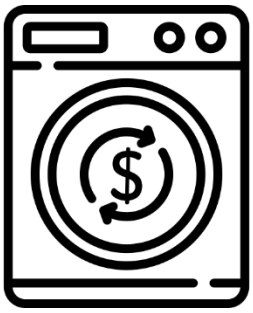

# A machine learning approach to money laundering detection inspired by GANs

Ricardo Ribeiro Pereira
Master's Degree in Computer Science
Computer Science Department
2021

**Advisor**
Pedro Manuel Pinto Ribeiro, Assistant Professor, Faculty of Science, University of Porto

**Supervisors**
David Aparício, PhD AI Researcher, OutSystems
Jacopo Bono, PhD Research Data Scientist, Feedzai

U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, _____/_____/_____

# Acknowledgments

First, I would like to thank my advisor and mentor, professor Pedro Ribeiro, as well as my supervisors David Aparício and Jacopo Bono, for all the guidance they offered during (and before) this project.

I would also like to thank Feedzai for the opportunity they gave me to work on this topic, and my colleagues at Feedzai for helping me throughout the project, namely Maria Inês Silva, Miguel Araújo, João Ascensão, Marco Sampaio, Ahmad Eddin, João Caetano and Pedro Bizarro.

Lastly, I am grateful to my loving family for their unconditional support through the years, which allowed me to pursue my studies and eventually get to where I am today.

# Abstract

Money laundering is the process of disguising the origins of illegally obtained money. It is used to fuel a large variety of other serious crimes, like drug and human trafficking. As such, banks are obligated by law to monitor and report suspicious activity to the authorities. Current detection systems usually consist of rules that catch different known patterns of money laundering. However, these rules are very simple and publicly available, since they are based on the applied legislation. As such, money launderers can easily adapt their behaviour to avoid them.

We propose a new machine learning solution to detect more complex patterns of money laundering that is inspired by the GAN architecture, which addresses the problem of lack of labelled data. The goal of the generator is to create synthetic money laundering activity that goes undetected by the rules, while the discriminator is trained to distinguish between real legitimate transactions and synthetic illicit ones. We show how our generator is able to launder hundreds of thousands of dollars without being detected by the rules, thus identifying weaknesses in the rule system. Also, the discriminator is able to reliably alert these synthetic transactions and can therefore be used as a complementary detection model. Overall, our system is able to detect new and more complex patterns of money laundering, which can have real world impact by reducing criminal activity.

# Resumo

Lavagem de dinheiro é o processo de esconder a origem de dinheiro obtido ilegalmente. Esta prática é utilizada para sustentar uma grande variedade de crimes, como o tráfico de drogas ou humano. Por esse motivo, os bancos são obrigados por lei a monitorizar e reportar atividades que considerem suspeitas às autoridades. Os sistemas atuais de deteção consistem, habitualmente, em regras que apanham diferentes padrões conhecidos de lavagem de dinheiro. No entanto, essas regras são bastante simples e conhecidas publicamente, uma vez que são baseadas na legislação vigente. Por estes motivos, os criminosos conseguem facilmente adaptar os seus comportamentos de forma a evitar que sejam apanhados.

Neste trabalho apresentamos uma nova solução de aprendizagem automática para detetar padrões de lavagem de dinheiro mais complexos, que é inspirada na arquitetura das GANs, o que resolve o problema da falta de dados etiquetados. O objetivo do gerador é criar atividade de lavagem de dinheiro sintética que não é detetada pelas regras, enquanto que o discriminador é treinado para distinguir transações reais legítimas das sintéticas ilícitas. Nós mostramos que o nosso gerador é capaz de lavar centenas de milhares de dólares sem ser detetado pelas regras, evidenciando, assim, as fraquezas do sistema de regras. Para além disso, o discriminador é capaz de alertar consistentemente estas transações sintéticas e, como tal, pode ser utilizado como modelo de deteção complementar. Em suma, o nosso sistema consegue detetar novos padrões de lavagem de dinheiro que são mais complexos, o que pode ter impacto no mundo real, reduzindo a atividade criminal.

# Contents

# List of Tables

# List of Figures

# Introduction 1

"Everyone wants to be rich". Some people achieve this goal through sources of revenue that are widely accepted as legal, others choose an illicit course of action. If they succeed, this second group is then faced with a problem: they cannot enjoy the spoils of their illicit activities right away. The assets they acquired can be traced back to their source, connecting the criminals to their crimes, increasing the risk of getting caught. In order to avoid this, criminals aim to conceal or disguise the origin of property that resulted from illegal activity, in a process known as money laundering [LR06]. Through money laundering, criminals are able to finance and profit from illegal activities such as: counterfeiting, drug trafficking, human trafficking, illegal gambling, prostitution, terrorism and theft. Due to the severity of the crimes involved and the countless people and economies affected, money laundering detection is a problem of utmost importance.

Money laundering is usually divided into 3 stages, in a process that begins with the spoils of the crimes ("dirty" assets) and ends with funds that can be used (somewhat) safely. The first stage is named *placement*, which involves the introduction of the unlawfully obtained property (usually money) into the financial system. This step can take several forms, some of the most common ones being structuring/smurfing (split cash in small amount deposits), currency smuggling (physically smuggling cash to another jurisdiction and depositing it there), camouflage (use stolen identity documents to open accounts and/or place funds) and cash-intensive businesses (make deposits in accounts of businesses whose revenue is typically cash, like car washes and casinos) [Sal05, ICL12]. The second stage is *layering*, a set of activities intended to distance the funds from their point of criminal origin, usually involving a complex network of financial transactions between different financial institutions and jurisdictions [SW08]. One commonly observed pattern during this stage is the use of mule accounts, which are bank accounts that function as nodes of the transaction network. Criminal organisations recruit people who open these mule accounts, which then receive high volumes of money from illicit sources and are instructed on how and where to transfer that

money. The third stage is *integration*, in which the illicit funds that have been cleaned can be spent or invested, usually in real estate, luxury assets or business ventures [FAT99].

In turn, anti-money laundering (AML) refers to the policies, laws, and regulations created to prevent financial crimes. Banks and other financial institutions are legally obligated to have prevention and detection methods in place in order to mitigate the problem of money laundering. However, they are not responsible (or able) to decide which transactions are deemed money laundering. Banks are responsible for monitoring their accounts and, if they detect some behaviour that may be indicative of illicit activities, they create what is called a Suspicious Activity Report (SAR) and submit it to the authorities. If they fail to comply with the policies in place, there are sanctions in the form of fines and the bank's reputation is also affected (in a practice known as "name and shame"). So, the main reason for banks to actively control money laundering is for compliance. This is different from fraud, for example, where banks are continuously losing money in compensations, so there is a more direct incentive to prevent and detect that problem.

However, detecting money laundering is a complex problem for various reasons. First, finding illicit activities among legitimate transactions is difficult given the sheer volume of data that you need to process. Second, the interactions between money launderers and banks can be viewed as an adversarial game, where the both parties are constantly changing their strategy in order to take the upper hand. So, the bank needs to continuously adapt to the different money laundering schemes from different entities in order to keep up. And third, money launderers usually make their money move across different institutions and jurisdictions in order to avoid that a single entity can monitor their entire operation. This is because banks and other financial institutions can only see (ergo monitor) transactions that involve their internal accounts (at least one of the counterparties is an account of the bank). For example, a cyclic pattern like the one depicted in Figure 1.1 is impossible to detect from the point of view of a single bank.

The AML measures put in place by the financial institutions are typically rule-based systems [WCS+18]. Using rules is an easy way to encode domain knowledge from analysts about known money laundering patterns, while being very interpretable, allowing transparency and oversight from the regulators. However, they have four big shortcomings: they are known to have high false-positive rates, they capture very simple patterns at the account level, they are easy to evade and they cannot easily adapt to changes in the strategies of the money launderers.

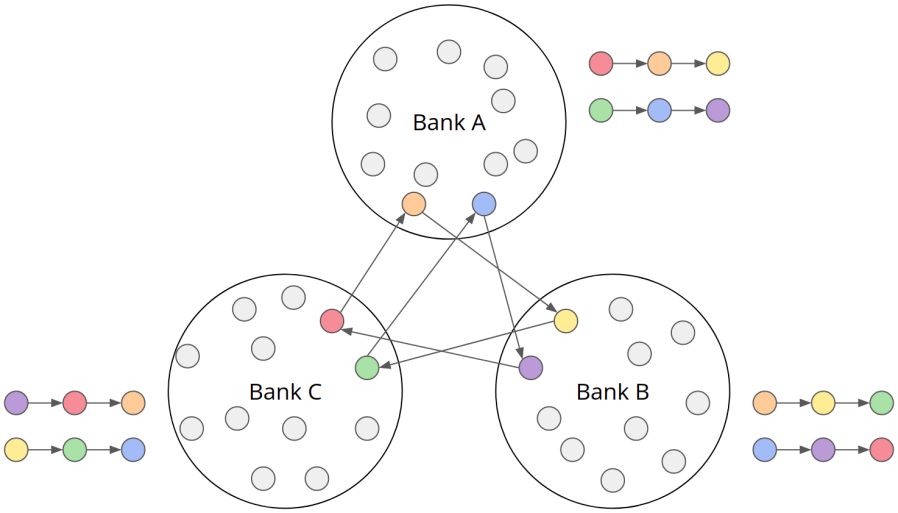


Figure 1.1: Example of a single cycle of transactions between accounts of different banks, which is viewed by each bank as two unrelated chains.

An alternative to rule-based systems are systems based on machine learning. Typically, machine learning systems are trained in a supervised fashion. However, supervised learning requires labelled data, i.e. enough examples of legitimate and illicit activities. The problem in the money laundering scenario is that large labelled datasets are typically unavailable. Alternatively, one may train unsupervised machine learning systems. These systems discover different patterns in the data which can then be analysed to distinguish between normal behaviour and outlier behaviour. Unsupervised techniques can be problematic for money laundering detection as well because criminal actors try to mimic legitimate activities in order to avoid detection [LSA+20].

## 1.1 Goals and Contributions

In this work, we describe an innovative way to create a money laundering detection system that is able to capture complex money laundering schemes which the AML rule-based systems cannot detect. These money laundering schemes are mimicking the money flow pattern of layering, which is characterised by high volumes of money being received and subsequently sent by mule accounts, usually from/to different banks. Importantly, we do not require any labelled data, since we simultaneously train an artificial money laundering generator, which has as a goal to maximise the money flowing through a bank while not being caught by the rules. These generated instances are used as positive examples of money laundering to train a deep learning classifier that distinguishes between real legitimate data and generated illicit data.

## 1.2 Thesis outline

This thesis is structured into 5 major chapters:

**Chapter 1 - Introduction** We introduce the topic of money laundering detection, giving some required domain knowledge and briefly covering the main types of solutions and their shortcomings. We also give a very brief description of our solution and the thesis outline.

**Chapter 2 - Related Work** We provide an overview of different solutions that were proposed to solve the problem of money laundering detection, as well as a description of generative models that served as an inspiration for our solution.

**Chapter 3 - Method** We present the method that we propose. Each section covers one component of the complete architecture in detail.

**Chapter 4 - Results** We describe the experimental setup we used to evaluate each component of our method, as well as the main results from testing the complete architecture.

**Chapter 5 - Conclusions** We conclude the thesis by summarising its contributions and findings, and list possible directions that could be followed as future work.

# Related Work 2

In this chapter, we present an overview of money laundering detection techniques, as well as types of generative models, specially different GAN architectures that served as inspiration to our method. We should mention that, to the best of our knowledge, there is currently no work on the topic of generation of realistic money laundering examples.

## 2.1 Money Laundering Detection

### 2.1.1 Rule-based systems

The most common method employed by banks to do money laundering detection is through the use of rules [WCS+18, SWC+16, WRD+03], which are a set of deterministic conditions to be verified based on the features of a transaction or account. This method has two main advantages compared to other options. First, it is easy to encode domain knowledge from analysts into the solution, since they can design different rule scenarios that capture different known patterns of money laundering. Second, rules represent an interpretable solution, which is important in order to show that the bank is compliant with its jurisdiction's regulations. Furthermore, rules systems have been argued to be less prone to over-reporting than risk based systems, as well as to be preferable for litigation purposes (since a risk based system leaves space for ambiguity) [UVW09].

However, rule-based systems have four big shortcomings. First, such systems are characterised by high false-positive rates (FPR), which means that the rules create many alerts for non suspicious transactions, leading to large volumes of alerts being investigated by analysts unnecessarily. Second, they are too simple to capture complex money laundering schemes, since most rules are evaluated at a single account level. Third, since these rules are based on publicly available legislation to which the bank must comply, it is easy for the money launderers to adapt their behaviour in order

to avoid getting caught Fourth, there is no easy way to adapt a rule-based system to changes in the strategies of the money launderers [WRD+03].

### 2.1.2 Machine learning techniques

In order to mitigate these problems, there has been some work developed towards machine learning-based solutions. A review of machine learning methods for AML was presented in [CTN+18], which divides them into unsupervised and supervised techniques, depending on the type of machine learning algorithm used. Recently, active learning and graph-based methods have been proposed to address the perceived shortcomings of traditional approaches.

**Unsupervised methods** Given the lack of labelled data, the majority of the solutions are based on unsupervised methods. This includes techniques like clustering [AB15, AB16, CNT+14, LH11, LCQ+17, SNY+16, WD09], in which data points are grouped into different categories; and anomaly detection [CSMV17, Gao09, PL16, RH11, TY05], which identifies data points that deviate from the dataset's normal behaviour. Although anomaly detection assumes illicit behaviours to be rare and differ significantly from legitimate ones, that assumption may not hold whenever money launderers intentionally mimic legitimate behaviour. [LSA+20] argue that, for this reason, the flexibility of unsupervised methods may be offset by a decrease in detection.

**Supervised methods** In order to try to improve upon the detection of the unsupervised models, one could choose a supervised method instead. Supervised learning classifiers applied to AML include Logistic Regression [WDC+19], Decision Trees [WY07], Random Forests [JLH+20, SWC+16, WDC+19], Bayesian Networks [KLRH13, RH11, Wag19], Neural Networks [LJZ08, Wag19], Scan Statistics [LZ10] and SVMs [DJSW09, KT11, TY05, Wag19]. [WDC+19] conclude that Random Forest outperforms several other methods in the context of binary classification for financial forensics in a graph network of Bitcoin transactions. [OTS+21] expand on these results by engineering features that leverage graph information and train a Random Forest on the enriched data. However, supervised techniques require large-scale labelled data. In practice, AML labels are typically incomplete, delayed, and expensive. Incomplete because money laundering schemes are often undetected, so assuming correct labels for all money laundering examples is unrealistic. Delayed because investigations are highly complex and ground truth is not immediately available. Expensive as manual annotation is costly and requires large teams of expert AML analysts reviewing examples.

Because of these reasons, most of these works assume that real data corresponds to legitimate activity and use synthetic positive examples to train the supervised models.

**Active learning** In order to mitigate the problem of having few and expensive labels, [DJSW09, LSA+20] propose to collect labels for a subset of examples through an active learning policy that samples the most informative examples from the unlabeled data. In their work, [LSA+20] claim to match the supervised results, while labelling approximately 5% of the examples, thus providing a cost-efficient alternative to both supervised and unsupervised learning. The active learning setup closely follows the system detailed in [BLP+21].

**Graph-based methods** In AML datasets, observations are typically interrelated. Nonetheless, traditional machine learning techniques fail to leverage graph information such as an entity's transaction history or its relationships with other entities. A potential solution is to engineer features that convey that information to the model [CR17, OTS+21]. Recently, however, novel methods have been proposed that automatically extract useful features. Recurrent Neural Networks (RNN), for example, can be used on sequences of financial transactions [BAG+20]. Graph Convolutional Networks (GCN) have been proposed in the context of AML by [WCS+18, WDC+19]. [WDC+19] benchmark GCN against commonly used supervised methods and concludes Random Forest to provide better performance, despite lacking graph information. Finally, [LLL+20] argue against dense subgraph detection and focus on multi-step flows instead. The authors propose FlowScope to identify dense flows in large transaction graphs and benchmark it favourably against several other graph-based approaches. [SZZ+21] improve upon this work and propose CubeFlow so that the flow of funds can be combined with relevant attributes, such as transferring timestamps. These last two works have similarities with our proposed method, both in terms of goal and data representation. However, they employ a greedy solution to find these money flows and they are both static (unable to learn).

## 2.2 Generative Models

A powerful approach to data representation in Machine Learning consists of using parametric generative models to estimate the joint distribution of the variables in a given dataset [BTLLW21, RH21]. Since such parametric models aim at encoding maximal information on the data distribution, even if they are not analytically tractable, they often offer the advantage of allowing to sample from and/or evaluate the learned

joint distribution or its marginal distributions. In turn, this allows for various other tasks, such as estimation of relevant expectation values, data augmentation/synthesis, out-of-distribution detection or classification, just to name a few. These generative models can be broadly split into two types of approaches to model the distribution of instances $x \sim p(x)$, namely:

- latent space models that assume a map (generator) $x = G_\theta(z)$, with parameters $\theta$, from instances $z$ following a known distribution $z \sim p(z)$
- models that directly assume parametrizations for $p(x)$ or its conditional distributions

Since the proposed solution in this work is inspired by latent space models, we now summarise their main approaches.

**Normalising Flows** (NFs) These assume an invertible and differentiable generator, thus allowing to obtain the log likelihood of the data via a change of variables from the unknown target space distribution to the latent space distribution. Intuitively, this type of model can be seen as a sequence of parametric invertible deformations of a known distribution to the unknown distribution. Its main shortcoming is that the latent space has to be of the same dimension as the target space representation of the data, which is often not true [DKB14, RM15].

**Variational Auto Encoders** (VAEs) These drop the invertibility assumption of normalising flows and typically assume a latent space of lower dimensionality. However they do not parametrise the generator function directly, but instead the likelihood $p_\theta(x|z)$. Then a tractable proxy for the objective is obtained by introducing a second parametric approximation to the posterior $q_\phi(z|x) \approx p_\theta(z|x)$, with parameters $\phi$ [KW13].

**Generative Adversarial Networks** (GANs) This method focuses on comparing the distribution of generated instances and the distribution of real instances via sampling using an auxiliary parametric discriminator model. Thus, unlike NFs or VAEs, it does not attempt to compare and parametrise distributions in latent space nor the likelihood, and only parametrises the generator function. Because of this, GANs do not allow for the evaluation of distributions directly but only to generate samples from the (implicitly) learned distribution [GPAM$^+$14].

Since GANs focus on an optimization objective defined in the space of real samples, they are particularly well suited to be adapted to our optimization goal. Instead of learning to approximate the distribution of the real data, we aim to constrain the generated samples to money flows that evade rules systems and maximise the amount of money flowing, which can be a part of or different from the distribution of real data. Therefore, in the remainder of this section we focus on reviewing some aspects of GANs.

### 2.2.1 GANs

The Generative Adversarial Network (GAN) architecture was first suggested by Ian Goodfellow in [GPAM+14]. It consists of two parametric models (typically deep neural networks): the generator $G_\theta(z)$ and the discriminator $D_\phi(x)$. $G$ is mapping the latent space variable $z$ into the space of the real data. $D$ is a discriminative model that estimates the probability that $x$ comes from the real data rather than from the distribution induced by $G$. We train $D$ to maximise the probability of correctly classifying the real and the generated examples. By contrast, $G$ is trained to minimise this probability, which means that the generated data should approximate the distribution of real data. As such, these models have opposing goals and the training process is said to be adversarial. The typical training approach is to alternate between optimising the parameters of the generator and of the discriminator (usually via stochastic gradient descent) until the generated data is indistinguishable from real data and the discriminator is randomly guessing the class of the examples [CWD+18].

