because it squares categorization errors, it penalizes them significantly more. This loss motivates the model to have a wider class division. Hinge loss can also be used in the form of the **Cubed hinge loss**. It encourages an even larger margin by penalizing misclassifications more heavily than the squared one.

Symbol	Loss name	Equation
log	log (cross-entropy) loss	$-\sum_j \mathbf{y}^{(j)} \log \sigma(\mathbf{o})^{(j)}$
D_{CS}	Cauchy-Schwarz Divergence	$-\log \frac{\sum_j \sigma(\mathbf{o})^{(j)} \mathbf{y}^{(j)}}{\|\|\sigma(\mathbf{o})\|\|_2 \|\|\mathbf{y}\|\|_2}$
hinge	hinge (margin) loss	$\sum_j \max(0, \frac{1}{2} - \hat{\mathbf{y}}^{(j)} \mathbf{o}^{(j)})$
hinge^2	squared hinge (margin) loss	$\sum_j \max(0, \frac{1}{2} - \hat{\mathbf{y}}^{(j)} \mathbf{o}^{(j)})^2$
hinge^3	cubed hinge (margin) loss	$\sum_j \max(0, \frac{1}{2} - \hat{\mathbf{y}}^{(j)} \mathbf{o}^{(j)})^3$

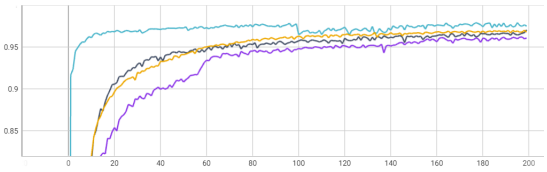
Table 4.1: The equations for the used losses for the target model. $\mathbf{y}$ is the true label as one-hot encoding, $\hat{\mathbf{y}}$ is the true label as +1/-1 encoding, $\mathbf{o}$ is the output of the last layer of the network, $\cdot^{(j)}$ denotes the jth dimension of a given vector, and $\sigma(\cdot)$ denotes probability estimate.

4.3.3 Performance Comparison

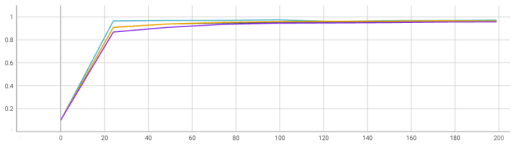
As was said in the last section, the smallest network configuration seemed very promising due to the good performances and fewer resources requirement. Thus, I tried the 3 variants of Hinge loss and the Cauchy-Schwartz divergence as they obtained good results in (11). I executed only 100 epochs with the hinge loss because it proved to have terrible accuracy compared to the other ones.

In 4.7 can be seen that hinge loss has an accuracy twice as less as the other losses. This means that it converges much slower, thus I decided to not execute it 100 epochs. Due to this in 4.6 there are only 4 losses, excluding the hinge one.

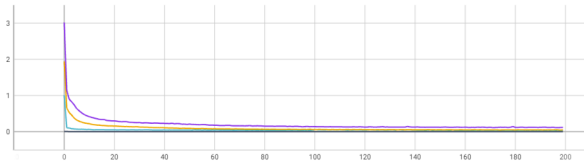
(a) Legend representing the loss each color represents



