Chapter 3

Generation of Synthetic Datasets in the Context of Computer Networks using Generative Adversarial Networks

Thiago Caproni Tavares (IFSULDEMINAS), Ariel Góes de Castro (UNICAMP), Leandro C. de Almeida (IFPB), Washington Rodrigo Dias da Silva (UFSCAR), Christian Esteve Rothenberg (UNICAMP) and Fábio Luciano Verdi (UFSCAR)

Abstract

This chapter explores the domain of generative artificial intelligence, focusing on its transformative impact in producing network datasets. Our tutorial delves into Generative Adversarial Networks (GANs), initially pivotal in image and media synthesis but now extended to computer networking, offering nuanced capabilities in generating synthetic data and facilitating data-driven predictions, classifications, and experimentation. This work discusses the utility of GANs in computer networks, specifically in generating synthetic data while preserving privacy and enhancing dataset representativeness. The tutorial section of this book chapter elucidates on generating synthetic time series data using GANs, with practical applications ranging from telemetry data generation to creating synthetic Packet Capture (PCAP) files. Further contributions of this work include exploring the use of synthetic data in training Reinforcement Learning (RL) agents for resource optimization, highlighting the broader implications and emerging trends in applying generative AI in computer networks.

3.1. Introduction and Motivation

Generative artificial intelligence encompasses a suite of algorithms and models endowed with the capacity to produce diverse forms of data, including images, videos, text, and assorted digital media. In contemporary times, this paradigm has gained noteworthy traction among individuals outside academic circles, attributed mainly to the advent of ChatGPT [34], a prominent instance of a Large Language Model (LLM). This approach establishes a mechanism for comprehending and engendering human language expressions. Moreover, it represents an evolutionary progression from conventional Language Models, characterized by integrating substantial parameter magnitudes that yield exponential enhancements in learning capabilities.

An additional category of generative models is represented by Generative Adversarial Networks (GANs). Primarily introduced within the context of image synthesis, GAN has garnered considerable attention due to its efficacy in capturing intricate and high-dimensional data distributions. This ability is realized through two distinct Neural Networks — an entity designated as the generator and another as the discriminator — that engage in an interplay following the principles of game theory, as delineated in [15]. The fundamental operational paradigm involves the generator network producing synthetic data samples to deceive the discriminator. In parallel, the discriminator network undertakes the role of a judge, assessing the similarity between real and generated/synthetic data. The main objective is to generate a scenario wherein the discriminator's capacity to discern actual data from its synthetic counterparts is markedly diminished.

In this context, GAN has seen an extension of its application into the domain of Computer Networks over recent years. Navidan et al. [38] have undertaken a classification of GAN based on their specific application objectives. To illustrate, GANs can generate synthetic data to serve as inputs for distinct learning models. Alternatively, GANs may allocate one of their constituent neural networks—the generator or the discriminator—toward data prediction or classification tasks, respectively.

The utilization of GANs in the context of Computer Networks is contingent upon the specifics of the particular application. When employed as a synthetic data generator, GANs work as a simulator. Within this context, the synthetic data produced emulates the inherent distribution of the original dataset, thereby ensuring the preservation of privacy considerations. Moreover, these networks prove instrumental in tasks such as dataset augmentation and balancing, culminating in a dataset characterized by enhanced representational capacity. Consequently, the resultant model emerges as a conduit to share intricate dynamics of real environments while obfuscating inherent complexities and maintaining data quality integrity.

Within computer networks, the application of machine learning manifests across distinct modalities: direct deployment within real setups, engagement within simulated environments, or application of GANs to emulate specific environmental attributes. One of the advantages of GAN is its ability to imitate and embody the inherent characteristics of a given system, thereby creating a closer approximation to reality. This proficiency augments the efforts of both industrial practitioners and researchers, enabling the construction of experimental frameworks guided toward the intelligent orchestration of resources. [16] presents a group created in October 2023 by ATIS Alliance that aims to survey generative AI/ML use cases across the network, demonstrating the importance of generative AI in the context of Computer Networks.

This tutorial book chapter attempts to elucidate the fundamental principles of generating synthetic time series data using GANs. Furthermore, it will delve into the practical applications of GANs in generating telemetry data derived from a Programmable Data Plane (PDP) and creating Packet Captures (PCAPs). This tutorial showcases the diverse utility of synthetic data in various contexts, such as training an RL agent with the objective of PDP resource optimization and generating realistic PCAPs.

This chapter is structured as follows. Section 3.2 delves into the core concepts of GANs, and Section 3.3 explores their applications in In-band Network Telemetry (INT) and PDPs. Section 3.4 covers synthetic data generation's theoretical and practical aspects, including creating telemetry data and synthetic PCAP files. Section 3.5 discusses two use cases where synthetic data can be applied. Finally, Section 3.6 summarizes key findings and discusses emerging synthetic data generation trends and applications.

3.2. Fundamentals of Generative Adversarial Networks

This section provides a foundational overview of Generative Artificial Intelligence, introducing the core concepts and presenting a small literature survey. Section 3.2.1 delves into the intricacies of GANs, elucidating their operational principles and demonstrating their application in training RL agents. Section 3.2.2 covers the fundamental principles of PCAP generation, outlining the methodologies and techniques employed in creating synthetic network traffic data for analysis and testing purposes.