The objective function used to train the GAN that was proposed by [GPAM+14] was

$$V(G, D) = E_{x \sim p(x)} \left[\log(D(x))\right] + E_{z \sim p(z)} \left[\log(1 - D(G(z)))\right] \tag{2.1}$$

This is equivalent to minimising the Jensen-Shannon divergence between the distributions of real and of generated data. Despite the initial successes of this objective function, various limitations have been observed, such as training instabilities (e.g due to lack of gradients for $G$ to learn) or mode collapse (i.e., the generator only learns specific modes of the distribution) [GKL+21, GSW+20, SGZ+16]. Thus various modifications of the objective function have been proposed. One such proposal is the Wasserstein GAN (WGAN) by [ACB17], which modifies the objective function such that the distance between the distributions of real and generated data is measured by the Earth Mover Distance, which they show to help mitigate those two problems. Overall, the objective function is a choice that is largely independent of the network architecture used to implement the generator and the discriminator.

Another important ingredient that may limit the learning capacity of a GAN is the architecture of $G$ and $D$. That is specially relevant whenever important structural inductive biases or symmetries need to be imposed on the generated data. Earlier architectures based on fully connected neural networks, in many cases, have too many free weights and biases, thus leaving it to the learning process to discover such structures from the data, which can make the training slow, unstable or even impossible. In order to reduce the amount of parameters to be learned and to take advantage of the structural patterns of real images, [RMC15] suggested the use of convolutional layers as the basis for the GAN architecture. They also found that batch normalisation helped significantly to stabilise the training procedure and they suggested using the ReLU activation function in $G$ for all layers except for the output (which uses tanh) and the LeakyReLU in D for all layers. The GAN that we use in our method draws several architectural choices from their work.

Lastly, [DCK18] suggested a setup that has several similarities with our proposed system, even though it is applied in a different setting and with a different optimization objective. Their goal was to generate molecules with specific desired chemical properties. They represent molecules as a graph with colored nodes and edges, and train a GAN combined with a reinforcement learning (RL) objective to generate them. The discriminator ensures that the generated samples resemble real data and therefore constitutes valid compounds, while the RL policy rewards the generator if the synthetic samples verify some desired properties (e.g., to be easily synthesizable). The generator consists of a Multi Layer Perceptron, while the discriminator and the reinforcement learning objective are Relational Graph Convolutional Networks with the same architecture but independent parameters. The discriminator is trained to distinguish between real and fake molecules, using the gradient penalty WGAN objective, the reward network is trained to assign a score to each molecule (supervised regression model), and the generator loss is a linear combination of these losses.

# Methods

# 3

In this work, we present an AML method designed to capture suspicious behaviour that escapes rule-based AML systems. As discussed previously in Chapter 1, it is difficult to apply machine learning systems to the problem of money laundering detection, given the lack of labelled data. A key element of our setup is that we do not need labelled data, but instead exploit our money laundering domain knowledge to jointly optimise an artificial adversary. This adversary generates synthetic transactions that avoid triggering the rules while the new complementary AML system is trained to distinguish these generated examples from real ones.

The specific pattern of money laundering that we are mimicking in this work is the money flow. It is a layering pattern in which high volumes of money are received and subsequently sent by the mule accounts, creating a long chain of transactions between the illicit source of the money and its final destination. In order to avoid being detected, the transactions in this chain are usually made between different financial institutions, often across different jurisdictions.

There are 4 main problems that we need to solve in order to make this architecture work:

- How do we represent transactional data?
- How to learn to avoid triggering the rule system?
- How to ensure that we create money flows?
- How do we generate synthetic examples?

The method that we propose consists of different components working together (see Figure 3.1), each of which solving one of the aforementioned problems. They are:

- A generator responsible for creating synthetic examples of money flows. The goal of the generator is to maximise the amount of money flowing through the bank without being detected by the AML system.
- A discriminator responsible for distinguishing generated samples from real data samples. The goal of the discriminator is to capture the synthetic samples (i.e. our proxy for money laundering activity).
- A differentiable version of the rule-based system, able to provide gradient information to the generator. In this way, the generator can learn to avoid triggering the rule-based system.
- A money laundering objective function, which incentivizes the generator to make money flow through the bank.
- A sampling strategy capable of selecting representative legitimate transactions which (together with synthetic samples from the generator) are used to train the discriminator.

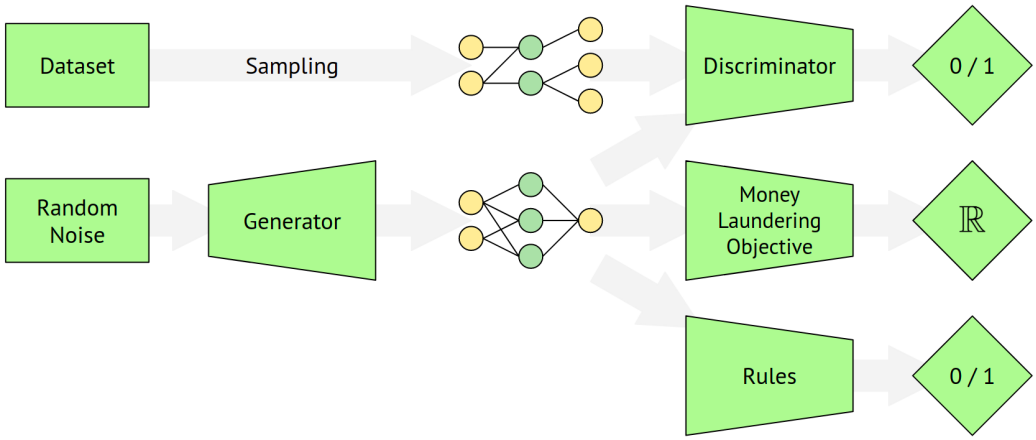


Figure 3.1: Schematic representation of how the different components in our method fit together.

We discuss each of these components in depth in the following sections. We start by describing how we represent the transactions that the generator creates and that are fed to the discriminator, to the money laundering objective function and to the rules evaluation component, in Section 3.1. Then, we cover the fixed components of our method, that will provide feedback to the generator, separately, i.e., the rules component in Section 3.2 and the money laundering objective function in Section 3.3.

After that, we describe the generator in Section 3.4, whose training is guided by these fixed components. Before providing details about the discriminator, we explain which sampling strategy we use to obtain real samples for the training of the discriminator in Section 3.5. Finally, we describe the discriminator in Section 3.6, which serves as a new machine learning model component to complement the conventional rule-based AML system.

## 3.1 Data representation

The money laundering pattern that we are trying to capture, the money flow, can be categorised using little information about each account, since it relates to the structure of the transactions instead of the attributes of each individual account. In order to build the transaction graph, we need to know which accounts are involved in each transaction, when they happened and how much money was sent. Because of this, we only need to have four features per transaction: the id of the sending account, the id of the receiving account, the amount being transferred, and the timestamp of the transaction.

As we previously discussed, we are trying to catch money flows that are indicative of layering, which is characterised by complex networks of transactions between different financial institutions. However, from this complex network, each individual bank only has records of transactions that involve its internal accounts. Therefore, one might have some information of transactions between external accounts and the bank's internal accounts, but no information about external accounts that do not interact with internal accounts at all (see Figure 3.2).

In terms of types of accounts involved, we can divide these transactions into three types: an external account (source account) sending money to an internal account, an internal account sending money to another internal account, and an internal account sending money to an external account (destination account). From these three transaction types, we ignore the internal-internal transactions since the layering pattern of money flows usually involve accounts from different financial institutions. Therefore, we can represent the relevant transactions as a tripartite graph, where a set of external accounts transfers money to a set of internal accounts, which in turn transfers money to a set of external accounts. The edges in this tripartite graph represent transactions between the corresponding pair of accounts and the weight of the edge is the amount transferred.

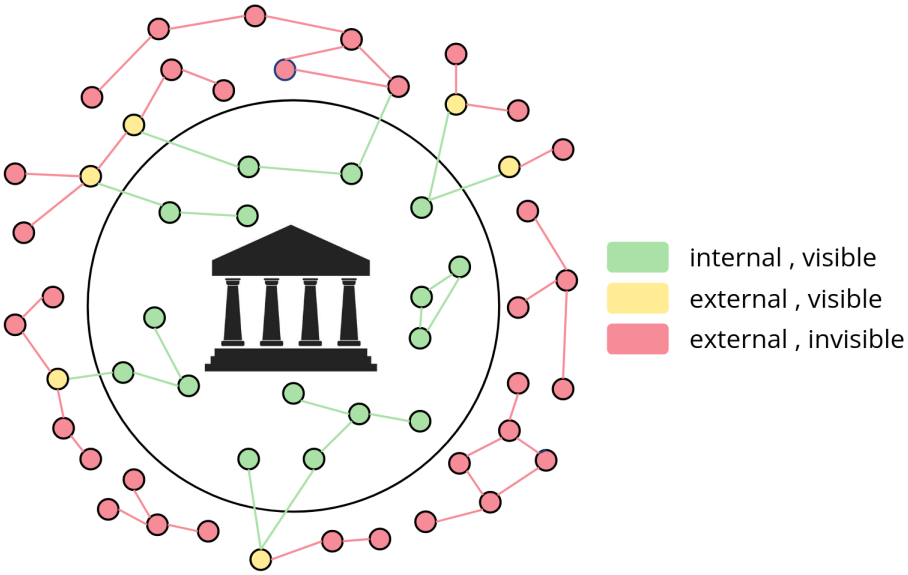


Figure 3.2: Example of a bank's visibility of the transaction graph. The nodes represent the accounts and the edges represent the transactions. The bank can only see its internal accounts (green nodes) and external accounts that make transactions with its internal accounts (yellow nodes). Every red account and red transaction are invisible to the bank.

Usually, in the AML domain, each account is monitored periodically (e.g. on a daily basis). A set of profiles is gathered, consisting of temporal aggregations of statistics of that account's behaviour (e.g. "sum of amounts sent in the past week"), which is then used to decide the triggering of the rules. Given that the smaller time unit used for these profiles and the monitoring period is not smaller than a day, we choose to discretize our data into time units of days. All the transactions between a given pair of accounts that fall in the same time unit are merged together and represented as a single edge with weight equal to the combined total amount transferred in those transactions.

We use a 3D tensor to represent this weighted tripartite graph with time information, as is depicted in Figure 3.3. Let M denote the size of the first dimension. Each index in this first dimension corresponds to a different internal account. Let S and D denote the number of external source/destination accounts that will send/receive money to/from internal accounts, respectively. Let S+D denote the size of the second dimension, since these sets do not overlap by design. Each index in the second dimension corresponds to a different external account (with source accounts in smaller indexes and destination accounts in larger indexes). Let T denote the size of the third dimension. Each index in this third dimension corresponds to a different day of the dataset, with the smallest

index corresponding to the most recent day of the dataset and the largest index to the oldest. The value in entry $(x, y, z)$ is the total sum of amounts of the transactions made between the internal account $x$ and external account $y$ on day $z$. If there are no transactions that fall on those coordinates, the entry will be 0.

| Source | Target | Amount | Day |
|---|---|---|---|
| α | A | 3.14 | 1 |
| α | A | 15.92 | 2 |
| α | B | 65.35 | 1 |
| β | A | 89.79 | 1 |
| A | B | 32.38 | 2 |
| A | ɣ | 46.26 | 2 |
| B | ɣ | 43.38 | 1 |
| B | δ | 32.79 | 2 |
| B | δ | 50.28 | 2 |

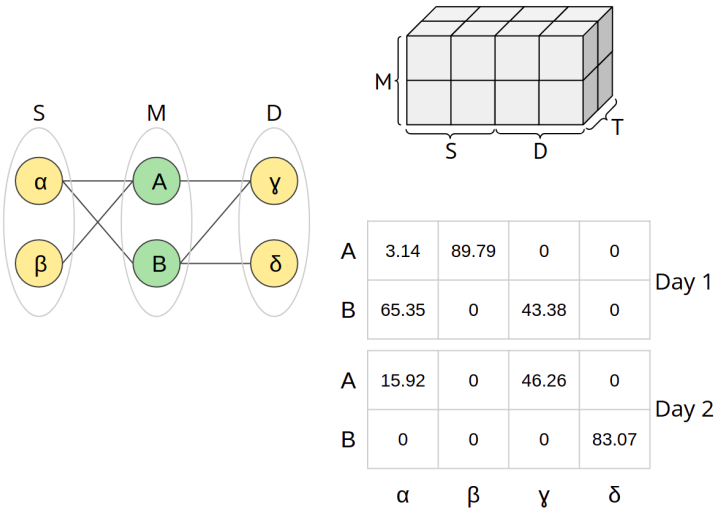


Figure 3.3: Example of transactions represented as a tensor. The table on the left shows an example of transactions. Latin letters stand for ids of internal accounts and Greek letters for ids of external accounts. There are no records of external-external transactions, since that information is not available to the bank, and internal-internal transactions were ignored. In the center, we can see the corresponding tripartite graph of interactions between accounts. On the top right corner, we show how this graph is represented as a 3D tensor, with the values of each cell of the tensor shown below it.

We use this representation to avoid having to generate sequences of transactions with variable size or having to directly predict timestamps for each transaction. This way, we know exactly the format of the data that we will generate and classify, allowing us to use any type of deep learning models in our method. This simplifies the generator's architecture, since we can use dense or convolutional layers, instead of recurrent units or other time series tools. Also, since the smallest time unit that the AML rules need to use to calculate profiles is one day, we do not need more time granularity. The main drawback is that it limits the size of the output that we are generating, e.g. if $M = 5$ then the generator can use at most 5 internal accounts. However, we know from consulting domain experts that more than 90% of suspicious activity reports contain up to 5 unique accounts, so this limited number of accounts should still be enough to represent the majority of money laundering strategies.

Overall, the output of the generator and of the sampling strategy, as well as the input of the discriminator, money laundering objective and rules will all have the same format: a 3D tensor of non-negative real numbers with shape $(M, S + D, T)$, where the value of each entry is the amount of money being transferred between two corresponding accounts on the corresponding day.

## 3.2 Rule Proxy Network

Rule-based AML systems usually consist of several scenarios that capture different patterns of suspicious activity. The rule-based AML system with which we work is no different, but given the limited set of features per transaction that we use, only seven of those scenarios apply to our setting.

1. Scenario 1 compares the account activity in the recent past month with the activity in a longer prior time period, to detect if there is a significant change of the behaviour.
2. Scenario 2 detects accounts that are receiving and sending high volumes of similar amounts in short periods of time.
3. Scenario 3 detects accounts that frequently make round-amount transactions, defined by the number of trailing 0's of the amount.
4. Scenario 4 triggers when an inactive account (an account that does not make any transaction in a long period of time) makes a transaction.
5. Scenario 5 triggers when the total amount of money transferred in a short period of time is above some threshold, usually imposed by the bank's jurisdiction.
6. Scenario 6 serves as a buffer for the previous scenario, triggering when the total amount of money is significantly close to the legally imposed threshold, but strictly below it.
7. Scenario 7 detects accounts that make many transactions with a large number of distinct counterparties.

Even though we used this specific rule-based AML system throughout this work, the method we propose can be applied to any set of rules that are based on a set of features mentioned in Section 3.1. Other rule scenarios based on other types of features could be used as well, but it would require changing the data representation.

We want the generated transactions to not trigger any rule scenario, because the transactions that trigger the rules are already being caught by the existing AML system. In order to improve upon such a system, we need money laundering data that avoids these triggers. In a way, we are finding the weak points in the current AML system by generating synthetic money laundering data that is able to go undetected, and then training the complementary system (the discriminator) to catch those money laundering strategies.

The generator can only learn to avoid triggering the rule-based AML system if it can get feedback on how well it is doing it. The problem with using the rule-based solution directly is that, since the generator is a deep learning model trained with backpropagation, this feedback cannot be just a label, we need a gradient to do gradient descent. However, if the rules are implemented as simple if-then-else statements, their output is a piecewise constant function of their input, which will result in zero gradients with respect to the input almost everywhere. Thus, when receiving input from a parametric generator, if we take derivatives with respect to the generator's weights, as is conventional in the training of deep learning models, the gradient will be zero almost everywhere. Because of this, the numerical training of the network through backpropagation will not be possible. Our solution to this problem is to smooth out the piecewise rules function by training another neural network to serve as a proxy for the rules (given an instance, it approximately attributes the same label as the original rules). The gradient from this proxy's loss can indicate in which direction in the space of inputs the function is decreasing, thus providing feedback to the generator. The output of the generator will then be an input to this proxy network. In this way, if the generator's output triggers any scenario of the rule-based system, the proxy network will provide gradient information to the generator on how to update its learnable parameters, in order to avoid being detected again in the future.

The proxy network classifies each internal account, in each day, according to if it triggered any of the scenarios of money laundering (this is the same procedure as the typical rule-based AML systems). This proxy network receives a 3D tensor $\mathbf{T_i}$ of shape $(M, S + D, T)$ as input (as described in Section 3.1) and outputs a 3D tensor $\mathbf{T_o}$ of shape $(M, R, T)$, where $R$ is the number of rule scenarios (in this work, $R = 7$). The entries in its output have value close to 1 iff the corresponding internal account triggered the corresponding rule scenario in the corresponding day, otherwise they are close to 0. For example, if a given set of transactions from the second internal account would only trigger the third scenario on the seventh day, then every entry of the output tensor should be 0, except for entry $(1, 2, 6)$ that should be approximately 1.

Another simpler option is to output a tensor that simply predicts whether there was any rule triggered. However, not knowing exactly which scenario triggered the rules discards important information, such as: are the transferred amounts too high? are there too many connected accounts involved in the scheme? are there too many transactions between the fraudulent accounts? Not only does this affect the generator model that receives a possibly less detailed gradient, but also makes it harder to debug and/or improve the proxy network. Because of this, we choose the option of outputting a 3D tensor of shape $(M, R, T)$.

For each account and for each day, the rule proxy network outputs a score indicating which rules triggered (if any). These labels are treated as independent from each other, meaning that the network's prediction for one entry of the output tensor does not directly affect the predictions of other entries. This is because the rule scenarios are computed independently from each other and the rules are scheduled once a day every day.

The network architecture is divided into two sequential blocks: profiling and prediction.

In the profiling block, the rule proxy network aggregates information from the input tensor in order to calculate profiles for each internal account. For example, one of the profiles extracted could be "sum of amounts sent in the past 7 days". This is achieved using convolutional layers that slide across either the second dimension (counterparties) or the third dimension (time). We explain this profiling block with more detail in the following sections.

In the prediction block, the rule proxy network combines the profiling features to decide if they meet the required conditions to trigger each of the rule scenarios. Again, now that we have the profiles, each decision can be made independently, always using the same logic, for every point in time. As such, we can implement this as convolutional layers that slide across the time dimension, always making the same combination of profiles to make the prediction.

To implement the network architecture described above, it is important to note that, unlike a conventional classification problem, where a set of data points with labels is provided, sampled from an unknown distribution, here we know the exact logic of the rules. Therefore, we considered several options to include inductive biases, based on such logic, to ease the convergence of the learning process. The first option is to have both the profiling and prediction blocks fully learnable, meaning that the network learns all the weights and biases from the examples (conventional supervised

learning). The second option is to implement the logic of the profiling block, meaning that we set the parameters (weights and biases) of the first layers to some fixed values that correspond to the calculation of the known required profiles, while having the prediction block learnable. The third option is to manually implement the logic of the profiling block as well as the prediction block.