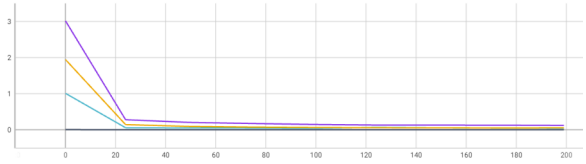
(b) Training target model accuracy



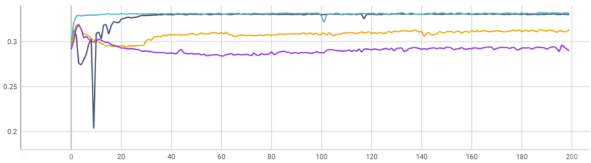
(c) Validation target model accuracy



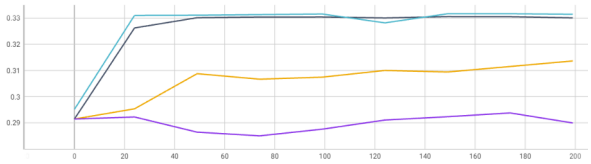
(d) Training target model loss



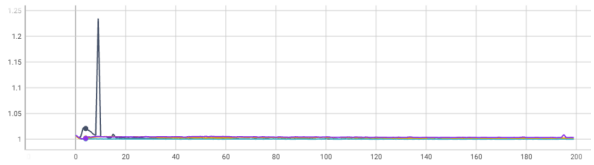
(e) Validation target model loss



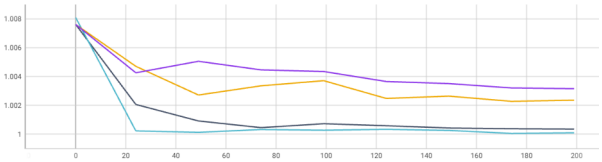
(f) Training confidence attacker loss



(g) Validation confidence attacker loss



(h) Training target model on GAN loss



(i) Validation target model on GAN loss

Figure 4.6: Performance metrics for 4 different losses number. Hinge loss is not represented due to the dramatic difference in accuracy. All of the losses are applied on a confidence attacker with 2 residual units and a target model with 4 residual units.

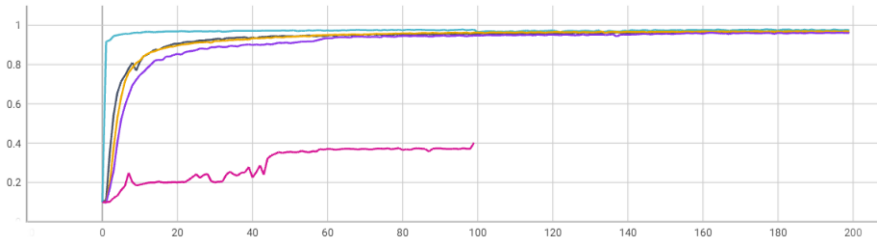


Figure 4.7: The training accuracy for the target model graph including the hinge loss, that is executed for 100 epochs.

Due to the fact that the hinge loss had such a terrible accuracy, we will further analyze the losses by omitting the hinge loss. As can be seen in 4.6b representing the **training accuracy for the target model**, cross-entropy obtains the highest accuracy, with hinge cubed being the second. The Cauchy-Schwartz being the third, competing and sometimes getting higher accuracy than the second place.

However, at the last epoch, D_{CS} gets the second-best accuracy, with log having 0.975 and D_{CS} having 0.9705. $hinge^3$ has 0.9679 and $hinge^2$ has 0.9609. It is clear that log converges faster at the beginning. However, more epochs are needed in order to make a final conclusion about the loss, because in the last epochs D_{CS} presents growing tendencies, while other losses remain stable.

When it comes to the **validation accuracy for the target model** D_{CS} and $hinge^3$ overlap, being the second best after log, with $hinge^2$ being the least successful. For the validation dataset, the convergence speed rating is the same as for the training, with log being the leader.

The **training loss for the target model** has D_{CS} with the smallest loss and log with a slightly bigger one. Considering that D_{CS} is very close to log at other performance metrics, if it had a bigger loss, at least at the beginning it could've gotten a better accuracy, better signifying what should be improved. At the last epochs, the rating is still the same as in the beginning, but with a much smaller difference between the losses. The **validation loss for the target model** has a similar behavior as for the training set in regard to the rating and behavior of the losses.

The **training confidence attacker loss** has the losses rating in reverse compared to the target model. This makes sense, if the target model understands its task worse, it is easier to attack. Here, the losses keep a pretty good distance, every one of them being far from each other. In the **training confidence attacker loss** the losses are much more differentiated and spaced than for the training. Despite this, their rating is still

kept.

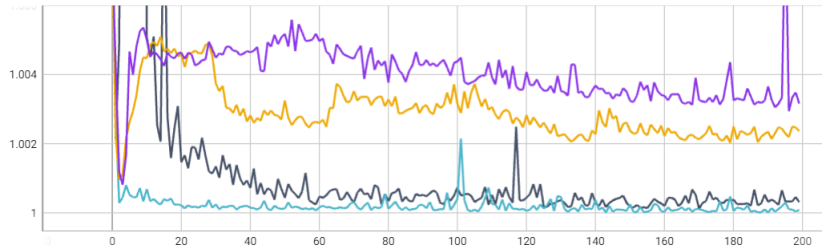


Figure 4.8: The training target model on GAN loss graph, close-up view of the results.