We explore different architectures for the proxy network because, as we increase the inductive bias encoded in the architecture, there is a trade off between the increase in architectural complexity and possible increase in performance. The more the network is learnable end-to-end, the easier it would be to build the architecture, since we can initialise all parameters randomly and let it learn and improve with the training data. The more the network is fixed, the more tricky it is to build the architecture, since each profile and condition should be translated precisely into the parameters of the network, which means that there is no training necessary. The performance of the fixed one is guaranteed to be near perfect if constructed well, while the performance of the learnable one can be very good as well or not.

We will now describe the three aforementioned versions of the architecture.

### 3.2.1 Fully Learnable Architecture

In this version of the rules proxy network, every parameter of the network is learnable, so we only need to tune hyperparameters, e.g. number of layers, sizes of the kernels.

In the profiling block, we introduce some inductive bias by choosing sizes for the convolutional filters that correspond to counterparty dimensions and time windows required by the rules. For example, since we know that the rules use profiles based on total amount received or sent by an internal account, the filters sizes should match the number of source or destination accounts and never mix some of these two types of accounts. Furthermore, the sizes of temporal filters are also chosen to be natural time intervals (one week, one month, etc). By using combinations of these values as kernel sizes, the first layers should learn to extract profiling features with real meaning. The output of this set of layers is a 3D tensor $\mathbf{T_p}$ of shape $(M, P, T)$, where $P$ is the number of profiles extracted from the input.

In the prediction block, we use convolutional layers with 1D kernels, with length equal to the number of features of the previous layer, that slide across the internal account and time dimensions of the tensor. Each of these kernels combines the profiles that we previously calculated in order to attribute a score per account per day, always applying the same logic. The output of the proxy network is tensor $\mathbf{T_o}$ with the predictions.

### 3.2.2 Semi Learnable Architecture

Another option is to implement the profiling block manually (i.e., with fixed weights and biases) and then have learnable layers in the prediction block. Using this architecture, we force the first layers of the network to calculate the same profiles that the rules use. As such, we can expect better predictive performance since we have less parameters to learn and the information is optimally pre-processed. Using scenario 2 as an example, it uses two profiling features to make its decision: total amount of incoming/outgoing transactions in the last 7 days. If the first learnable layers have access to these values directly, it is much easier to learn a decision based on them. In contrast, the fully learnable architecture would firstly need to learn to extract these profiles and then learn to make a decision based on them, potentially resulting in compounding errors.

In the profiling block, in order to mimic the rule-based AML system, we need several combinations of total amounts or number of transactions, from or to external accounts, during various time periods. Starting from the original input tensor $\mathbf{T_i}$, we get tensor $\mathbf{T_{p0}}$ of shape $(M, 6, T)$ , where the second dimension contains:

1. The number of incoming transactions,
2. The total dollar amount of incoming transactions,
3. The number of outgoing transactions,
4. The total dollar amount of outgoing transactions,
5. The number of transactions with round amounts,
6. The dollar amount of transactions with round amounts.

We need to count the number of transactions in a differentiable way. We do this by mapping entries with a positive amount to a value close to 1, while leaving the empty ones unaltered. After this mapping is done, counting the number of transactions becomes a simple addiction of the values of the entries. We can do this mapping using the function from Eq. 3.1, where $ReLU$ is the Rectified Linear Unit function and $S$ is the Sigmoid function. $k$ is used to rescale the x axis in order to map smaller values closer to 1, but since the amounts of the transactions are usually larger than 5, we can set $k = 1$. Figure 3.4 shows this function when $k = 1$.

$$f_1(x) = ReLU(2S(kx) - 1) \tag{3.1}$$

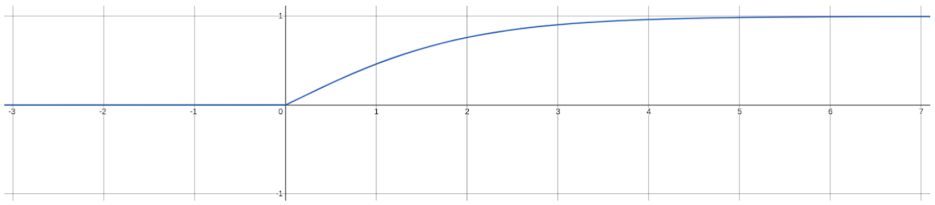

Figure 3.4: Plot of the function that we use to map the amount values into counts.

We need to find the round amount transactions (i.e. transactions with an amount that is a multiple of 10000) in a differentiable way. We do this by mapping entries whose amount is close to a multiple of 10000 to a value close to 1, while the others are set to 0. After this mapping is done, we can count round amount transactions using the mapped values directly, or we can select the original round amounts by multiplying $\mathbf{T_i}$ by the mapped values (effectively setting non round amount entries to 0). We can do this mapping using the function from Eq. 3.2, where *ReLU* is the Rectified Linear Unit function. $v_1$ is the factor that defines round amounts, which in our case is 10000. $v_2$ is controlling how close a value should be to a multiple of $v_1$ in order to be considered a round amount. Setting $v_2 = 500$ gives $\sim$1% of $v_1$ as the maximum distance to a round amount, which was what we chose. Figure 3.5 shows this function when $v_1 = 10000$ and $v_2 = 500$.

$$f_2(x) = ReLU(v_2 \cos\frac{2\pi x}{v_1} - (v_2 - 1)) \tag{3.2}$$

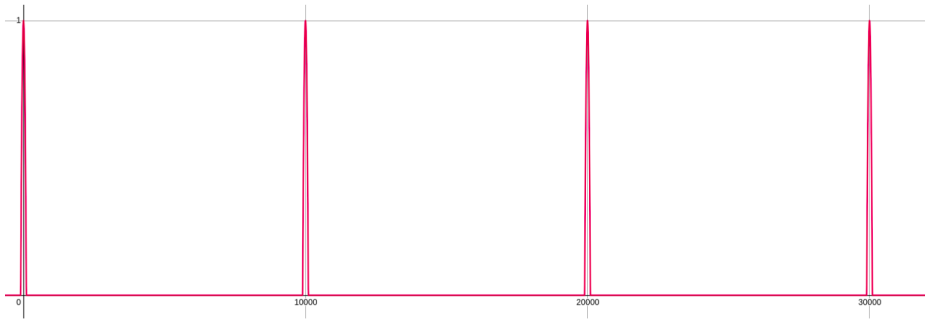

Figure 3.5: Plot of the function that we use to detect round amount transactions.

These two functions serve the purpose of calculating what we need (counts and round amounts) in a differentiable way, which is a necessary property for deep learning methods, because we need a gradient to train. We chose simple functions that have the desired effect and chose sensible parameters. We found that they fulfil their purpose, so we did not explore how changing these transformations or their parameters affect our method.

Next, we aggregate over the desired time windows, namely one week, two weeks, one month, six months and ten months. Each of them is implemented as a convolution filter of the desired size, moving along the third dimension of the $\mathbf{T_{p0}}$ tensor, as we exemplify in Figure 3.6. Notice that in order to keep the temporal dimension with the same size, we require padding the tensor in one side, with a number of entries equal to the length of the kernel minus 1. The result is tensor $\mathbf{T_{p1}}$ of shape $(M, 30, T)$.

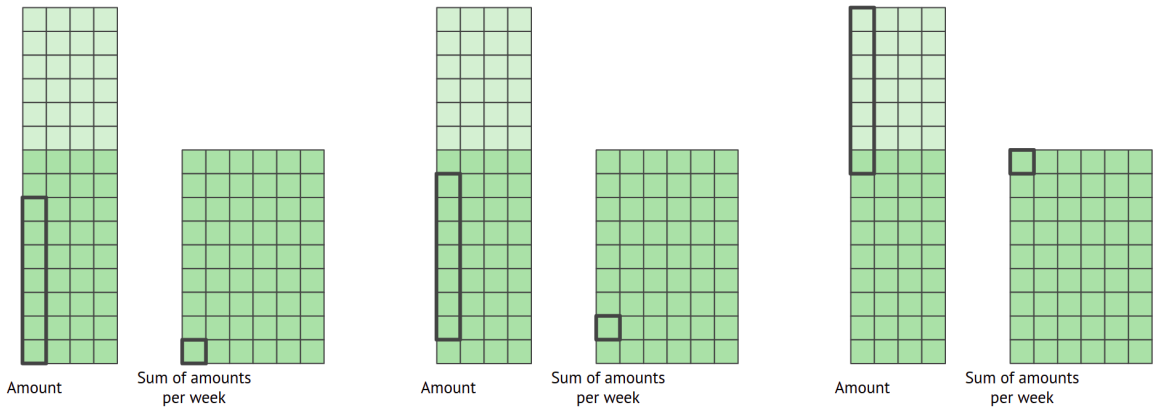


Figure 3.6: Example of using a convolution layer to calculate profiles. We show three stages of the kernel (black rectangle) sliding through the time dimension (vertical, in this plot), in order to extract a profile (black square). The input tensor is the one on the left (with the "Amount" feature), the output tensor is the one on the right (with the "Sum of amounts per week" feature). The lighter cells in the input tensors correspond to the padding required.

The product of the number of windows and aggregations leads to 30 features, 18 of which are used in the rules, and hence in the next layer. Besides these 18 features, we still need 2 other: the number of unique external accounts with which each internal account made incoming/outgoing transactions in the last 7 days. These need to be obtained directly from tensor $\mathbf{T_i}$, using similar operations as we described before but in reverse order: first we use a convolutional filter that slides over the third dimension to make the 7 days aggregations, then another convolutional filter that slides over the second dimension to sum those aggregations.

These are all the profiling features we need to use for the rules. Nevertheless, we add 2 extra features to mark the padding of the biggest time filters, since some of the rule scenarios are based on averages and the padding zeros may skew these values unknowingly. So we add 2 features with 1's in the last six/ten months (signalling padding) and 0's elsewhere, to mark the days that may be affected by padding values.

The output of this set of layers is the tensor $\mathbf{T_p}$ of shape $(M, 22, T)$.

Next, in the prediction block, we use the same learnable layers architecture as the fully learnable network that we described in Section 3.2.1. The output of the proxy network is tensor $\mathbf{T_o}$ with the predictions.

### 3.2.3 Fully Manual Architecture

Lastly, we can implement the rules completely inside a neural network with fixed parameters. This is the most complex solution to implement, since it requires us to manually choose and fix all the weights and biases of all operations inside the network. However, if done correctly, it should yield labels identical to the original rules, without requiring any type of training.

In the profiling block, we use the same steps we described in the semi learnable architecture in Section 3.2.2.

In the prediction block, we combine the profiles in order to mimic the logic behind each rule scenario. Luckily, all conditions from all rule scenarios can be implemented as a linear combination of the profiling features that we have already calculated. So, by choosing carefully the weights and biases of the convolution filters and using a ReLU as the last activation function, we can calculate the triggers of each condition in one layer. With the correct weights and biases, we can ensure that a value of 0 corresponds to the original condition not being met, while positive values correspond to the condition being met.

Then we combine these conditions using logic operations of conjunctions and disjunctions. The disjunction operation is straightforward to implement, we can just add the values corresponding to the conditions. If at least one of the conditions of the proxy rules is met, then the corresponding output is positive.

The conjunction is a bit more complex. One solution is to use the minimum operation, which has the desired behaviour of being positive iff all conditions are positive. This option has the property that the gradient only flows through the entries of the tensor that are responsible for the minimum value. This can be a good property because, if we have a situation where, for example, the amount is substantially above its threshold but the number of transactions is just slightly above its threshold, then the generator can learn to not trigger this rule scenario by creating examples with less transactions but with the same total amount. However, this highly targeted feedback can cause the generator to output transactions with a larger variation of amounts or that trigger

the rules more often. Furthermore, the training process of the generator can become more unstable, since the generator is only learning to avoid the triggers one way at a time, which may lead to a cyclic behaviour of learning and forgetting which patterns trigger the rules.

Another solution to implement the conjunction operation is to use the detach function. This function removes terms from the computational graph when doing backpropagation to calculate gradients. Using this, we can easily have different behaviours during the forward and the backward pass. For example, to implement $x \wedge y$, we write $(min(x, y) - (x + y)).detach() + (x + y)$. In the forward pass, the $(x + y)$ terms will cancel out and we are left with the previous solution. However, in the backward pass, the detached part will be ignored and we are left with an addition that will distribute the gradient fairly.

## 3.3 Money Laundering Objective

The generator should not only try to avoid the rule-based AML system, but also be stimulated to do money laundering which, in our case, means creating money flows that are typical of layering. This is a classical pattern of layering in which mule accounts receive large volumes of money and subsequently send it, in order to distance the money from its illicit origins. Also, we know, from consulting with domain experts, that money launderers do not usually leave much money in the internal account for a long time, because the faster the money gets to its final destination, the less likely it is for it to be apprehended. Furthermore, since we are not considering deposits and withdrawals, the amount of money that an internal account sends should not be much different than what it receives. Because of these two facts, the goal is not only to maximise the amount of money flowing, but also minimise the amount of money blocked (transferred from a source to the internal account and then not leaving) and created (an internal account sending much more money than what it receives).

We assume that a mule's objective is to maximise the money flowing while minimising the money blocked or created, as defined in the previous paragraph. If we define the total amount of incoming money to an internal account as $x$ and the total amount of outgoing money from the same account as $y$, we can then formalise this objective as maximising the function from Eq. 3.3.

$$\Lambda_{\text{flow}}(x, y) = x + y - \beta \left| x - y \right| \tag{3.3}$$

The term $(x + y)$ is responsible for maximising the amount flowing, while the term $|x - y|$ is trying to minimise the amount of money blocked and created. The parameter $\beta$ controls the relative strength of the balancing term.

In Figure 3.7, we show the shape of this function, as well as points A through F exemplifying the learning trajectory that we expect our model to follow: balancing $x$ and $y$ by getting closer to the crease, and climbing it by increasing both $x$ and $y$.

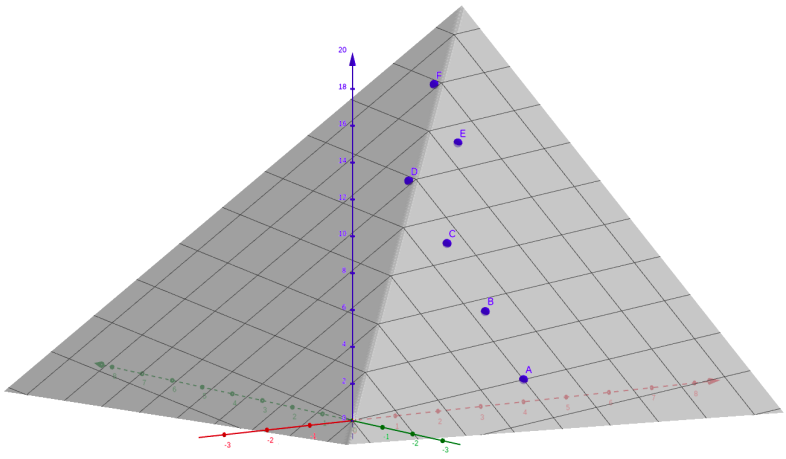


Figure 3.7: Example of expected learning trajectory given the objective function. $x$ (red axis) and $y$ (green axis) have the same definition of Eq. 3.3 and $z$ (blue axis) is the output of $\Lambda_{flow}$.

## 3.4 Generator

The generator is responsible for creating synthetic examples of money flows that evade the rule-based system. It is a deep learning model, trained with feedback from the rules proxy network (see Section 3.2), the money laundering objective (see Section 3.3) and the discriminator (see Section 3.6). Recall that the data format that we are using to represent the transactions (i.e., the format of the output of the generator) is a 3D tensor of shape $(M, S + D, T)$ as described in Section 3.1. The first dimension runs over the set of internal accounts, the second dimension runs over the external accounts, and the third dimension runs over the set of time steps. Each entry of the tensor is either 0 (no transaction) or the value of the amount transferred.

The generator architecture is composed of three main blocks: (1) a block of dense layers, gradually mapping a noise vector to a coarse temporal representation of interactions between accounts, (2) a block of transposed convolutional layers, gradually

refining the temporal representation of interactions between accounts up to a single day, and (3) a block of transformations that is responsible for ensuring the sparsity of the tensor and make the number of transactions independent from the amount. We explain each of these blocks with more detail in Sections 3.4.1-3.4.3.

Finally, we rescale the output by a constant amount in order to speed up training. Since the model starts with small random values for its parameters, it would take several epochs before learning to generate high amount transactions in a stable way. So, the last step of the generator is to multiply its output by a constant amount larger than 1. We performed some preliminary experiments to choose the best value for this rescaling constant, as well as test other strategies (e.g. generating amounts in logarithmic scale) but we found the linear rescaling to work better.

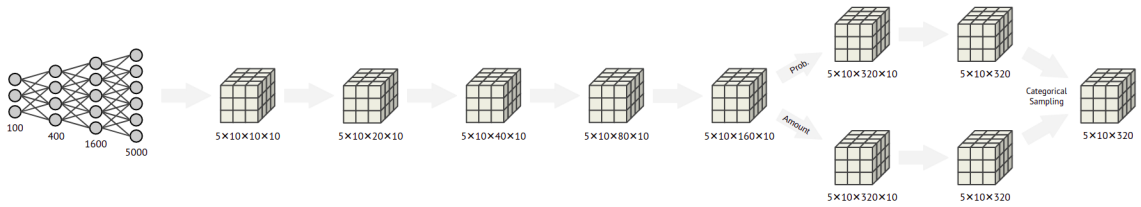


Figure 3.8: Complete generator's architecture.

### 3.4.1 Mapping noise to tensor

We want to be able to create complex patterns of money laundering that involve multiple accounts that act together in an organised way. As such, our generator needs to be able to coordinate transactions between all different pairs of accounts. Because of this, typical GAN architectures for image generation based on convolutions are not a good fit to our use case, since in our use case there is no clear notion of locality (whereas in image generation neighbouring pixels tend to be similar). In fact, in our 3D output tensor, the order of internal accounts (first dimension) and external accounts (second dimension) does not bear, a priori, any meaning. Only the third dimension, which is encoding the timestamp of the transactions, has a clear ordering and could potentially show some seasonal behaviour.

In order to generate money laundering operations with coordinated accounts without imposing local behaviour, the network is designed with a first set of fully connected layers that upscale the original small input noise vector into a larger vector. Then it is reshaped to a $4D$ tensor of shape $(M, S + D, T_0, F)$ (see example in Figure 3.9).

The first two dimensions already encode what we want to output, but the size of the third dimension is significantly smaller than what we want to generate. We can interpret each $(1, 1, 1, F)$ slice of this tensor as a feature vector of length $F$ encoding the interactions between a pair of accounts in a time window spanning several days of the final output. This provides a coarse temporal representation of the behaviour of accounts to be refined in subsequent layers. Since, in this first set, we use fully connected layers, all these entries can be generated either independently or with any structure of dependencies – to be learned by the model in the optimization process.

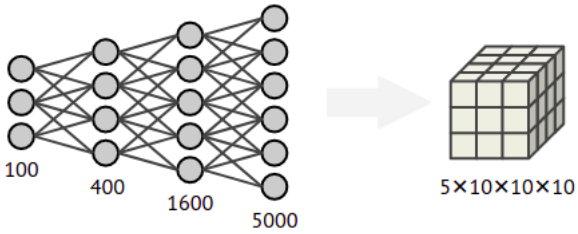


Figure 3.9: First stage of the generator: map noise to tensor with coarse temporal representation.

### 3.4.2 Increasing time granularity

The second block of the network is responsible for gradually increasing the time granularity of the 4D tensor created in the first block. This is very similar to the process of image generation using GANs: in shallower layers we have a coarse representation of the image, while in deeper layers it is represented with much more detail (higher image resolution). The difference is that, instead of pixel definition, here we are increasing temporal definition (see example in Figure 3.10). To do this, we use transposed convolutional layers with filters that slide across the time dimension and encompass all features of the previous layers.

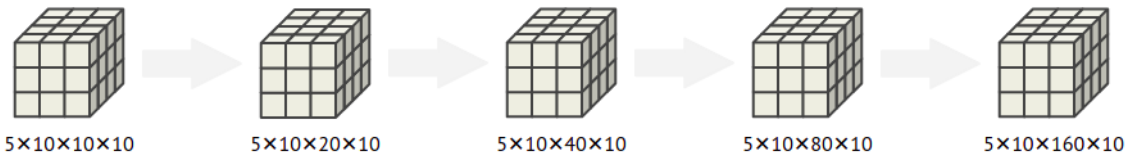


Figure 3.10: Second stage of the generator: increase time granularity.

### 3.4.3 Sparsifying the transactions

In transaction networks, each person only sends/receives money to/from a very limited number of people compared to the total number of people in the network. As such, if we use our 3D tensor representation for the transactions, the tensor should be very sparse, with the vast majority of entries being 0.

In order to enforce this sparsity, we randomly sample some of the entries of the tensor and only those will contain a transaction. This is achieved by branching the output tensor from the previous stage into two tensors: one to contain the amounts of each potential transaction and another one to contain the probability of the corresponding transactions to occur (see example in Figure 3.11). Each branch has its own transposed convolutional layers in order to allow some independence between probability and amount of each transaction. Also the last layer maps the 4D tensors into 3D tensors with the shape that we should output, one with the amount information and the other with the probability. The last activation function of the amount tensor is the $Softplus(x) = \log(e^x + 1)$, which is very similar to the ReLU that we have been using but always strictly bigger than 0, a desired property for the entries of the amount tensor. The last activation function of the probability tensor is a Sigmoid, to ensure that the result is a value between 0 and 1.

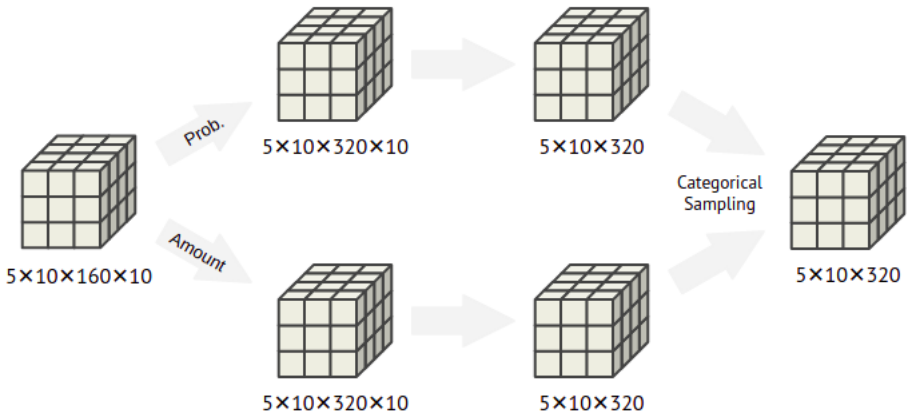


Figure 3.11: Third stage of the generator: make the tensor sparse.

The categorical sampling step is done via Bernoulli sampling on the probability tensor. Each entry becomes 1 with probability given by the value in the entry of the probability tensor, and 0 otherwise. The result is the adjacency tensor of the accounts. Then, to select the corresponding amounts to obtain the final output of the generator, we multiply element-wise the adjacency tensor by the amount tensor (see example in Figure 3.12).

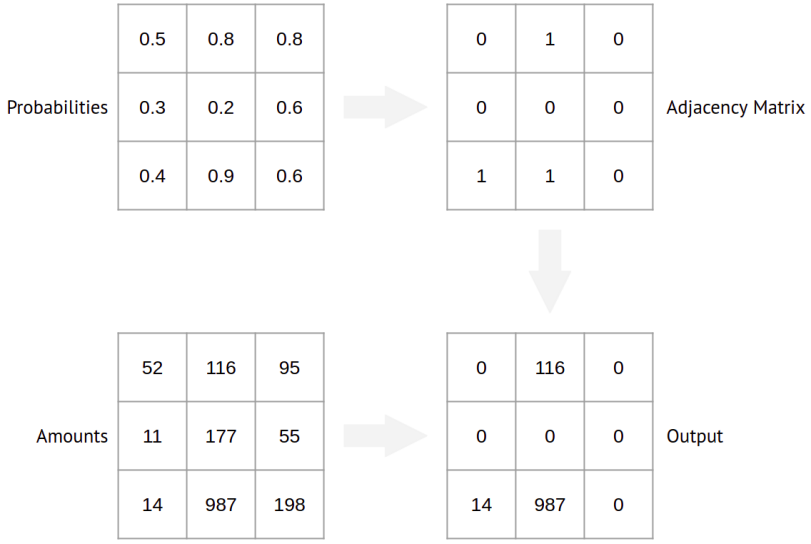


Figure 3.12: Example of categorical sampling operation, showing how to produce the output tensor from the probabilities and amounts tensors. Here, we only show 2D matrices instead of 3D tensors for depiction simplicity.

This approach not only solves the problem of making the output tensor very sparse, but also introduces some randomness (ergo some variability) to the generated data and allows us to sample transactions with probability somewhat independent from the amount (e.g., transactions with low amount and high probability or vice-versa).

### 3.4.4 Loss

The generator should mimic a money launderer, with the goal of making as much money flow through the bank as possible without triggering the AML system. We use the money laundering objective to measure the quality of the money flows of the synthetic examples. We use the proxy network in place of the rule-based AML system to detect and learn to avoid the triggers. So, in other words, the generator should be trained to maximise the money laundering objective of Eq. 3.3 while minimising the predicted label of the proxy network. These two loss functions are adversarial in nature, in the sense that the money laundering objective will push the generator towards the opposite behaviour compared to the proxy network loss. We use a hyperparameter $\alpha$ to balance the strength of these two loss functions. As such, the generator will be trained to minimise the loss function from Eq. 3.4.

$$\Lambda = (1 - \alpha)\Lambda_{\text{proxy}} - \alpha\Lambda_{\text{flow}} \tag{3.4}$$

$\Lambda_{\text{flow}}$ stands for the loss from the money laundering objective function, which is the output of the function from Eq.3.3. $\Lambda_{\text{proxy}}$ stands for the loss from the proxy network, which is dependent on the version of the proxy network. If we use the fully learnable or semi learnable versions, where the last activation function is a Sigmoid, then the loss is the binary cross entropy $\text{BCE}(x, y) = -(y \log(x) + (1 - y) \log(1 - x))$, where x is the predicted value and y is the target label. However, since the generator should always avoid triggering the rules, the target label is always 0, so we can simplify the proxy loss to be $\Lambda_{\text{proxy}}(\text{G}(z)) = -\log(1 - \text{proxy}(\text{G}(z)))$. If we use the fully manual version, where the last activation function is a ReLU, then the loss is directly the output of the proxy network.

When we include the discriminator in the AML system, we need to add a new term into the loss function, corresponding to the loss of the discriminator. The hyperparameter $\alpha$ will balance the strength of the money laundering objective relative to the other two losses, while a new hyperparameter $\gamma$ will regulate the strength between the proxy network and the discriminator. As such, the generator is trained to minimise the loss function from Eq. 3.5.

$$\Lambda = (1 - \alpha)(\gamma \Lambda_{\text{proxy}} + (1 - \gamma)\Lambda_{\text{disc}}) - \alpha \Lambda_{\text{flow}} \tag{3.5}$$

$\Lambda_{\text{flow}}$ and $\Lambda_{\text{proxy}}$ have the same meaning as in Eq. 3.4. $\Lambda_{\text{disc}}$ stands for the loss from the discriminator, which also depends on the GAN architecture that we use. If we use the original GAN loss, $\Lambda_{\text{disc}}(\text{G}(z)) = -\log(1 - \text{D}(\text{G}(z)))$. If we use the WGAN loss, $\Lambda_{\text{disc}}(\text{G}(z)) = \text{D}(\text{G}(z))$.

## 3.5 Sampling Strategy

The discriminator consists of a deep learning model that classifies transactions as real (i.e., from a real dataset) or synthetic (i.e., from the generator). During training, it is fed both types of instances and, as such, we need to sample transactions from our real dataset.

The samples should remain true to the real distribution of transactions and be small enough to fit in our data representation. However, we do not want to make it too easy for the discriminator to distinguish between real and synthetic data, so sampling a small number of low amount transactions is not ideal, since this is the opposite of the expected behaviour for the generated data.

Another property that we want to guarantee is that all accounts in each of our samples belong to the same connected component (set of nodes for which always exists a path between any pair of nodes). We want to ensure this property because, in reality, when money laundering analysts are investigating an alert, they start from an account or transaction between a pair of accounts and propagate their search space to neighbouring accounts and their transactions. The result is a money laundering case of connected entities, which in our simplified scenario (where each account is an entity and their relations are transactions) means that all samples should be connected subgraphs of the original data.

We will now describe three different methods for sampling from the real dataset that we tried.

The first method is a simple random walk. To build each sample, we start in a seed node selected at random and repeatedly pick a neighbour from our current node to jump to, adding it to the sample. The only restriction is that the number of source, internal and destination accounts cannot become larger than their maximum size allowed by our data representation. So, in each step of the random walk, when selecting the next node to jump to among the neighbours of our current node, the candidate nodes are the ones that: we have not visited before (do not belong to our sample) but belong to a set (source, internal, destination) that we have not filled yet; have already been visited (belong to our sample), which allows to backtrack from dead ends. For example, if a neighbour of the current node is a source account that does not belong to our sample yet, but we have already reached the maximum number of source accounts that fit in our data representation, it will not be considered as a candidate for which to jump next. We stop sampling nodes when (1) we reach a predetermined maximum number of jumps or (2) our sample has the maximum number of source, internal and destination accounts that fit in our data representation.

The second method is to follow the order of a breadth-first search. To build each sample, we start in a seed node selected at random and fill a queue with its neighbours. Then, we pop the front of the queue to select the next node to add to our sample and insert its unseen neighbours on the queue. Here, the same restriction as before applies: we do not want more than the maximum number of source, internal and destination accounts, so every time we pop or are about to insert a node in the queue belonging to an already filled set, we ignore it. We stop sampling nodes when (1) the queue is empty or (2) our sample has the maximum number of source, internal and destination accounts that fit in our data representation.

A third option is to do community detection on the graph of transactions and extract communities that fit inside our data representation. The algorithm that we use is the Girvan–Newman method from the networkx package[1]. This method detects communities by progressively removing edges from the original graph, breaking it down into smaller connected components. We stop when the component containing the seed node becomes small enough to fit inside our representation of the data, while still being a tripartite graph (at least one node in each of the three sets). This component becomes our sample.

## 3.6 Discriminator

The sampling strategy and the generator architecture discussed in the previous sections provide the required input for the discriminator, which we now describe. The goal of this deep learning model component is to complement the traditional rule-based AML system. Since the generator is trained to create transactions that make as much money flow through the bank as possible without triggering the rules, we use its outputs as approximations of positive instances of money laundering. Using the sampling strategy that we discussed in the previous section, we can sample instances that fit within our data representation and use them as negative instances of money laundering. So, the discriminator is trained to solve a classification problem: given a set of transactions represented as we previously described, which ones are synthetic (positives) and which ones are real (negatives)?

The discriminator architecture is a slightly modified mirror image of the generator, still composed by three main blocks: (1) a block of convolutional layers, gradually reducing the temporal granularity of the interactions between accounts, (2) an aggregation step that will make the prediction of the discriminator permutation invariant in relation to the order of the accounts, and (3) a block of dense layers, mapping the features extracted by the previous stages to a single output, the class prediction.

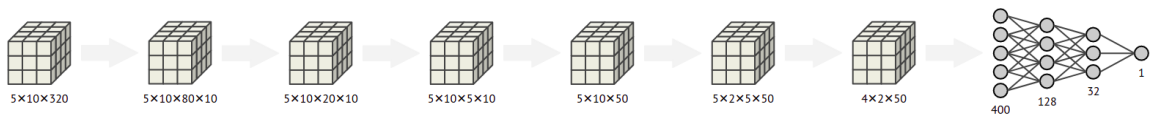


Figure 3.13: Complete discriminator's architecture.

[1]https://networkx.org/documentation/stable/release/release_2.5.html

### 3.6.1 Decreasing time granularity

We need to go from an input tensor of shape $(M, S + D, T)$ to a single output value, so we need to decrease the size of our data representation. We decrease the size of the time dimension first because we can do this using convolutional layers with filters that slide across the third dimension of the tensor, where each kernel is capturing a specific pattern that is invariant over time and over accounts. Furthermore, convolutional layers are more computationally efficient than dense layers, so we use them in the first layers of the network, when the data representation is too big for dense layers.

In order to increase expressivity, we add a fourth dimension of fixed size to these intermediate tensors, similar to what we described in the generator's architecture, which then collapses at the end of this block (see example in Figure 3.14). The result is a 3D tensor where the first dimension runs over internal accounts and the second dimension runs over external accounts. A $(1, 1, X)$ slice of the tensor can be seen as a feature vector that is encoding the behaviour of transactions between the corresponding pair of accounts.

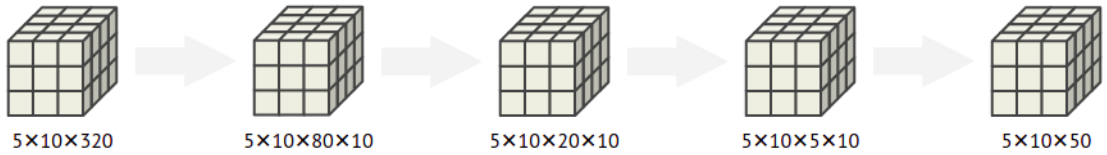


Figure 3.14: First stage of the discriminator: decrease time granularity.

### 3.6.2 Enforcing permutation invariance

One necessary property for our discriminator is that it must be permutation invariant with respect to the order of the accounts. This means that the discriminator should output the same answer even if we change the order of the internal accounts or the order of external accounts in our representation of the transactions. Only the topology of the graph and the amounts transacted at the edges is relevant. Thus the ordering of the internal accounts in the tensor is arbitrary. The same can be said of the external accounts in the second dimension, with the remark that we can only swap source accounts with other source accounts (first half of this dimension) and destination accounts with other destination accounts (second half of this dimension), since they represent different types of counterparties of the internal accounts. In contrast, the order of the entries in the time dimension of the original input tensor is very important, since the order of the entries maps directly to the passage of time and seasonality or local behaviours may exist.

Now that we have one feature vector representing the transactions between each pair of accounts, we split the second dimension into two parts: one regarding the transactions from source accounts, and one regarding the transactions to destination accounts (see example in Figure 3.16). Thus we obtain a 4D tensor where the dimensions that we want to be permutation invariant are the first (internal) and the third (external). We can then apply any number of commutative functions to aggregate these dimensions into features, for example the maximum, minimum, mean, standard deviation, count etc (see example in Figure 3.15).

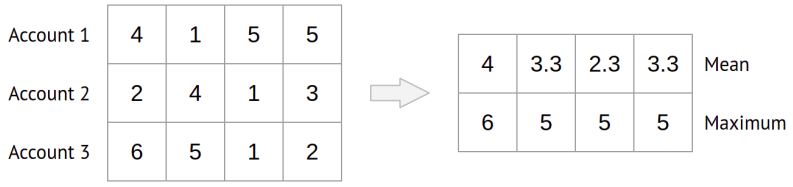


Figure 3.15: Example of permutation invariant aggregations. On the left side, as input, we have a matrix with 3 internal accounts, each with 4 features. We then apply the mean and maximum aggregations (which are commutative) over the internal accounts (columns) and get the matrix on the right as output. Notice that changing the order of the internal accounts in the input matrix would not change the output matrix.

The result is a 3D tensor where the first dimension relates to the aggregations that we used. For example, using the mean and maximum as aggregation functions would lead to a first dimension of size 2, where the first index would contain the mean of value over all accounts and the second index would contain the maximum. The second dimension of size 2 corresponds to the type of transaction (incoming or outgoing, from the point of view of the internal accounts). And in the third dimension we have the features that were extracted from the tensor. We reshape this 3D tensor into a 1D vector of features, which is passed on to the third and last block.

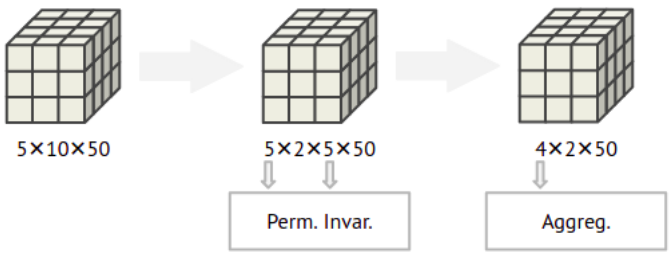


Figure 3.16: Second stage of the discriminator: enforce account permutation invariance.

### 3.6.3 Mapping tensor to prediction

From the 1D vector of features from the previous block, we use dense layers to make the final prediction of the discriminator. If the goal is to make a binary classification of the input tensor, using the cross entropy loss, the last activation function is a sigmoid, meaning that the output of the discriminator will be a value between 0 and 1, ideally close to 0 if the input was real/legitimate or close to 1 if the input was synthetic/money flow pattern. If instead we use the WGAN loss, we do not have an activation function after the last layer. In this case, lower output values correspond to inputs perceived as real/legitimate and higher output values correspond to inputs perceived as synthetic/money flow pattern.

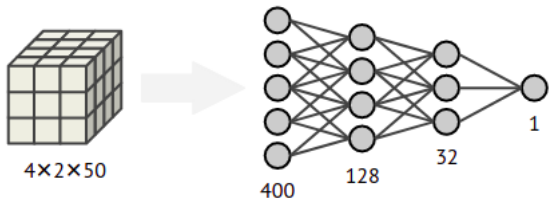


Figure 3.17: Third stage of the discriminator: map feature tensor to final prediction.

# Results 4

In this chapter we describe the setup and results of the experiments done to validate our architecture and its components.

We start by describing the dataset we use to perform the experiments in Section 4.1. Then, we discuss experiments on various implementations of smaller components of the system, namely the rules proxy architecture in Section 4.2 and the sampling strategy in Section 4.3.

Then we have the experiments of the complete architecture. In Section 4.4, we explore a training strategy where the generator and the discriminator are trained separately. In Section 4.5, we use the classical GAN training strategy of learning the generator and discriminator together.

In all of the experiments, we use the data representation of a 3D tensor with shape $(5, 10, 303)$. This represents money laundering examples with at most 5 internal accounts, 5 source accounts, 5 destination accounts and a timespan of 303 days. We chose 303 days since that is the timespan of the real dataset we use in our experiments.