The **training target model on GAN loss** has the D_{CS} with a big spike at the beginning, thus marking that it fluctuated a lot then. In 4.8 we can see that in places other than the spike the log loss has the smallest loss. However, despite the spike the D_{CS} had, it is very close to the best one - the log. We can also observe that $hinge^3$ is the third best and $hinge^2$, approximately the same as in other performance metrics. The **validation target model on GAN loss** preserves the same behavior as it did on the training set, the differences being easier to observe.

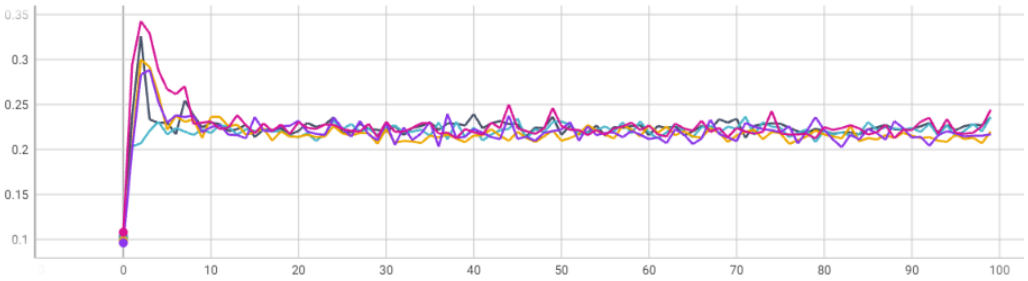
Considering everything said, D_{CS} seems worth trying to execute more epochs, with log definitely being a good option, maybe even the best. Thus, even if log is the most used one loss, and maybe the only loss tried with a model, it doesn't mean it is not the best as it was in (11). Thus, it is relevant, but it is definitely not the only loss available and others should be tried too.

Considering that the losses don't change the network's structure, they shouldn't make a big difference in resource consumption, but every theory needs practical proof. Thus, let's analyze the maximum GPU usage every one of them had and the time they executed.

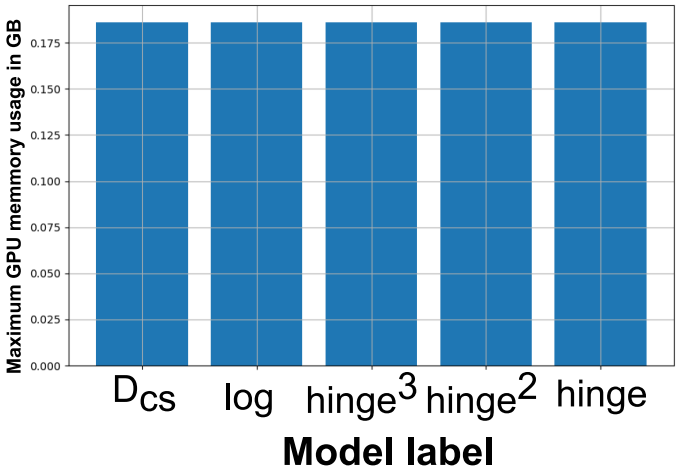
4.3.4 Resource Usage Comparison

The first graph 4.9a represents the average execution per epoch, like the size experiments. The legend of the graph is the same as for the 4.6. The second graph 4.9b represents the maximum GPU usage in the whole training sequence, like the size experiments.

As can be seen in 4.9a the **execution time** for each epoch executes for the models variates, however, not drastically. The only substantial difference is at the start, but when training multiple epochs, that doesn't make a tremendous difference. Thus, we



(a) The time in minutes the epoch executes



(b) The maximum allocated GPU memory, in GB

Figure 4.9: Resource usage for the 5 models with the associated losses. The first graph represents the mean from the 200 epochs in 100 chunks of the time it takes to execute the epoch. All the losses were applied on a confidence attacker with 2 residual units and a target model with 4 residual units.

can confidently say that in regard to execution time, these losses are similar.

In 4.9b we can notice that he **GPU memory usage** has practically the same values for each loss. These make sense because they don't load the GPU enough to be noticeable, even if some of them need more computations. Moreover, they don't change how big are the models, which influences the resources needed the most.

4.4 Using TrustGAN with Other Systems

As the main TrustGAN idea is to improve the model's confidence it is crucial that it works when integrating it with other models too. In (8) there are two datasets tried: MNIST and CIFAR-10. Thus, I decided to try to incorporate the system with other network systems too. This requires making the system even more flexible when

it comes to data loading, saving, and even training.

One of the first natural ideas I came up with is to try and integrate TrustGAN with a system one of my colleagues, Ioana Petrariu, has. She also used PyTorch, which makes our integration easier. We both have the same coordinator, thus we meet frequently. We decided to try to integrate TrustGAN with the GCN model of hers. Her model is based on (12). For writing the code she was inspired by the official documentation of PyTorch (13). Our decision resides in the fact that it is one of the easiest to understand and that it gets good performances easily. Thus, we wanted to see how it will behave under a confidence attacker.

Obviously, in order to integrate the new models we needed to look into Ioana's implementation and understand it. We analyzed what is the dataset composed of and how it is loaded. We needed to implement saving and loading because it wasn't present. Moreover, we analyzed the training process and what data the model needs for the forward pass and its construction. After we got a general view of the system, we started thinking of a template that can be used by others to integrate their system.

Template creation of the system is an intricate task because the system's code base wasn't designed with as much flexibility as a template needs. For example, a person that wants to integrate TrustGAN might not want to save the data in separate files and process it. There are a lot of things that someone might want to do differently. When I thought of the forward pass of the network I didn't even think that someone wouldn't want to just pass an input, but also other data, like in Ioana's case: an edge index, which meant which nodes does an edge connect (13).

Due to the fact that my intention is to create this system abiding by the SOLID principles, i.e. Single-Responsibility Principle, Open-Closed Principle, Liskov Substitution Principle, Interface Segregation Principle, and Dependency Inversion Principle, the system's structure needs extensive planning. The fact that when creating the template it can't be tested by itself makes the task even more complex. The whole idea is that it requires some implementations that will make it work.

On top of this, Python doesn't have a very solid OOP base. Thus it was hard to try to model some requirements for the system to work. Someone that isn't careful enough might miss an additional parameter that needs to be returned and debug hours on end. An alternative idea would be to let most of the functionalities and elaborate a guide on how to change functions in order to achieve a specific goal. However, this violates the Open-Closed principle and the Interface Segregation Principle. I didn't want that

in order to use the system one needs to understand it in depth.

Considering everything, I tried making abstract functions that need to be implemented when inheriting a class. The parent has most of the functionality needed and the child just needs to configure some things and change the behavior as desired. I did this for the data loading, saving, target model training, and obtaining the target model output. I succeeded to make GCN work with the confidence attacker and all the functionalities I mentioned in the paper. I even let the person that wants to use the system be able to implement a class named Modifier if they want to change the data in any way. It is also possible to change the DataLoader type, by just changing the name of an import.

Thus, I proved that TrustGAN can and should be integrated with other systems. Moreover, creating a template is an optimum solution, because it won't be a burden to adapt it to a custom task, other than the one it was tested on. Considering everything I contributed, I consider that TrustGAN is one step closer to being integrated with a plethora of other systems in order to make them more trustworthy.

Conclusions and Future Work

This paper presented the reader with a system named TrustGAN and the improvements I succeeded in applying. In **Chapter 1** we expanded our understanding or just reviewed it on 3 topics: CNNs, GANs, and Residual Units. In **Chapter 2** we found out who are the other people that tried to improve a model's confidence and what did they create for this. Moreover, we understood that improving a model's confidence is very relevant and can have a plethora of solutions.

As we got to understand the context in which the system was elaborated we found out how it was implemented and planned in **Chapter 3**. This chapter provided us with the main features of the theoretical foundation of the system and the practical ones. Having all the needed information explained, in **Chapter 4** we found out how can the pipeline be improved, from the code base to the networks. We even found out that TrustGAN isn't a system that only work for the initial task it was integrated with, it is not fragile.

In addition to this, I proved that TrustGAN can be integrated with other systems and make them more trustworthy. I also helped them understand the system better with UML diagrams and short explanations. I consider the new functionalities I brought in useful, they make the system easier to use and more versatile.