## 4.1 Dataset

We use an internal banking transaction dataset from Feedzai (henceforth referred to as Banking dataset) for our experiments. From this dataset, we gather only the transactions relevant for our project, namely the ones made from an external account to an internal account or vice-versa. The resulting dataset consists of 200,000 transactions made between 100,000 unique accounts (approximate numbers), spanning over 10 months.

Given the number of accounts and transactions and the time span of the dataset, the 3D tensors sampled from it are very sparse, meaning that most entries have a value of 0, representing that no transaction took place.

## 4.2 Rule Proxy Network

In this section we compare the three different rule proxy network architectures described in Section 3.2: the fully learnable, the semi-learnable and the fully manual networks. All three architectures have the goal of accurately reproducing the original rules system. Thus, the three architectures should produce the same labels for the same inputs, but they calculate the labels differently. In order to perform such comparison, we build a labelled dataset of tensors representing transactions, train the proxy networks with those tensors as input, and measure how frequently their output labels match the labels produced by the original rules.

This labelled dataset is built using transactions from the Banking dataset. We split the data into train, validation and test sets with 20000, 5000, 5000 internal accounts, respectively. Each internal account is represented as a matrix of size $(50, 303)$, corresponding to the second and third dimension of the data representation described in Section 3.1. Then, for each internal account, we calculate the rules labels in the shape of a $(7, 303)$ matrix. Here we are using the fact that the rules (and consequently the rule proxy network) evaluate each internal account independently. As such, we can sample internal accounts and their corresponding labels at random to build a 3D tensor and the proxy network uses the first dimension of this tensor as the batch dimension.

We use this labelled dataset to train the fully learnable and semi-learnable versions of the proxy network. The fully manual version does not need training since all of its parameters (i.e., weights and biases) are manually set and fixed. We perform a grid search over various values of several hyperparameters, listed in Table 4.1. The performance metric that we use is the area under the ROC curve with a maximum false positive rate of 0.01%. We used such a small value for the maximum false positive value because the number of negatives is several orders of magnitude larger than the number of positives (since we are counting positives and negatives at the level of each entry of $\mathbf{T_o}$). After selecting the best fully learnable and semi-learnable models, we use the validation set to obtain the optimal thresholds for distinguishing positives and negatives for each network for each rule scenario. Finally, we measure the performance of the 3 versions of the proxy network on the test set.

In Table 4.2 we show the number of true positives (trigger the proxy and the rules), false positives (triggers the proxy but not the rules), false negatives (trigger the rules but not the proxy) and true negatives (triggers neither the proxy nor the rules) of the predictions of the fully learnable proxy network, for each of the rule scenarios. We can see that there are rule scenarios (e.g. 2 and 5) for which the network performed

| | |
|---|---|
| learning rate starting value | 0.01, 0.001 |
| learning rate patience | 10, 20 |
| learning rate decay factor | 0.5, 0.1 |
| weight decay | 0.001, 0.0001, 0.00001 |
| number of profiles learned (P) | 60, 120, 180 |
| number of prediction layers | 4, 6, 8 |

Table 4.1: Values of hyperparameters of the proxy network grid search. We implement a learning rate decay strategy in which we start the training with a certain starting value and if the performance of the network does not improve after a certain number of epochs (patience) we multiply the learning rate by the decay factor. We use weight decay as a regularisation technique for the network optimizer. $P$ is the number of profiles extracted in the profiling block (only applicable in the fully learnable version). Lastly, we try different numbers of layers in the prediction block, gradually decreasing the size of the second dimension from $P$ to $R$.

significantly better than others (e.g. 1, 4, 6). The number of true negatives is several orders of magnitude larger than the rest because the 3D tensor is very sparse (as discussed in Section 4.1). These results of the fully learnable architecture correspond to an overall precision of 50% and recall of 47%.

| | Rule 1 | Rule 2 | Rule 3 | Rule 4 | Rule 5 | Rule 6 | Rule 7 | Total |
|---|---|---|---|---|---|---|---|---|
| TP | 16 | 86 | 57 | 1 | 660 | 13 | 61 | 894 |
| FP | 178 | 19 | 69 | 69 | 122 | 89 | 361 | 907 |
| FN | 423 | 30 | 63 | 67 | 111 | 146 | 181 | 1021 |
| TN | 1.5e6 | 1.5e6 | 1.5e6 | 1.5e6 | 1.5e6 | 1.5e6 | 1.5e6 | 1.1e7 |

Table 4.2: Results from fully learnable proxy network.

In Table 4.3 we show the number of true positives, false positives, false negatives and true negatives of the predictions of the semi-learnable proxy network, for each of the rule scenarios. We can see a significant increase in the overall performance of the semi-learnable proxy network, compared to the fully learnable version. This is expected since the profiling block of the semi learnable architecture is already optimised before training, meaning that the network only needs to learn the parameters for the prediction block. These results correspond to an overall precision of 70% and recall of 90%.

In Table 4.4 we show the number of true positives, false positives, false negatives and true negatives of the predictions of the fully manual proxy network, for each of the rule scenarios. We can see that this network has near perfect performance, only making 2 mistakes in the entire test set. They happened because we used the

| | Rule 1 | Rule 2 | Rule 3 | Rule 4 | Rule 5 | Rule 6 | Rule 7 | Total |
|---|---|---|---|---|---|---|---|---|
| TP | 349 | 114 | 100 | 58 | 759 | 112 | 232 | 1724 |
| FP | 315 | 44 | 42 | 57 | 103 | 102 | 73 | 736 |
| FN | 90 | 2 | 20 | 10 | 12 | 47 | 10 | 191 |
| TN | 1.5e6 | 1.5e6 | 1.5e6 | 1.5e6 | 1.5e6 | 1.5e6 | 1.5e6 | 1.1e7 |

Table 4.3: Results from semi learnable proxy network.

smooth function from Eq. 3.2 in order to detect the round amount transactions and the resulting approximate profile was slightly below the rule threshold. Nevertheless, this version is the most accurate proxy for the rules and it does not require any training (even though the implementation is the most complex of the three), so we use it in the remaining experiments.

| | Rule 1 | Rule 2 | Rule 3 | Rule 4 | Rule 5 | Rule 6 | Rule 7 | Total |
|---|---|---|---|---|---|---|---|---|
| TP | 437 | 83 | 51 | 69 | 721 | 225 | 212 | 1798 |
| FP | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| FN | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 2 |
| TN | 1.5e6 | 1.5e6 | 1.5e6 | 1.5e6 | 1.5e6 | 1.5e6 | 1.5e6 | 1.1e7 |

Table 4.4: Results from fully manual proxy network.

## 4.3 Sampling Strategy

In this section we compare the three different sampling strategies described in Section 3.5: random walk, breadth-first search and the community detection. Our goal is to understand how the sampling strategy affects some of the relevant metrics in the money laundering domain, namely the total amount of money flowing through the internal accounts, the amount per transaction and the number of transactions per sample. So, in our experiments we use each of the methods to sample 10,000 tensors with shape $(5, 10, 303)$ from the Banking dataset and measure these statistics. The results are depicted in Figure 4.1. As we can see, the results from the three methods are similar, even though the community detection method has slightly less transactions per sample and consequently less total amount of money flowing.

However, we should note that these results are specific to this dataset and may be a consequence of the dataset being very sparse, which led to the majority of the graphs of interactions between accounts in the samples being trees (connected components with $N$ nodes and $N - 1$ edges), as we can see in Figure 4.2.

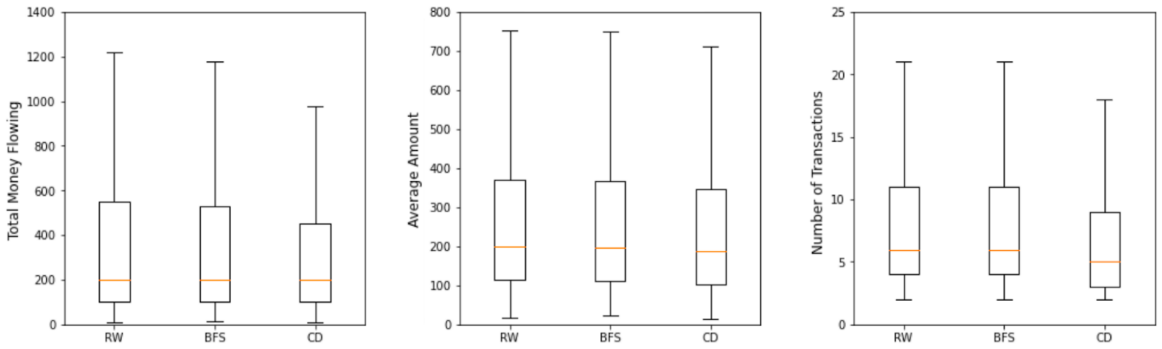


Figure 4.1: Boxplots of statistics measured on samples from different sampling strategies. We are not showing outliers in this plot.

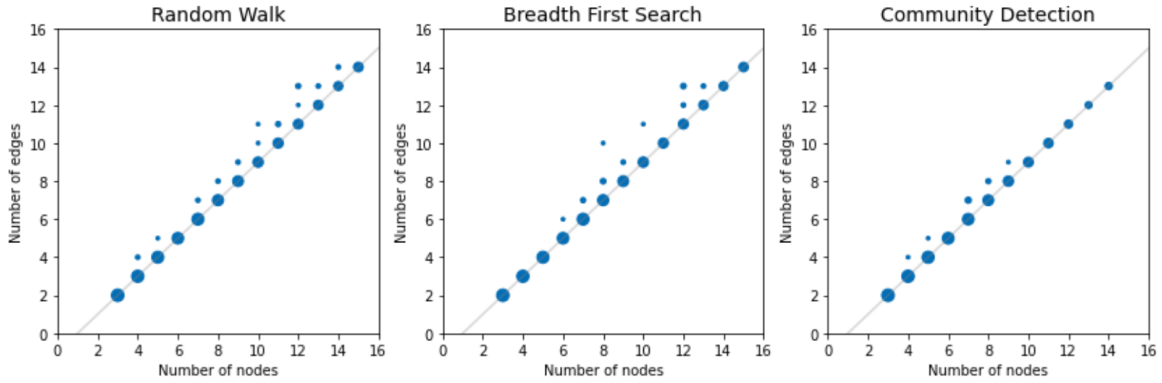


Figure 4.2: Scatter plot of the number of edges per number of nodes in samples from each of the methods. The nodes represent accounts and there is an edge between two nodes if there is at least one transaction between the corresponding accounts. The size of the points represent their frequency and points on the grey line correspond to trees.

We use the breadth-first search method to collect real samples for the next experiments, since it is the fastest to run and results are similar among the three methods. In Figure 4.3 we show the normalised frequency of amount per transaction and the number of transactions per sample in the samples extracted from the Banking dataset using this method.

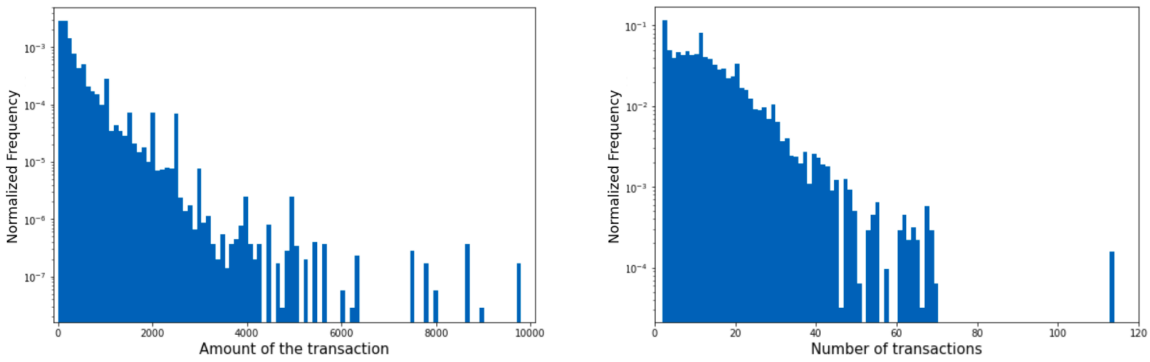


Figure 4.3: Normalised frequency of amount per transaction and number of transactions per sample of the real data.

## 4.4 Iterative generator and discriminator training

In the previous sections, we cover the experiments of the sampling strategy and the rule proxy network, testing each of those components separately. In this section, we test our complete architecture, with all of its components working together (see Figure 3.1). However, training GANs is known to be an unstable process, since the target of both models (each other) is constantly changing during training.

So, in order to avoid this problem, in this section the generator and the discriminator are trained separately. First, in Section 4.4.1, we train a simple generator that only receives feedback from the proxy network and the money laundering objective, which learns to create money flows that do not trigger the rules. Then, in Section 4.4.2, we fix this generator and train a discriminator to distinguish between real data and the outputs of the generator. After that, in Section 4.4.3, we fix this discriminator and continue the training process of the generator, but this time it also receives feedback from the discriminator, which it learns to fool as well as the rules. Lastly, in Section 4.4.4, we fix the generator again and continue the training process of the discriminator, using generated examples from the improved generator.

### 4.4.1 First Generator

In this section we describe the results of the generator with the architecture described in Section 3.4. This generator is trained with feedback from the proxy network and the money laundering objective, using the loss in Eq. 3.4. As already mentioned, we use the fully manual version of the proxy network for all experiments. We use two versions of this network that only differ in the way they implement the conjunction

of conditions in the prediction block. The first (which we call proxy_min) uses the minimum operation, and the second (which we call proxy_detach) uses the detach function (see subsection 3.2.3). We use the RMSprop optimizer [TH+12] to update the parameters of the generator during training. Some hyperparameters are fixed in these experiments: the batch size is set to 16, $\beta$ coefficient from Eq. 3.3 is set to 0.9, and the scaling factor that we apply to the result of the generator is set to 100. We tested different values for the batch size which returned similar results, so we chose the maximum size that fitted in memory. We also tested different scaling factors, with values in the order of 100 being the ones with a more stable training process. $\beta$ is fixed at 0.9 because, theoretically, the ideal value should be slightly less than 1. This is because it results in the derivative of $x$ and $y$ in Eq. 3.3 being always positive (increase the money flowing) but the derivative of the smaller value of them is much larger than the other (balancing in and out transactions).

We perform a grid search over various values of several hyperparameters, listed in Table 4.5. We perform 3 runs for each combination of hyperparameters, letting the generator train for 250 epochs[1], which in most cases was more than sufficient for convergence. To evaluate the generator performance, we measure the total amount of money flowing through the internal accounts without triggering the rules (no positive labels attributed to the output tensor of the generator) in more than half of the generated examples. Given the stochastic behaviour of the generator, we average the performance of 32 batches, i.e., 512 generated examples. In each run, we save a copy of the generator on the epoch with best performance that respected the restriction of not triggering the rules in more than half of the examples. The runs that never fulfil this restriction are discarded.

| | |
|---|---|
| alpha | $5e^{-8}$, $1e^{-7}$, $2e^{-7}$ |
| learning rate | $5e^{-5}$, $1e^{-4}$, $2e^{-4}$ |
| proxy network | *min*, *detach* |

Table 4.5: Values of hyperparameters of the first generator grid search. $\alpha$ is the coefficient from Eq. 3.4. The learning rate value is kept fixed throughout the training of the generator. The two versions of the proxy network correspond to the two solutions to implement the conjunction operation described in Section 3.2.3.

Figure 4.4 shows a summary of the results of this grid search. As expected, we see a positive correlation between higher values of $\alpha$ (the weight of $\Lambda_{flow}$) and the total amount of money flowing. Looking at the other two statistics, we see that this correlation is much stronger in the average amount plot than in the transaction frequency.

[1]In our experiments, an epoch comprises 128 batches.

This is an indication that increasing the value of $\alpha$ leads to more money flowing by increasing the amount present in each transaction, rather than increasing the number of transactions made. Regarding the proxy network architecture version, we see that using proxy_min leads to higher total money flowing than using proxy_detach, which could be expected as well for the reasons discussed in Section 3.2.3. Looking at the other two statistics, we see that the reason for this is the opposite of $\alpha$: there is no significant increase of the average amount being transferred but the number of transactions is substantially larger. Finally, we can see that changing the learning rate has little effect on any of the 3 statistics that we evaluate.

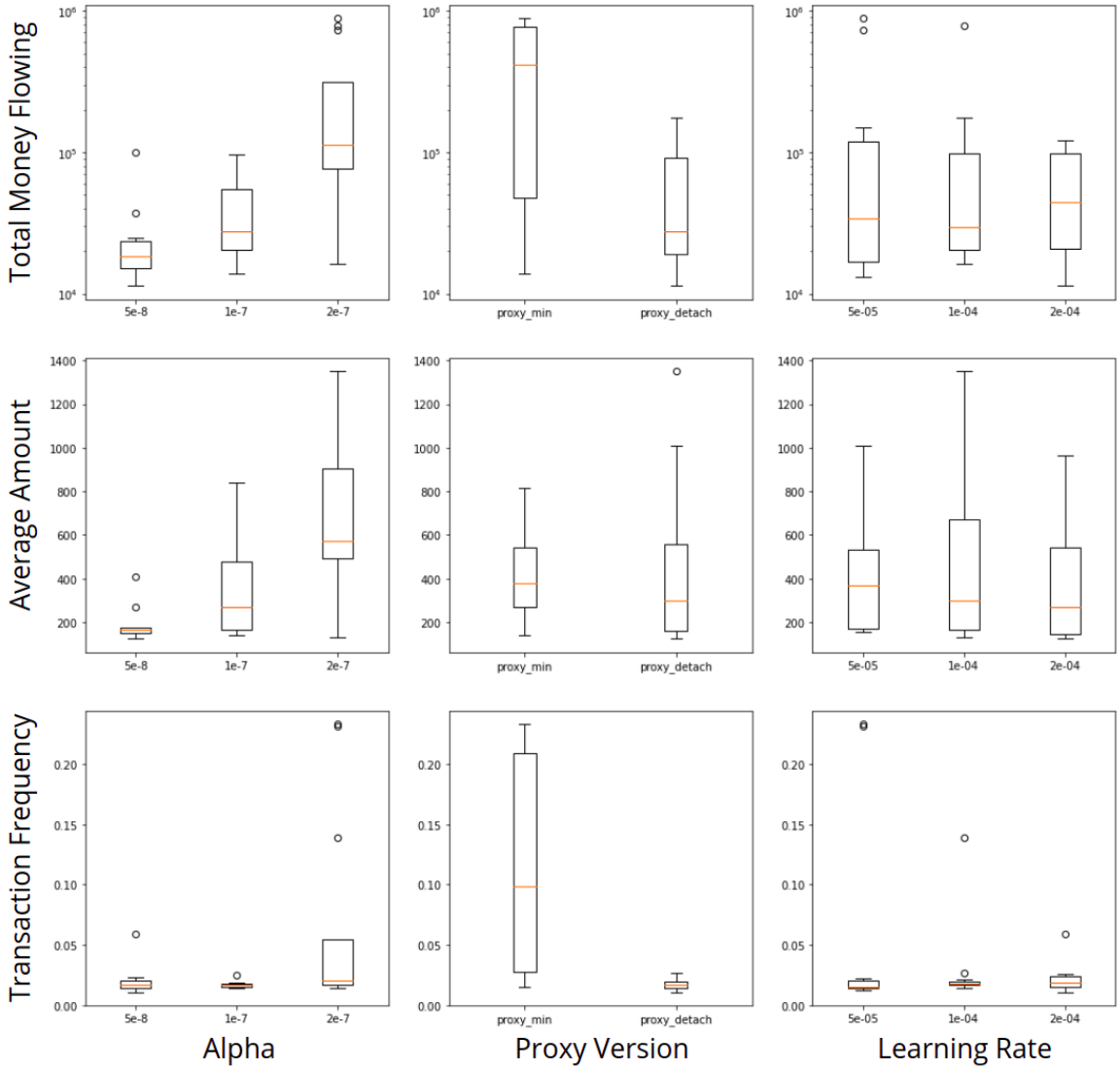


Figure 4.4: Results from grid search of first generator's hyperparameters.