Considering my contribution, there is a place for smaller networks in this system. Even if the author considered the bigger networks to be suitable, I proved that smaller ones aren't much worse. Moreover, the code can always be improved, because nothing is perfect. However, I made the work for someone that will use TrustGAN in the future easier.

Sadly, as it is with everything in this world, my time to create this work was also time-limited. Thus, I will present the things I wish I had time to implement and improve, in the hope that this paper rose interest in someone and that they want to add something good to this useful system.

When it comes to training, I think even more experiments are needed with the system. All of the hyperparameters this pipeline has should be tuned and experimented with. Now that the system is structured better this is easier. I am sure that at least the batch size can be made smaller as many papers state that it can improve training (14). In addition to this, I think it is worth trying to change the losses for the target model on GAN too, in order to find the optimal one, due to the reasons stated in (11).

Another useful contribution would be comparing how the confidence is estimated for TrustGAN and for the state of art systems for model confidence estimation. This could include the same exact graphs that were seen for performances and resource usage. This could indicate the need for optimization.

Considering the results in the losses experiments section, the Cauchy-Schwarz loss should be executed for 200 additional epochs and compared to cross-entropy. This should be done for a confidence attacker with 2 residual units and a target model with 4 residual units. It can bring significant improvements to the network. Thus, it is worth exploring. Moreover, some other experiments with a pre-trained target model would be useful. They would help to see how the behavior changes and thus see what other changes in hyperparameters are needed.

As was mentioned in 4.4 creating a template that will embed TrustGAN and will be flexible for adding new functionalities or eliminating them isn't a trivial task. It is completely different when starting from 0 with this task in mind and when creating a system and refactoring it in order to fit a different task. Thus, the template I created still needs work. It needs to be even more flexible and configurable. Obviously, another future contribution is integrating TrustGAN with other systems. This represents the "ultimate test" exposing the system's flexibility in other implementations.

Bibliography

[1] K. Lee, H. Lee, K. Lee, and J. Shin, "Training confidence-calibrated classifiers for detecting out-of-distribution samples," 2018, accessed on: 20.06.2023. [Online]. Available: https://arxiv.org/pdf/1711.09325.pdf

[2] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org Accessed on: 20.06.2023.

[3] C. C. Aggarwal, *Neural Networks and Deep Learning: A Textbook*, 1st ed. Springer Publishing Company, Incorporated, 2018, accessed on: 20.06.2023.

[4] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015, accessed on: 21.06.2023. [Online]. Available: http://arxiv.org/abs/1512.03385

[5] P. Baldi and P. J. Sadowski, "Understanding dropout," in *Advances in Neural Information Processing Systems*, C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Weinberger, Eds., vol. 26. Curran Associates, Inc., 2013, accessed on: 20.06.2023. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2013/file/71f6278d140af599e06ad9bf1ba03cb0-Paper.pdf

[6] Y. Gal and Z. Ghahramani, "Dropout as a bayesian approximation: Representing model uncertainty in deep learning," 2016, accessed on: 21.06.2023. [Online]. Available: https://arxiv.org/pdf/1506.02142.pdf

[7] C. Corbière, N. THOME, A. Bar-Hen, M. Cord, and P. Pérez, "Addressing failure prediction by learning model confidence," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019, accessed on: 20.06.2023. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2019/file/757f843a169cc678064d9530d12a1881-Paper.pdf

[8] H. du Mas des Bourboux, "Trustgan: Training safe and trustworthy deep learning models through generative adversarial networks," 2022, accessed on: 22.06.2023. [Online]. Available: https://arxiv.org/pdf/2211.13991.pdf

[9] T. Courtat and H. d. M. d. Bourboux, "A light neural network for modulation detection under impairments," in *2021 International Symposium on Networks, Computers and Communications (ISNCC)*, 2021, pp. 1–7, accessed on: 20.06.2023. [Online]. Available: https://arxiv.org/abs/2003.12260

[10] A. Baldominos, Y. Saez, and P. Isasi, "A survey of handwritten character recognition with mnist and emnist," *Applied Sciences*, vol. 9, no. 15, 2019, accessed on: 20.06.2023. [Online]. Available: https://www.mdpi.com/2076-3417/9/15/3169

[11] K. Janocha and W. M. Czarnecki, "On loss functions for deep neural networks in classification," *CoRR*, vol. abs/1702.05659, 2017, accessed on: 20.06.2023. [Online]. Available: http://arxiv.org/abs/1702.05659

[12] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *CoRR*, vol. abs/1609.02907, 2016, accessed on: 22.06.2023. [Online]. Available: http://arxiv.org/abs/1609.02907

[13] "PyTorch colab notebooks and video tutorials," https://pytorch-geometric.readthedocs.io/en/latest/get_started/colabs.html, accessed on: 22.06.2023.