We choose the best performing model from our hyperparameter search to be our first generator. As previously mentioned, we measure performance as the amount of money flowing per generated example (more is better), with the restriction that more than half of the generated examples do not trigger the rules. Our first generator was able to launder approximately 885000 dollars on average, while triggering the rule-based system at least once in 36% of the generated examples.

In Figure 4.5 we show the empirical distribution of amount per transaction and the number of transactions in generated examples from our first generator. We can see that, compared to the distribution of real data samples (see Figure 4.3), the number of transactions is orders of magnitude larger. Also, the generated transaction amounts have a smaller range of values but larger amounts are more frequent, especially between the 2,000 and 4,000 dollar range. These amounts are high compared to the modes of both distributions (real and synthetic), but not high enough to trigger the rules (which start to trigger closer to 10000).

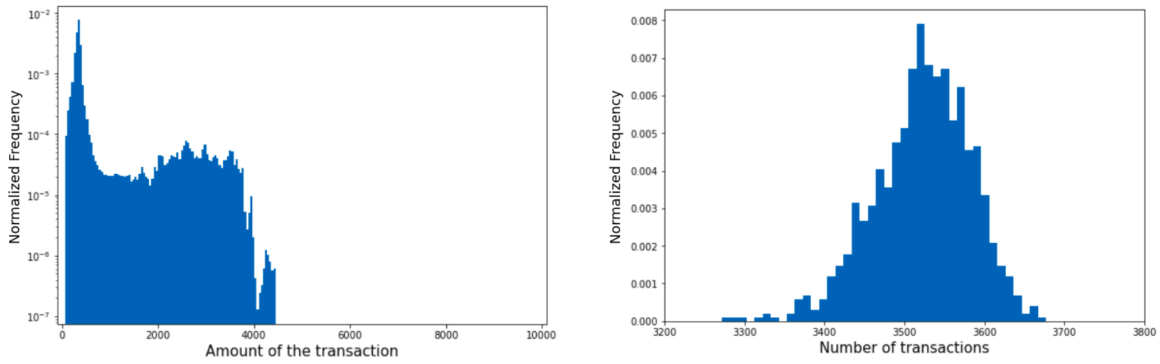


Figure 4.5: Normalised frequency of amounts and number of transactions per synthetic example of the first generator.

### 4.4.2 First Discriminator

We use the sampling strategy from Section 4.3 and the generator from Section 4.4.1 to build a dataset to train the first discriminator. We prepared a train-validation-test split with sizes 40000, 8000, 8000 respectively, with equal number of real and synthetic examples. Then we train the discriminator in a supervised manner with the goal of classifying real examples as 0 and synthetic examples as 1. For the discriminator, we adopted the example architecture described in Section 3.6 (see Figure 3.13). We use the RMSprop optimizer to update the parameters of the discriminator during training.

The only hyperparameter that we tune is the learning rate. We perform 3 runs for each learning rate value, listed in Table 4.6, and measure the area under the ROC to compare the performance. However, the model learns to completely separate real from synthetic examples in very few epochs[2], meaning that area under the ROC is 1.

| | |
|---|---|
| learning rate | $2e^{-5}$, $5e^{-5}$, $1e^{-4}$, $2e^{-4}$ |

Table 4.6: Values of learning rate of the first discriminator experiment. The learning rate value is kept fixed throughout the training of the discriminator.

In order to visualise the training progress of the discriminator, we plot the difference between the minimum score that the discriminator attributed to a synthetic example from the validation set and the maximum score that the discriminator attributed to a real example from the validation set. If this metric is positive, then we have a complete separation between the scores of the two classes. A perfect classifier would predict exactly 0 for real examples and 1 for synthetic examples, meaning that this metric would be 1.

In Figure 4.6 we show the value of this metric across the epochs of the training, for various values of learning rate. We can see that with higher values of learning rate, the discriminator approaches the perfect model more quickly. Nevertheless, every model learns a complete separation between the scores of the two classes.

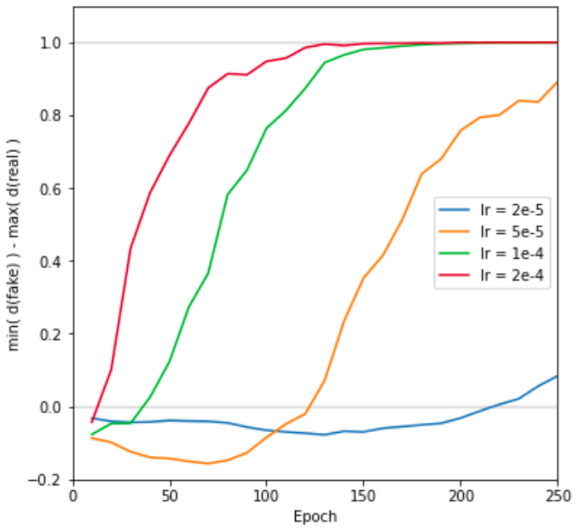


Figure 4.6: Results from training the first discriminator with various values of learning rate.

[2]An epoch was defined as 16 batches.

We chose one of the models trained with a learning rate of $2e^{-4}$ to be our first discriminator. However, instead of using the extremely overfitted model that trained for 250 epochs, we use a snapshot of the model after training for 100 epochs, in order to avoid saturating the gradient that the discriminator gives to the generator. Its minimum score for a synthetic example was approximately 0.99, while the maximum score for a real example was approximately 0.08.

If a model like this was used in production, it would be necessary to choose a hard threshold where instances that fall below it are real, and above it are synthetic. The threshold for this discriminator could be any value between 0.08 and 0.99 without affecting the validation performance. We assume the worst case scenario for the generator and set it as 0.08.

### 4.4.3 Improved Generator

In order to mimic the adaptive behaviour of the money launderers to a new AML solution, we improve upon the first generator by taking into account the added goal of avoiding triggering the discriminator. That is, our goal here is to train a generator that is capable of fooling the first discriminator, as well as the rules system. During this experiment, the parameters of the discriminator were fixed, i.e. the discriminator is not trained to adapt to the generator's behaviour.

Using the parameters of our first generator as initial parameters for the neural network, we proceed with the training as before. We fix most of the hyperparameters to be the same as the ones we use to train our first generator in Section 4.4.1, namely: batch size is set to 16, $\beta$ is set to 0.9, the scaling factor is set to 100, the learning rate is set to $5 \times 10^{-5}$, and we used the proxy_min version of the proxy network. Since the learning rate had little effect in the experiment from Section 4.4.1, we fixed this hyperparameter as well, with the value proposed in the original WGAN paper [ACB17].

We perform a grid search over various values of several hyperparameters, listed in Table 4.7 and measure how these hyperparameters affect the total amount of money flowing through the internal accounts in the generated samples, as well as the number of rule triggers and the mean score attributed by the discriminator. We perform 2 runs for each combination of hyperparameters, letting the generator train for 250 epochs[3]. To evaluate the generator performance, we measure how much money it was able to launder without triggering the rules in more than half of the generated examples and having a mean discriminator score below 0.08 (threshold set in the previous section).

[3]An epoch was defined as 16 batches.

| | |
|---|---|
| $\alpha$ | $2e^{-8}$, $5e^{-8}$, $1e^{-7}$ |
| $\gamma$ | 0.05, 0.1, 0.2, 0.5, 0.8, 0.9 |

Table 4.7: Values of hyperparameters of the improved generator grid search. $\alpha$ and $\gamma$ are the coefficients from Eq 3.5.

The results are depicted in Figure 4.7. We can see that increasing the value of $\alpha$ leads to an increased amount of money laundered per example, but also to an increase in the number of triggers and in the mean score from the discriminator. This is expected since $\alpha$ is the weight given to the money laundering objective (which we are trying to maximise) in relation to the other losses. Furthermore, changing the value of $\gamma$ did not have a significant effect on the amount of money laundered. However, there is a clear pattern of lower values of $\gamma$ leading to more triggers and lower discriminator scores, while higher values have the opposite effect. This is also expected since $\gamma$ is the weight given to the proxy loss (which we are trying to minimise) relative to the discriminator loss.

We chose the best performing model from our hyperparameter search to be our improved generator. Our improved generator was able to create money flows of approximately 387000 dollars on average, with an average discriminator score of 0.05, while triggering the rule-based system at least once in 34% of the batches. This means that the addition of our discriminator as a complementary AML method resulted in less than half of the maximum volume of money flows going undetected. However, we should note that the generators that we trained do not represent all possible strategies of money flows, so it might be possible to create examples with higher volumes.

In Figure 4.8 we show the distribution of amount per transaction and the number of transactions in generated examples from our improved generator. We can see that the distribution of amounts has a larger range of values than the first generator (see Figure 4.5), similar to the real data (see Figure 4.3). The number of transactions is significantly smaller than in the examples from the first generator, but it is still orders of magnitude larger than in the real data.

### 4.4.4 Improved Discriminator

In the previous section, we train an improved version of the generator to avoid being detected by the discriminator, while still evading the rule-based system. This results in a significant decrease in the predictive power of the discriminator in these newly generated examples, as we can see in Figure 4.9.

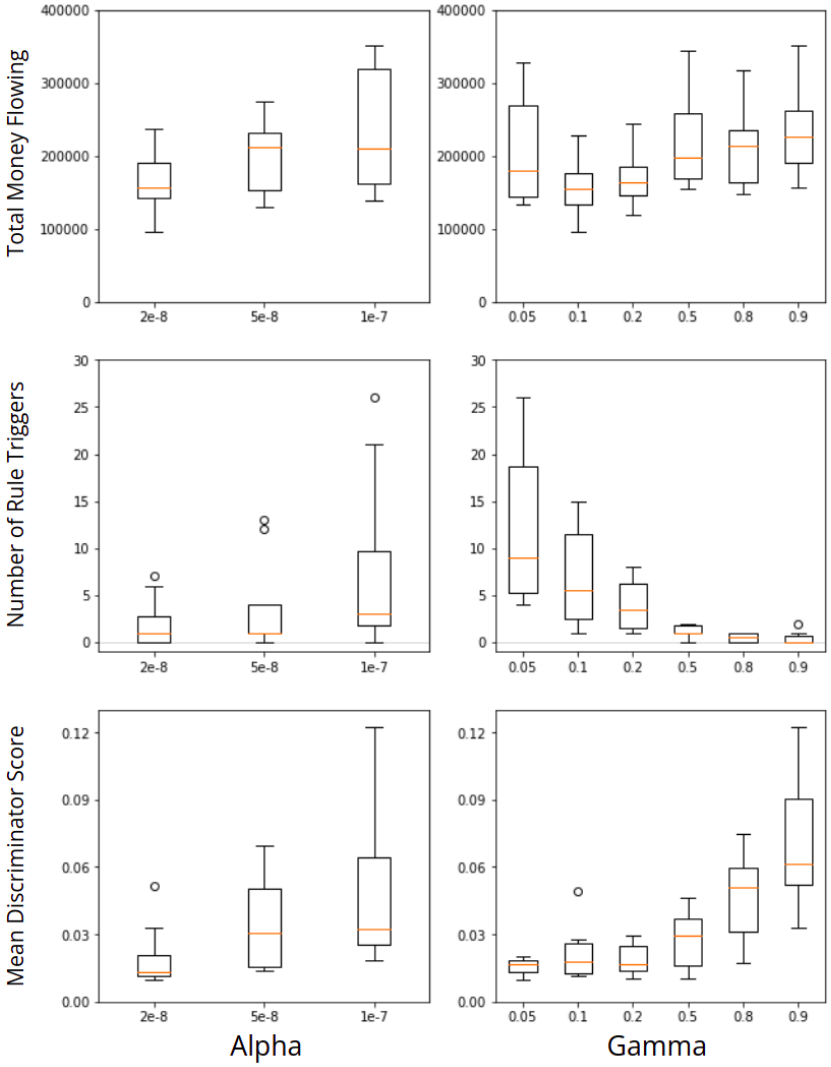


Figure 4.7: Results from grid search of improved generator's hyperparameters.

In this section, we describe how we fine-tune the discriminator such that it can correctly classify the original data as real and the generated data as synthetic. It is trained in same way as we describe in Section 4.4.2, but the synthetic examples are sampled from the improved generator (instead of the first generator) and the parameters from the neural network are initialised with the values from the parameters of the first discriminator.

Figure 4.10 shows the performance of the discriminator across the epochs of training, for various values of learning rate. The performance is depicted as we described in Section 4.4.2, measuring the gap between the scores of the two classes. We see that the discriminator approaches a perfect model, requiring less epochs than before.

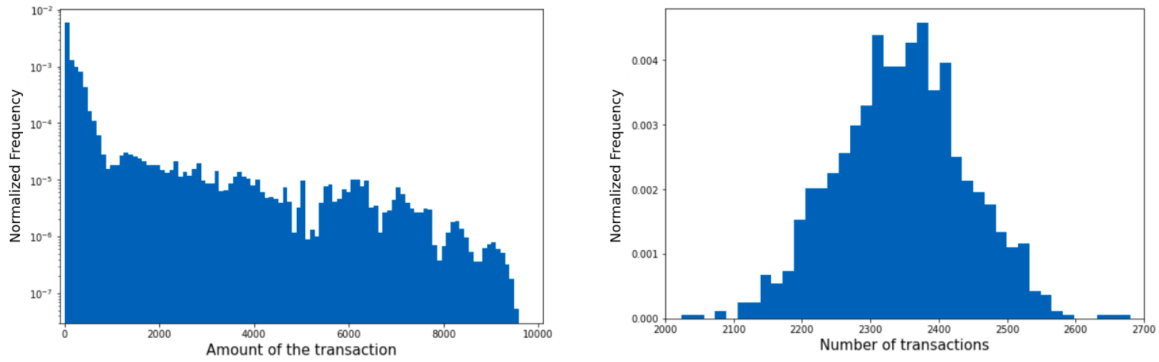


Figure 4.8: Normalised frequency of amounts and number of transactions per synthetic example of the improved generator.

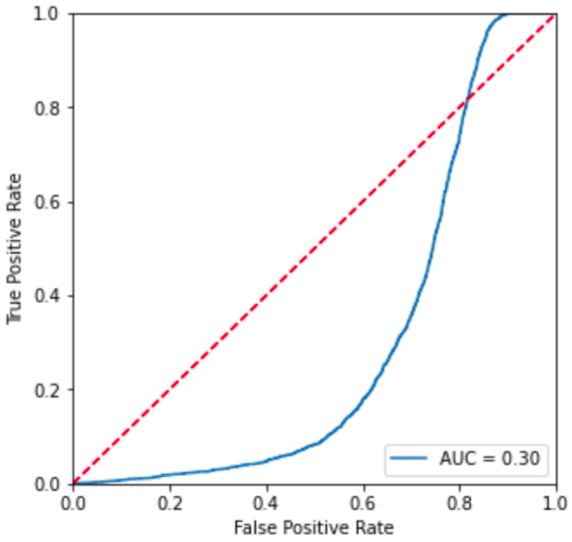


Figure 4.9: ROC curve of the first discriminator, using synthetic examples from the improved generator. The dotted line corresponds to the performance of randomly guessing the predictions.

## 4.5 Joint generator and discriminator training

In the previous section, we show that we can construct a generator that makes money flow through the financial institution without being captured by the AML rule system. We then train a discriminator to detect these synthetically generated money flows. We experimented with various iterations of generators and discriminators, who play a game of 'cat-and-mouse', taking turns to train and get the upper hand over the other. In this section, we train these models jointly, continuously updating their parameters like in classical GAN training.

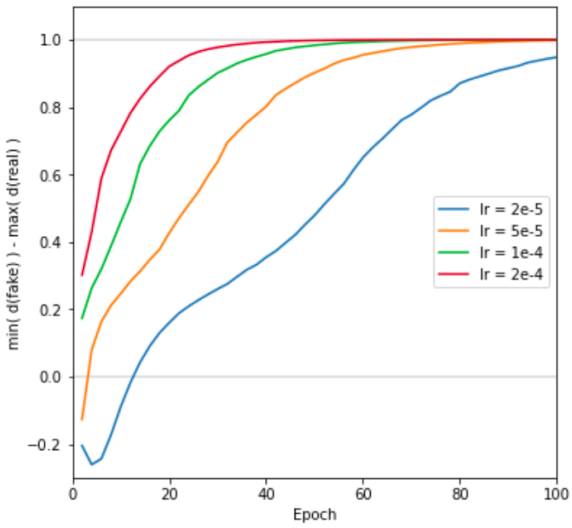


Figure 4.10: Results from fine-tuning the discriminator with various values of learning rate.

In this experiment, we use the same architectures for the components of our solution as we described in the previous section. The batch size is 16 and the loss of the generator is given by Eq. 3.5. We use the Wasserstein GAN loss for the $\Lambda_{disc}$, using the proposed values for the clipping parameter and number of iterations of the critic of the original paper [ACB17].

We start by performing a hyperparameter optimization by doing a grid search in Section 4.5.1. Then, in Section 4.5.2, we discuss how we can compare the performance of the different discriminators. Then, in Section 4.5.3, we compare the distributions of real and generated data. And lastly, in Section 4.5.4, we do an ablation study, showing how removing either the money laundering objective or the proxy network impacts the rest of the architecture.

### 4.5.1 Hyperparameter optimization

In order to evaluate how some of the hyperparameters affect our method, we perform a grid search over various values of several hyperparameters, listed in Table 4.8. We start one training run with each combination of hyperparameters and let it train for 300 epochs[4].

We show some relevant results from the grid search in Figure 4.11. We can see that increasing the value of $\alpha$ leads to higher volumes of money flowing, which is expected

[4]An epoch was defined as 16 batches.

| | |
|---|---|
| $\alpha$ | $2e^{-9}$, $5e^{-9}$, $1e^{-8}$, $2e^{-8}$, $5e^{-8}$, $1e^{-7}$ |
| $\beta$ | 0.9, 1.1 |
| $\gamma$ | 0.2, 0.5, 0.8 |
| learning rate | $5e^{-5}$, $1e^{-4}$, $2e^{-4}$ |

Table 4.8: Values of hyperparameters of the joint training grid search. $\alpha$ and $\gamma$ are the coefficients from Eq 3.5 and $\beta$ is the coefficient from Eq 3.3. The learning rate is the same for the generator and for the discriminator.

since $\alpha$ is the weight of the money laundering objective. Furthermore, we can see that increasing the value of $\gamma$ leads to less frequency of triggers while having little effect on the total amount of money flowing. These results are similar to what we saw in Section 4.4.3.