[14] F. He, T. Liu, and D. Tao, "Control batch size and learning rate to generalize well: Theoretical and empirical evidence," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019, accessed on: 20.06.2023. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2019/file/dc6a70712a252123c40d2adba6a11d84-Paper.pdf

Annexes

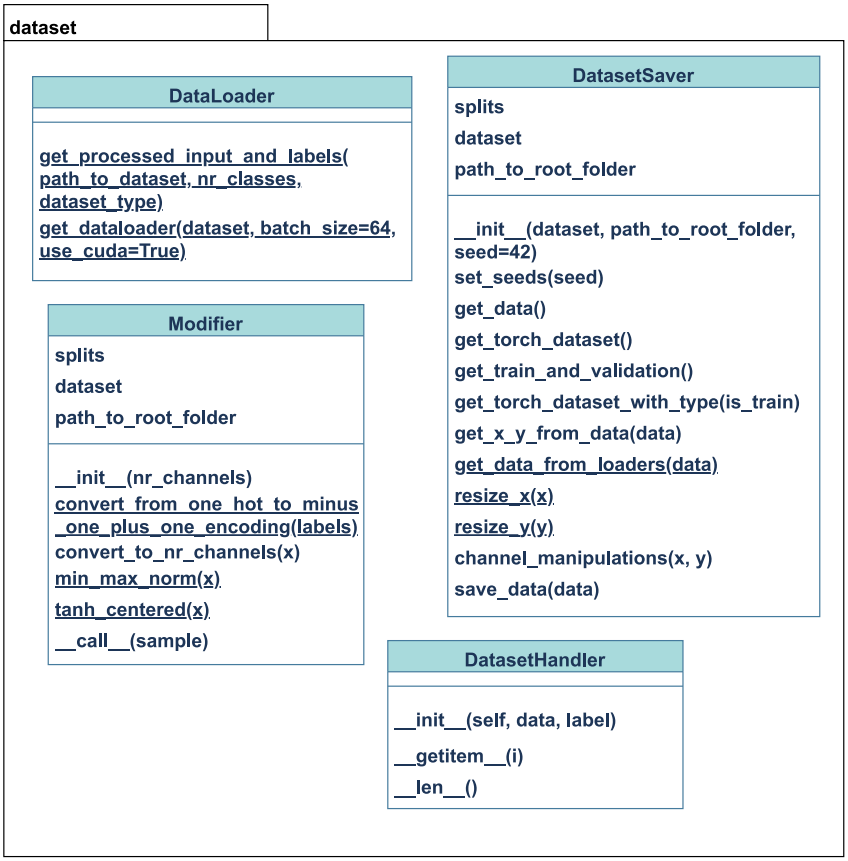


Figure Annex 1: UML diagram with the classes and their data members and functions from the dataset package from the refactored code

Figure Annex 2: UML diagram with the classes and their data members and functions from the training package from the refactored code