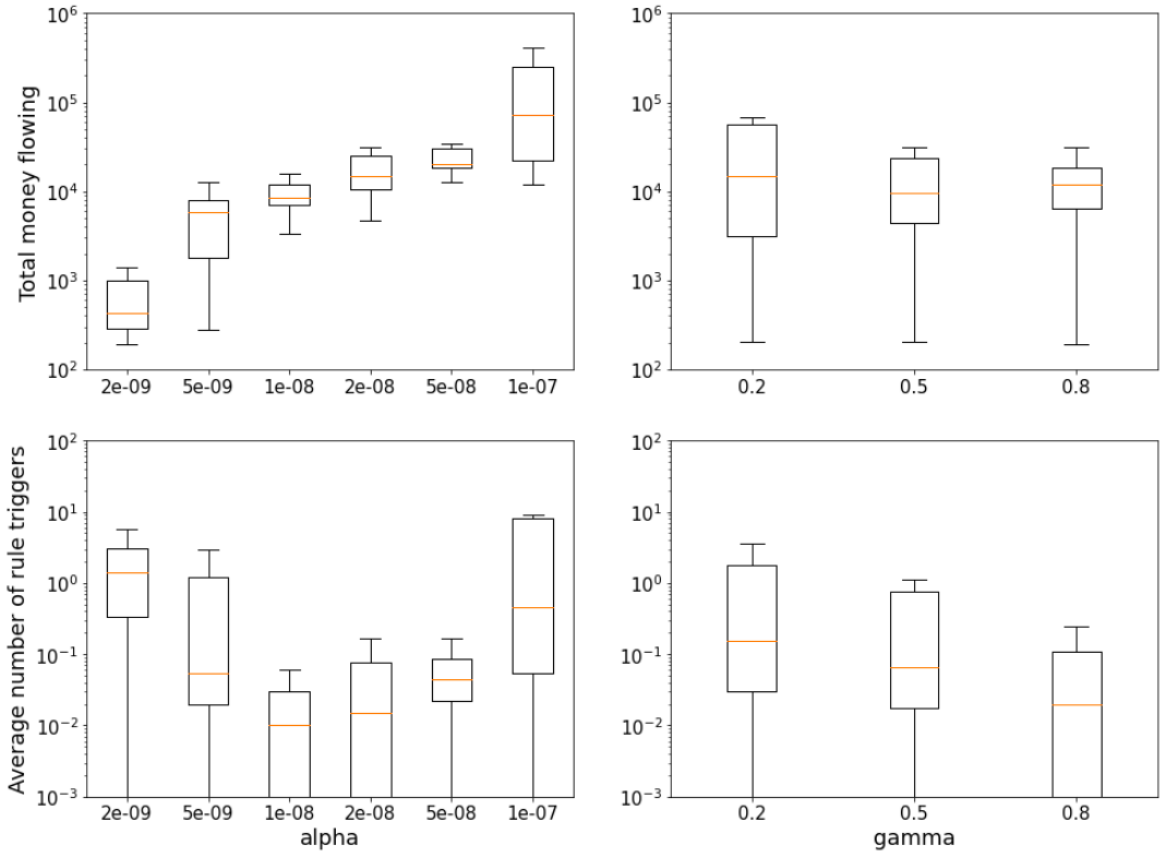


Figure 4.11: Distribution of total money flowing and average number of transactions per synthetic example for different values of the hyperparameters $\alpha$ and $\gamma$.

Lastly, here we can see that as we increase the value of $\alpha$, the average number of triggers first gets smaller and then bigger again. This is because, for lower values of

$\alpha$, there are too few transactions being made, which trigger scenario 4 (activity in dormant account). For higher values of $\alpha$, the increase of the volume of transactions leads to the triggering of scenarios 5, 6 and 7.

This experiment shows that different hyperparameter settings result in different distributions of generated examples (with different amounts of money flowing and different number of triggers). In other words, the generators learn different strategies when trained with different hyperparameters. As such, one way to increase the diversity of generated examples of money flows is to sample from different generators, trained in different settings. Another possibility would be to have special schedulers for the hyperparameters and save snapshots of the generators at different moments during training.

### 4.5.2 Discriminator Performance

Besides the total amount of money flowing and the number of rule triggers that we shown in Figure 4.11 (which are measuring the fitness of the generator), we also want to measure the predictive performance of the discriminator, in order to compare different hyperparameter setups. We do this by comparing the area under the ROC curve of different discriminators. However, if we only evaluate each discriminator using generated data from its corresponding generator, the results may be misleading. For example, say we are comparing two different runs of the complete architecture. We will call $G_1$ and $D_1$ to the generator and discriminator of the first run and $G_2$ and $D_2$ to the generator and discriminator of the second. We could measure the performance of $D_1$ by using a test set of real data and generated data from $G_1$, and do the same with $D_2$ and $G_2$, and arrive to the conclusion that $D_1$ is more accurate than $D_2$. However, that could be due to the distribution of generated data from $G_2$ being much closer to the real data (and as such more difficult to distinguish) than the distribution of generated data from $G_1$.

In order to make a fair comparison of the discriminators, we need to test them using the same dataset. So, we build a test dataset of synthetic examples by gathering several batches of outputs of all of the corresponding generators. The real examples can be sampled from a test dataset of real data. We use these datasets to measure the performance of the discriminators.

We show how different values of $\alpha$ and $\gamma$ affect the performance of the discriminator in Figure 4.12. The metric that we used was the area under the ROC curve of the predictions of the discriminator, measured after the 300 epochs of training have been

completed. The train AUC is obtained when the discriminator is distinguishing between the train set of real data and the generated data from its corresponding generator (the setup during training). The test AUC is obtained when the discriminator is distinguishing between the test set of real data and the test set of synthetic data, as we described in the previous paragraph.

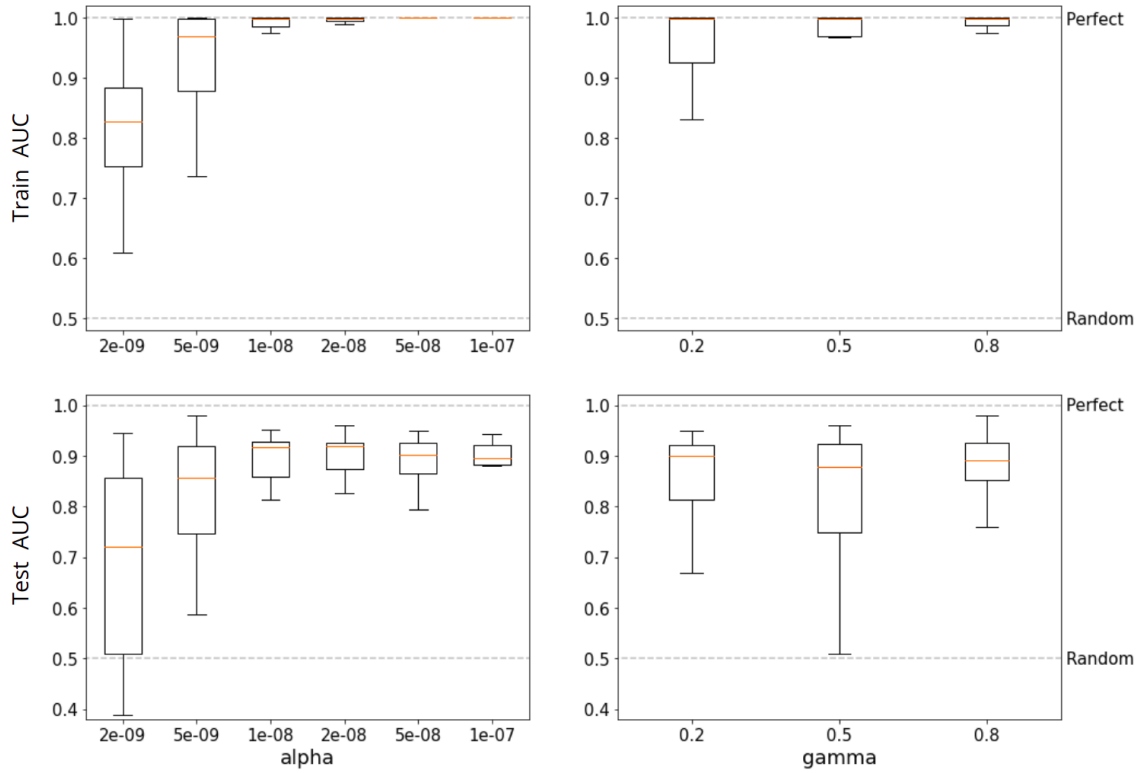


Figure 4.12: Distribution of the discriminator's area under the ROC curve for train and test, for different values of the hyperparameters $\alpha$ and $\gamma$.

We see that training with smaller values of $\alpha$ and $\gamma$ results in worse predicting performance (smaller train AUC). This is expected, since lower values of $\alpha$ means less weight for the money laundering objective, so the distribution of the generated data will be closer to the real data, meaning that it is harder to distinguish. Also, lower values of $\gamma$ means more weight to the discriminator (and less to the rules), which again teaches the generator to create data closer to the real data. For higher values of $\alpha$ and $\gamma$, the discriminator is near perfect during training.

The distribution of the test AUCs is lower than the train AUCs, but most discriminators show good predicting power on the test set as well. This means that the discriminators are not too overfitted to their corresponding generator and are able to

maintain good performance when classifying synthetic data from other generators as well. Again, lower values of $\alpha$ leads to worse discriminators, some being even worse than random classifiers. There is no clear correlation between $\gamma$ and the test AUC of the discriminator.

### 4.5.3 Comparison of real and generated data

In the previous section, we evaluate the predictive power of the discriminator. In this section, we examine the outputs of the generator, comparing real and synthetic data.

In Figure 4.13 we show the empirical distributions of total amount of money flowing, amount per transaction and number of transactions, in each example of real and generated data. The real data distributions are sampled from our test dataset. In the top row, the generated data distribution is the result of sampling 128 examples from each generator of the grid search that successfully learns to avoid triggering the rules system (on average less than 1 trigger per example). Out of the 108 runs of the grid search, 101 verify this criterion. In the bottom row, the generated data distribution is the result of sampling 4096 examples from the single generator of the run of the grid search whose discriminator has the best prediction score at test (measured as described in Sections 4.5.2). We will call this generator $G_1$.

Even though there is some overlap between the two distributions, generated data from all generators has consistently higher amounts of money flowing than real data. This is expected since the generator has the incentive of the money laundering objective to create bigger money flows. The increase of money flowing is not due to higher amounts of money being sent, since there is a substantial overlap of the real and generated distributions of said quantity. The generated data has higher money flows because the number of transactions is larger than real data, with the two distributions having almost no overlap (top right panel of Figure 4.13).

In the generated data from $G_1$, we see that the variance of the amount of money flowing and the number of transactions is much smaller compared to the previous generated distribution. However, the distribution of the amount per transaction is quite similar to the previous one. Also, it is interesting to note that $G_1$ is from the run of the best performing discriminator and we notice that the money flowing distributions have little overlap but no gap between them. As such, during training, this discriminator learns to distinguish between two close but non-overlapping distributions. It is possible that training with examples close to the "decision boundary" of real or generated data is the reason for the discriminator generalising well to other generators.

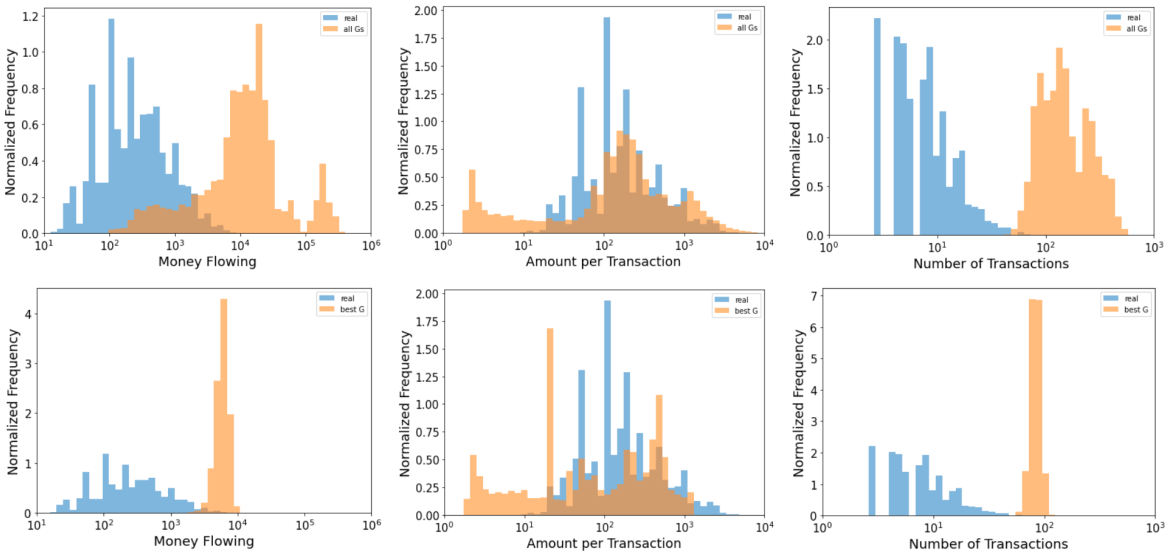


Figure 4.13: Distributions of total amount of money flowing, amount per transaction and number of transactions, in each example of real or generated data. In the top row, the generated data was gathered from all the generators of the grid search that do not trigger the rules. In the bottom row, the generated data was gathered from the generator corresponding to the best performing discriminator.

In Figure 4.14 we show three examples[5] of real data and generated data from $G_1$. We represent time across the $x$ axis and show how much money and how many transactions happen on each day. We verify again that the amount transacted in each day follows similar distributions for real and generated data alike. However, the frequency of transactions is much higher in generated data.

Most generators learn to avoid alerting the rules system, having on average less than one trigger per generated example (the full 3D tensor). We now wondered whether the generators are more closely saturating the thresholds imposed by the rules, compared to the real data. We test this hypothesis by measuring how close they are to trigger each of the rule scenarios using a continuous metric, instead of binary. We define this "continuous trigger" as shown in Eq. 4.1, in which we measure how close account $a$ was to trigger rule scenario $r$ on day $t$ using a value between 0 and 1, where 1 corresponds to a trigger. Each rule scenario is expressed in disjunctive normal form, with scenario $r$ being the disjunction of potentially several conjunctions. Each conjunction $c$ of $r$ can have multiple conditions, each comparing a profile $p$ with its threshold (usually of the form $profile < threshold$). So, we compare how close a profile is to reaching

[5]One example is a single 3D tensor.

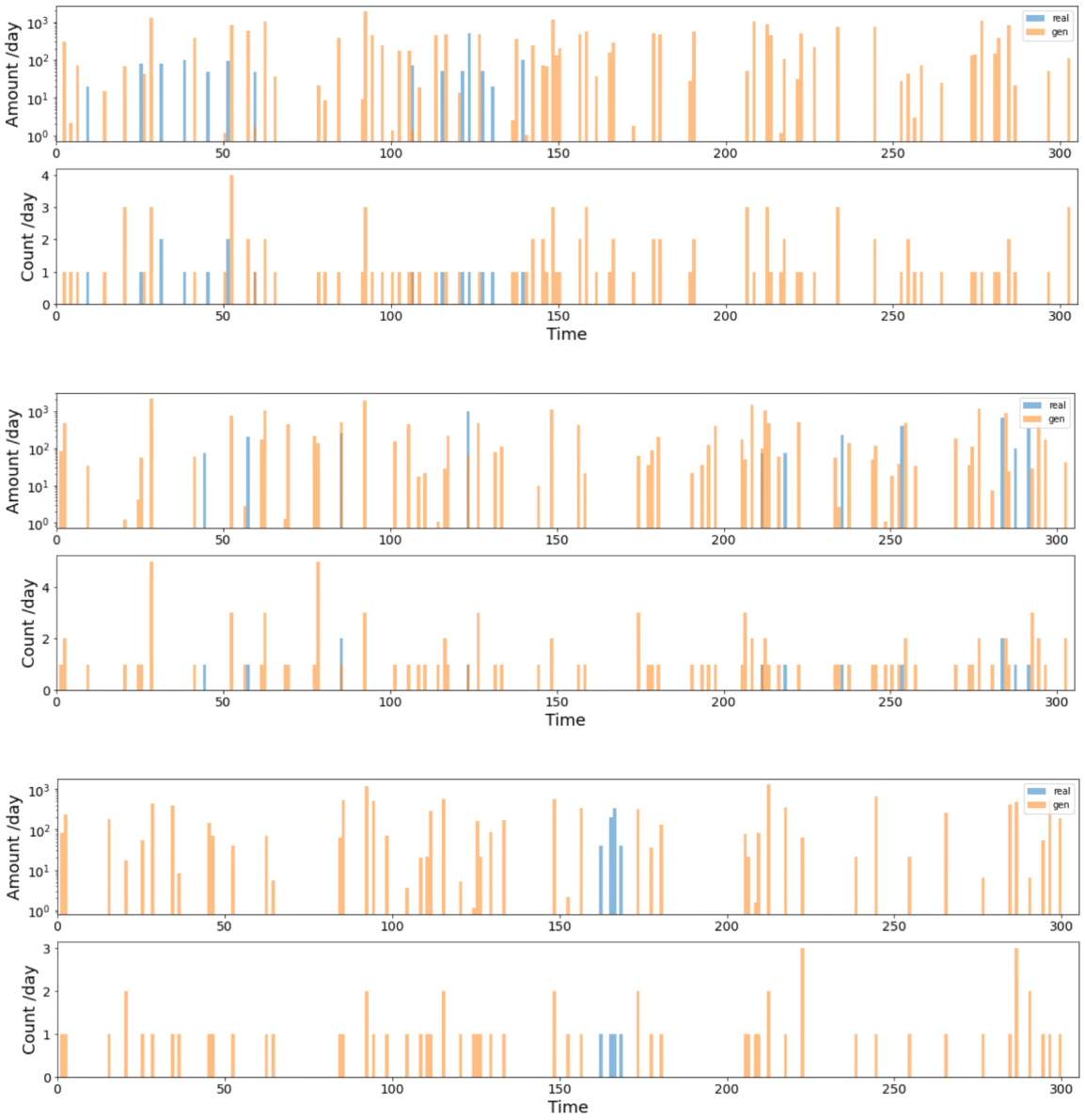


Figure 4.14: Three examples of real and generated data. The $x$ axis represents time with a granularity of a day (the third dimension of the tensor) and the $y$ axis shows either the sum of amounts or the number of transactions that happened on that day.

its threshold using the ratio between them; then we combine multiple profiles of a conjunction using the product operation; and lastly we combine multiple conjunctions of a rule scenario using the maximum operation. If instead the condition is of the form $profile > threshold$, we calculate the inverse ratio inside the minimum operation.

$$\text{contTrig}(a,t,r) = \max_{c \in r}(\prod_{p \in c} \min(1, \frac{p(a,t)}{\text{threshold}(p)})) \tag{4.1}$$

We show how close the generator is to trigger each rule scenario in Figure 4.15. Since the rules are applied for a single account on a specific day, we sample 100000 accounts and days which have a transaction from different examples of the real test dataset and from $G_1$. The generated data is closer to triggering scenarios 5 and 6, which monitor the total amount of money transacted per week, meaning that the generator cannot create much larger money flows than what it learned to do. Most generators are halfway through to trigger those rule scenarios, which can be the result of the discriminator's feedback that stops it from increasing further, or multiple rules combined together that keeps the situation like this. Overall, even though the majority of these samples is far from triggering the rules, the generated data is closer than the real data to trigger 4 of the 7 scenarios and they are similar in the other 3 scenarios.