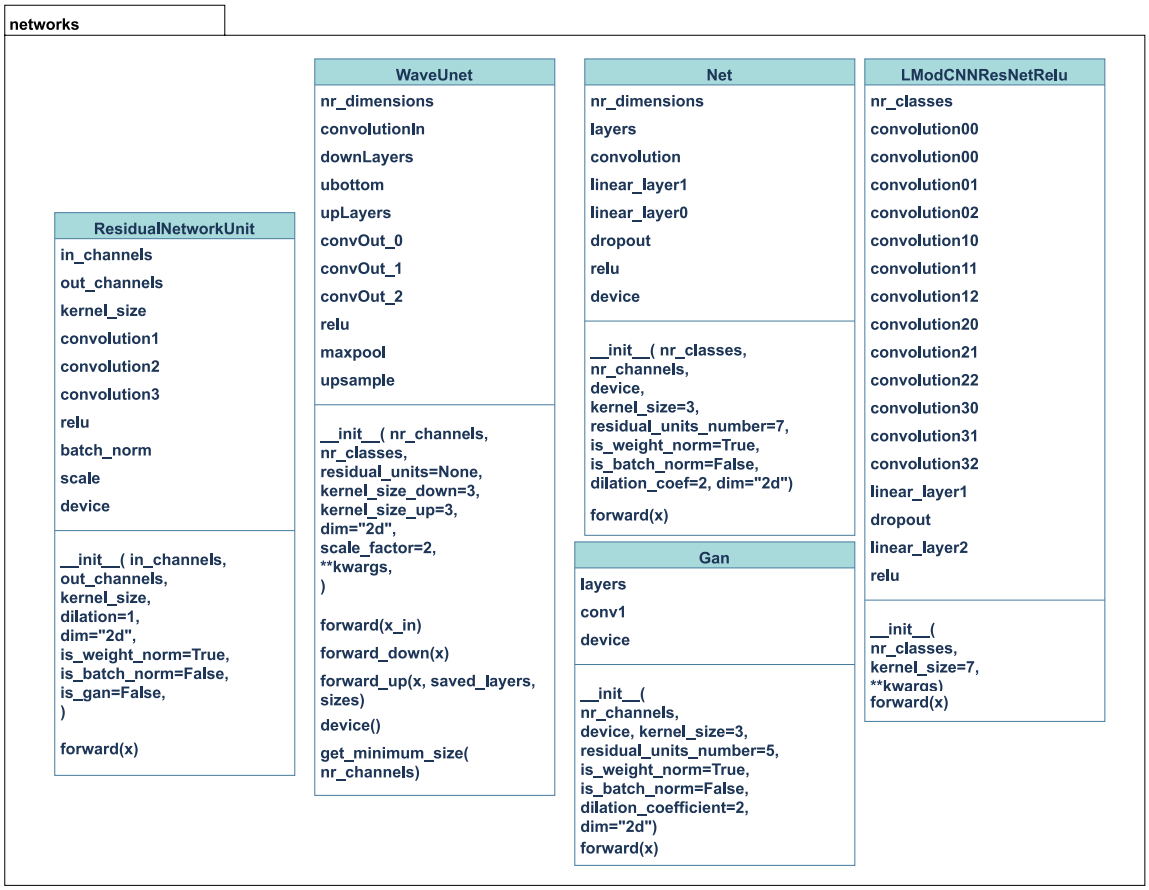


Figure Annex 3: UML diagrams with the classes and their data members and functions from the networks package from the refactored code

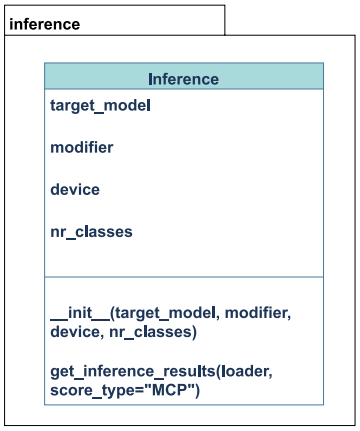


Figure Annex 4: UML diagram with the classes and their data members and functions from the inference package from the refactored code