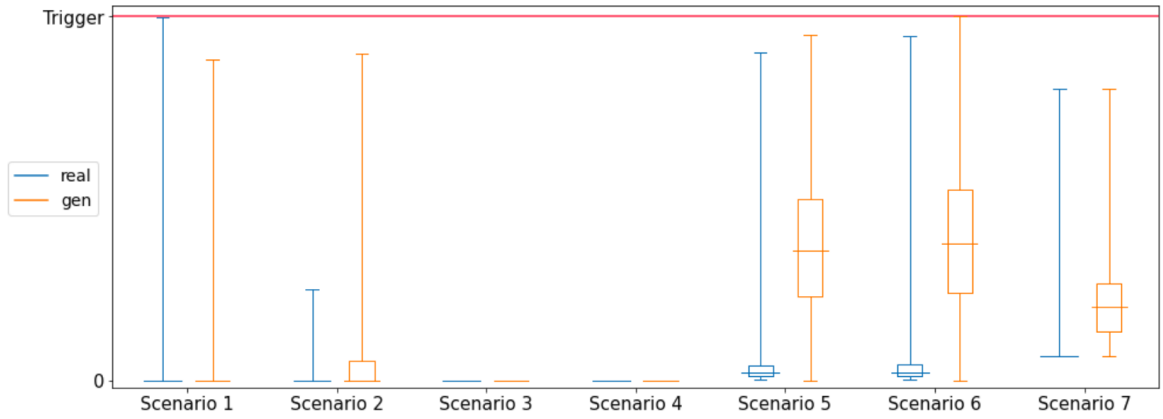


Figure 4.15: Boxplots depicting how close the real and generated data are to trigger each of the rule scenarios.

### 4.5.4 Ablation Study

We performed an ablation study in order to verify the necessity and the impact that the money laundering objective and the rule proxy network have on our method. To do this, we performed two smaller grid searches in which we set either $\alpha$ or $\gamma$ to 0, and trained one network for each combination of values of the remaining hyperparameters, using the same values from Table 4.8.

First, we experimented removing the money laundering objective by setting $\alpha$ to 0. The results from this experiment are depicted in Figure 4.16. The removal of the

money laundering objective makes the distribution of money flowing of generated data to overlap more with the distribution of real data than before. This means that the higher volumes of money flowing that we see in Figure 4.13 are a result of the money laundering objective's feedback and, in its absence, the generator reverts to mimicking the real distribution more closely. We found that this decrease in money flowing is the result of lowering both the amount per transaction and the number of transactions of the generated data examples.

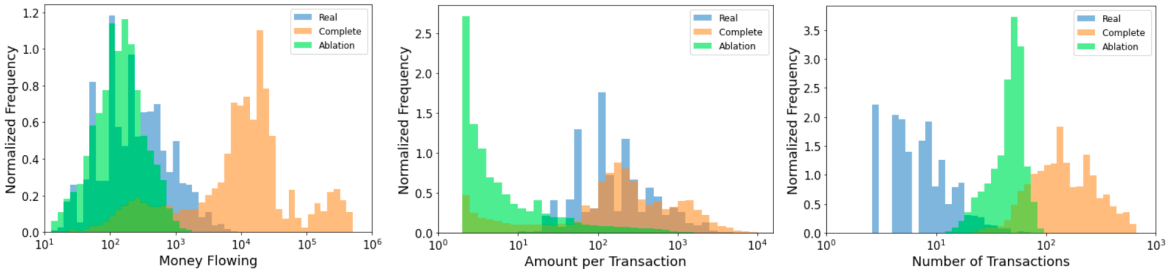


Figure 4.16: Distributions of total amount of money flowing, amount per transaction and number of transactions, in each example of real or generated data, using either the complete or ablated architectures (no money laundering objective).

Then, we experimented removing the rule proxy network by setting $\gamma$ to 0. The results from this experiment are depicted in Figure 4.17. In this case, the removal of the proxy network leads to larger money flows, due to higher amounts per transaction and much more frequent transactions. Actually, the 3D tensor representation is completely full (all entries have positive values). However, this results in a substantial increase of the number of triggers as well. We could increase the weight of the discriminator's feedback to the generator, which would bring the distribution of generated data closer to the distribution of real data, and thus avoid the triggers. However, we do not work in a typical GAN setting and it is not desirable for us to exactly mimic real data. Taken together with the money laundering objective, these results indicate that the rule proxy network is essential to avoid triggering the rules.

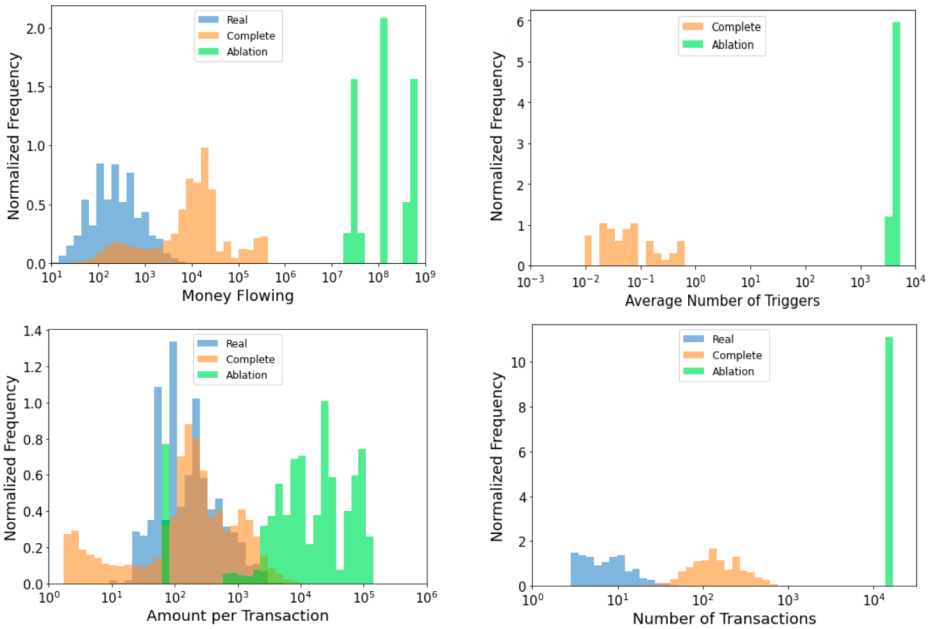


Figure 4.17: Distributions of total amount of money flowing, average number of rule triggers, amount per transaction and number of transactions, in each example of real or generated data, using either the complete or ablated architectures (no rule proxy network).

# Conclusion 5

In this work we proposed a deep learning solution to detect complex patterns of money laundering that rule-based AML systems commonly in place cannot capture. We train this classifier using real legitimate transactions as examples of the negative class, and synthetic data from another deep learning model (the generator) as examples of the positive class. This generator is trained to maximise the amount of money flowing through the internal accounts while avoiding triggering the AML system. Since the rules are not differentiable and the generator is trained using gradient descent, we train a third deep learning model to serve as a proxy for the rules, mimicking their triggering logic while providing a gradient to the generator.

We showed that the proxy network was able to mimic the rules with different levels of fidelity depending on the amount of inductive bias that we encode in the network architecture.

Also, the transactions produced by the generator were significantly different from the real transactions of the Banking dataset that we used. They move much larger volumes of money than real data, while still avoiding triggering the rules.

However, the fact that the distributions of real and synthetic data were quite different meant that distinguishing between them was quite easy. As such, most discriminators were able to achieve very good performance.

There are several experiments and possible changes to be explored in order to further improve this work.

- We use a small number of features to represent each transaction. However, analysts use much more in order to decide if some activities are suspicious or not, for example geography related features. Our method could be applied with other types of features encoded as well, but it would require changing the data representation and the architecture of the generator and the discriminator (maybe similar to [XSCIV19]).

- We are generating and detecting one specific pattern of money laundering, which is the money flow. There are several other known patterns of money laundering that could potentially be explored. The only requirement to generate other money laundering patterns using our method is to be able to write a reward function to encode it (like our money laundering objective is encoding the money flow goal).
- In this work, the distribution of real and synthetic data are quite different and easily distinguishable. We probably don't want our data to follow exactly the same distribution as real legitimate activities. But it would be interesting to try different degrees of realism and see how they impact the performance and robustness of the discriminator.

Overall, this work showed promising results on a problem with real world impact, which motivates the authors to continue working on improving upon these achievements.

# References


[AB15] Claudio Alexandre and Joao Balsa. Client profiling for an anti-money laundering system. *arXiv preprint arXiv:1510.00878*, 2015.

[AB16] Claudio Alexandre and João Balsa. Integrating client profiling in an anti-money laundering multi-agent based system. In *New Advances in Information Systems and Technologies*, pages 931–941. Springer, 2016.

[ACB17] Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein generative adversarial networks. In *International conference on machine learning*, pages 214–223. PMLR, 2017.

[BAG+20] Bernardo Branco, Pedro Abreu, Ana Sofia Gomes, Mariana SC Almeida, João Tiago Ascensão, and Pedro Bizarro. Interleaved sequence rnns for fraud detection. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 3101–3109, 2020.

[BLP+21] Ricardo Barata, Miguel Leite, Ricardo Pacheco, Marco OP Sampaio, João Tiago Ascensão, and Pedro Bizarro. Active learning for online training in imbalanced data streams under cold start. *arXiv e-prints*, pages arXiv–2107, 2021.

[BTLLW21] Sam Bond-Taylor, Adam Leach, Yang Long, and Chris G Willcocks. Deep generative modelling: A comparative review of vaes, gans, normalizing flows, energy-based and autoregressive models. *arXiv preprint arXiv:2103.04922*, 2021.

[CNT+14] Zhiyuan Chen, Amril Nazir, Ee Na Teoh, Ettikan Kandasamy Karupiah, et al. Exploration of the effectiveness of expectation maximization algorithm for suspicious transaction detection in anti-money laundering. In *2014 IEEE Conference on Open Systems (ICOS)*, pages 145–149. IEEE, 2014.

## REFERENCES


[CR17] Andrea Fronzetti Colladon and Elisa Remondi. Using social network analysis to prevent money laundering. *Expert Systems with Applications*, 67:49–58, 2017.

[CSMV17] Ramiro Daniel Camino, Radu State, Leandro Montero, and Petko Valtchev. Finding suspicious activities in financial transactions and distributed ledgers. In *2017 IEEE International Conference on Data Mining Workshops (ICDMW)*, pages 787–796. IEEE, 2017.

[CTN+18] Zhiyuan Chen, Ee Na Teoh, Amril Nazir, Ettikan Kandasamy Karuppiah, Kim Sim Lam, et al. Machine learning techniques for anti-money laundering (aml) solutions in suspicious transaction detection: a review. *Knowledge and Information Systems*, 57(2):245–285, 2018.

[CWD+18] Antonia Creswell, Tom White, Vincent Dumoulin, Kai Arulkumaran, Biswa Sengupta, and Anil A Bharath. Generative adversarial networks: An overview. *IEEE Signal Processing Magazine*, 35(1):53–65, 2018.

[DCK18] Nicola De Cao and Thomas Kipf. Molgan: An implicit generative model for small molecular graphs. *arXiv preprint arXiv:1805.11973*, 2018.

[DJSW09] Xinwei Deng, V Roshan Joseph, Agus Sudjianto, and CF Jeff Wu. Active learning through sequential design, with applications to detection of money laundering. *Journal of the American Statistical Association*, 104(487):969–981, 2009.

[DKB14] Laurent Dinh, David Krueger, and Yoshua Bengio. Nice: Non-linear independent components estimation. *arXiv preprint arXiv:1410.8516*, 2014.

[FAT99] Financial Action Task Force FATF. What is money laundering. *Policy Brief July 1999*, 1999.

[Gao09] Zengan Gao. Application of cluster-based local outlier factor algorithm in anti-money laundering. In *2009 International Conference on Management and Service Science*, pages 1–4. IEEE, 2009.

[GKL+21] Paulina Grnarova, Yannic Kilcher, Kfir Y Levy, Aurelien Lucchi, and Thomas Hofmann. Generative minimization networks: Training gans without competition. *arXiv preprint arXiv:2103.12685*, 2021.

[GPAM+14] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. *Advances in neural information processing systems*, 27, 2014.

[GSW+20] Jie Gui, Zhenan Sun, Yonggang Wen, Dacheng Tao, and Jieping Ye. A review on generative adversarial networks: Algorithms, theory, and applications. *arXiv preprint arXiv:2001.06937*, 2020.

[ICL12] Angela Samantha Maitland Irwin, Kim-Kwang Raymond Choo, and Lin Liu. An analysis of money laundering and terrorism financing typologies. *Journal of Money Laundering Control*, 2012.

[JLH+20] Martin Jullum, Anders Løland, Ragnar Bang Huseby, Geir Ånonsen, and Johannes Lorentzen. Detecting money laundering transactions with machine learning. *Journal of Money Laundering Control*, 2020.

[KLRH13] Nida S Khan, Asma S Larik, Quratulain Rajput, and Sajjad Haider. A bayesian approach for suspicious financial activity reporting. *International Journal of Computers and Applications*, 35(4):181–187, 2013.

[KT11] Liu Keyan and Yu Tingting. An improved support-vector network model for anti-money laundering. In *2011 Fifth International Conference on Management of e-Commerce and e-Government*, pages 193–196. IEEE, 2011.

[KW13] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[LCQ+17] Xurui Li, Xiang Cao, Xuetao Qiu, Jintao Zhao, and Jianbin Zheng. Intelligent anti-money laundering solution based upon novel community detection in massive transaction networks on spark. In *2017 fifth international conference on advanced cloud and big data (CBD)*, pages 176–181. IEEE, 2017.

[LH11] Asma S Larik and Sajjad Haider. Clustering based anomalous transaction reporting. *Procedia Computer Science*, 3:606–610, 2011.

[LJZ08] Lin-Tao Lv, Na Ji, and Jiu-Long Zhang. A rbf neural network model for anti-money laundering. In *2008 International Conference on Wavelet Analysis and Pattern Recognition*, volume 1, pages 209–215. IEEE, 2008.

## REFERENCES


[LLL+20] Xiangfeng Li, Shenghua Liu, Zifeng Li, Xiaotian Han, Chuan Shi, Bryan Hooi, He Huang, and Xueqi Cheng. Flowscope: Spotting money laundering based on graphs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2020.

[LR06] Michael Levi and Peter Reuter. Money laundering. *Crime and justice*, 34(1):289–375, 2006.

[LSA+20] Joana Lorenz, Maria Inês Silva, David Aparício, João Tiago Ascensão, and Pedro Bizarro. Machine learning methods to detect money laundering in the bitcoin blockchain in the presence of label scarcity. *arXiv preprint arXiv:2005.14635*, 2020.

[LZ10] Xuan Liu and Pengzhu Zhang. A scan statistics based suspicious transactions detection model for anti-money laundering (aml) in financial institutions. In *2010 International Conference on Multimedia Communications*, pages 210–213. IEEE, 2010.

[OTS+21] Catarina Oliveira, João Torres, Maria Inês Silva, David Aparício, João Tiago Ascensão, and Pedro Bizarro. Guiltywalker: Distance to illicit nodes in the bitcoin network. *arXiv preprint arXiv:2102.05373*, 2021.

[PL16] Thai Pham and Steven Lee. Anomaly detection in bitcoin network using unsupervised learning methods. *arXiv preprint arXiv:1611.03941*, 2016.

[RH11] Saleha Raza and Sajjad Haider. Suspicious activity reporting using dynamic bayesian networks. *Procedia Computer Science*, 3:987–991, 2011.

[RH21] Lars Ruthotto and Eldad Haber. An introduction to deep generative modeling. *GAMM-Mitteilungen*, page e202100008, 2021.

[RM15] Danilo Rezende and Shakir Mohamed. Variational inference with normalizing flows. In *International conference on machine learning*, pages 1530–1538. PMLR, 2015.

[RMC15] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[Sal05] Lawrence M Salinger. *Encyclopedia of white-collar & corporate crime*, volume 1. Sage, 2005.

[SGZ+16] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training gans. *Advances in neural information processing systems*, 29:2234–2242, 2016.

[SNY+16] Reza Soltani, Uyen Trang Nguyen, Yang Yang, Mohammad Faghani, Alaa Yagoub, and Aijun An. A new algorithm for money laundering detection based on structural similarity. In *2016 IEEE 7th Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON)*, pages 1–7. IEEE, 2016.

[SW08] Friedrich Schneider and Ursula Windischbauer. Money laundering: some facts. *European Journal of Law and Economics*, 26(3):387–404, 2008.

[SWC+16] David Savage, Qingmai Wang, Pauline Chou, Xiuzhen Zhang, and Xinghuo Yu. Detection of money laundering groups using supervised learning in networks. *arXiv preprint arXiv:1608.00708*, 2016.

[SZZ+21] Xiaobing Sun, Jiabao Zhang, Qiming Zhao, Shenghua Liu, Jinglei Chen, Ruoyu Zhuang, Huawei Shen, and Xueqi Cheng. Cubeflow: Money laundering detection with coupled tensors. In *PAKDD (1)*, pages 78–90. Springer, 2021.

[TH+12] Tijmen Tieleman, Geoffrey Hinton, et al. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

[TY05] Jun Tang and Jian Yin. Developing an intelligent data discriminating system of anti-money laundering based on svm. In *2005 International conference on machine learning and cybernetics*, volume 6, pages 3453–3457. IEEE, 2005.

[UVW09] Brigitte Unger and Frans Van Waarden. How to dodge drowning in data? rule-and risk-based anti money laundering policies compared. *Review of Law & Economics*, 5(2):953–985, 2009.

[Wag19] Dominik Wagner. Latent representations of transaction network graphs in continuous vector spaces as features for money laundering detection. *SKILL 2019-Studierendenkonferenz Informatik*, 2019.

[WCS+18] Mark Weber, Jie Chen, Toyotaro Suzumura, Aldo Pareja, Tengfei Ma, Hiroki Kanezashi, Tim Kaler, Charles E Leiserson, and Tao B Schardl.

Scalable graph learning for anti-money laundering: A first look. *arXiv preprint arXiv:1812.00076*, 2018.

[WD09] Xingqi Wang and Guang Dong. Research on money laundering detection based on improved minimum spanning tree clustering and its application. In *2009 Second international symposium on knowledge acquisition and modeling*, volume 2, pages 62–64. IEEE, 2009.

[WDC+19] Mark Weber, Giacomo Domeniconi, Jie Chen, Daniel Karl I Weidele, Claudio Bellei, Tom Robinson, and Charles E Leiserson. Anti-money laundering in bitcoin: Experimenting with graph convolutional networks for financial forensics. *arXiv preprint arXiv:1908.02591*, 2019.

[WRD+03] R Cory Watkins, K Michael Reynolds, Ron Demara, Michael Georgiopoulos, Avelino Gonzalez, and Ron Eaglin. Tracking dirty proceeds: exploring data mining technologies as tools to investigate money laundering. *Police Practice and Research*, 4(2):163–178, 2003.

[WY07] Su-Nan Wang and Jian-Gang Yang. A money laundering risk evaluation method based on decision tree. In *2007 international conference on machine learning and cybernetics*, volume 1, pages 283–286. IEEE, 2007.

[XSCIV19] Lei Xu, Maria Skoularidou, Alfredo Cuesta-Infante, and Kalyan Veeramachaneni. Modeling tabular data using conditional gan. *arXiv preprint arXiv:1907.00503*, 2019.