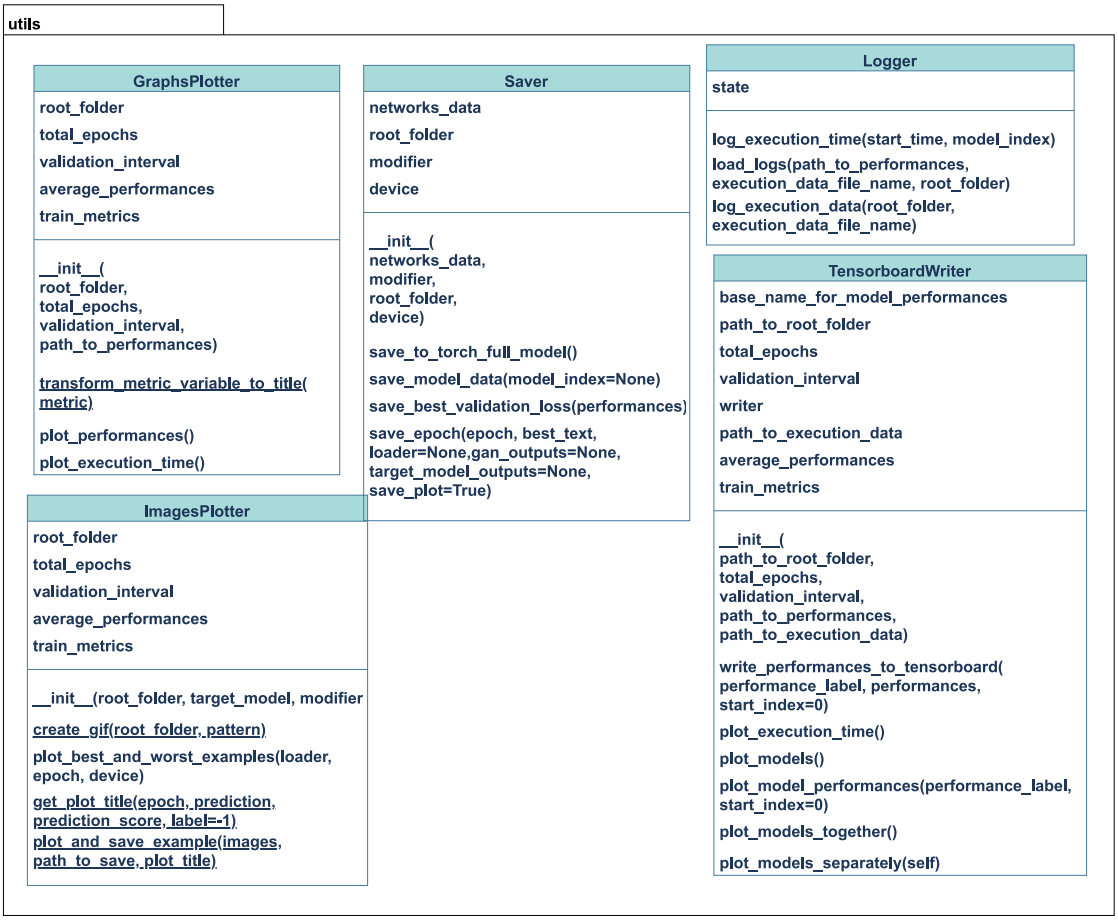


Figure Annex 5: UML diagram with the classes and their data members and functions from the utils package from the refactored code

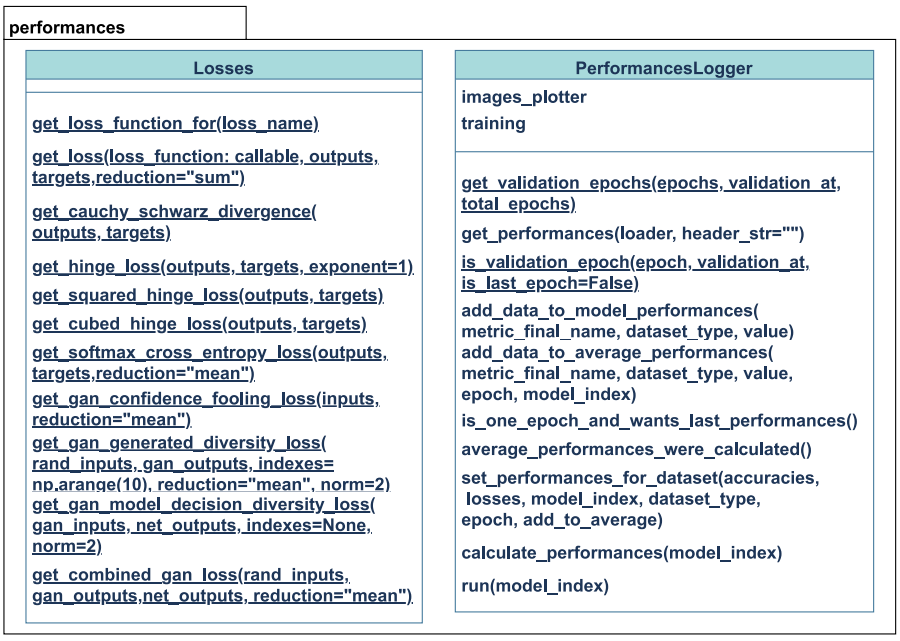


Figure Annex 6: UML diagram with the classes and their data members and functions from the performances package from the refactored code