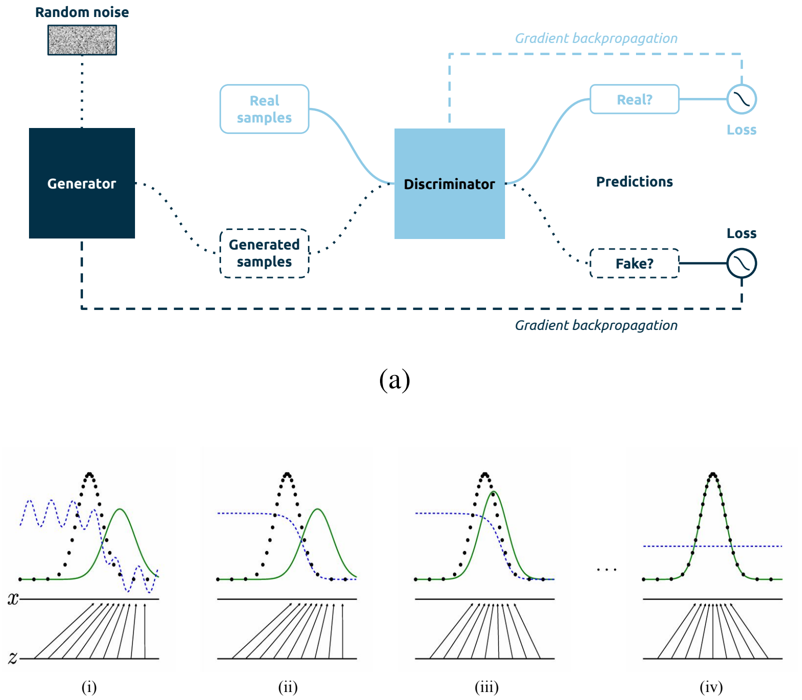
3.2.1. GANs and RL

Machine learning is commonly categorized into three paradigms: supervised, unsupervised, and RL [48]. As an illustration, the well-established GAN predominantly pertains to unsupervised learning. Within this framework are two competing modules: the generator and the discriminator. The first tries to produce synthetic data that mimics the actual data distribution, while the second tries to distinguish between real and synthetic data. The generator and the discriminator are trained in an adversarial manner until they reach an equilibrium where the discriminator cannot tell the difference between real and synthetic data (Figure 3.1).

Nevertheless, an alternative perspective emerges by considering the discriminator's role as akin to that of a supervised learning module. This is evident in the discriminator's utilization of actual data instances to ascertain the degree of resemblance between the values generated by the generator and those originating from actual data sources.

GANs have found diverse applications within the ambit of Computer Networks. Their utility spans data generation, system optimization, and data classification, as evidenced by existing literature [56]. The choice of the GAN variant depends on the specific utilization context, which necessitates tailored selections to align with distinct objectives. Although the predominant application of GAN has traditionally been within computer vision, notable strides have been taken towards their adaptation for time series network data. Among the different GAN frameworks enlisted, including Vanilla GAN, Bidirectional GAN (BIGAN), Conditional GAN (CGAN), InfoGAN, CycleGAN, Energy-based GAN (EBGAN), and Least Square GAN (LSGAN), there is an inclination towards encompassing time series data. Recent years have witnessed the ascendancy of specific GAN iterations that have found prominence within the time series domain, including Conditional Tabular GAN (CTGAN), DoppelGANger, and TimeGAN, which are specially tailored to learn the intricacies of time series data [37].

The application of GAN for data generation yields versatile utility across diverse scenarios. In this case, it can effectively address data imbalance concerns, serving to rectify skewed datasets or serve as a mechanism to impute absent values. Its strength



(a)

(b)

Figure 3.1: Adversarial training overview: The training pipeline is shown in figure (a), and an illustration of the process is shown in figure (b). The GAN trains by updating the discriminator function D (blue dashed line) and the generator function G simultaneously, enabling D to differentiate real data samples from the distribution p_x (black dotted line) from generated samples of p_g (green solid line). The z axis represents the input noise, and the x axis represents the real data domain. The arrows illustrate how mapping from z to x imposes non-uniformity (p_g) on transformed samples. Key steps include: (i) showing an adversarial pair near convergence, (ii) illustrating the training of D to discriminate data samples, (iii) after updates, D guides G to produce samples more akin to real data, and (iv) with enough capacity and training steps, G and D might converge to a point where neither can improve, as D cannot differentiate between p_x and p_g. Adapted from [15].

extends to privacy preservation, enabling the obfuscation of individuals' personally identifiable information. Presently, entities have emerged in the corporate landscape offering services tailored to the treatment of sensitive data, facilitating the creation of synthetic data generators that facilitate secure information exchange between organizations. In certain instances, these data generators are pivotal in constructing novel datasets. Such datasets subsequently find application within varying contexts, often in conjunction with complementary machine learning methodologies, such as RL, as demonstrated by [24].

RL is fundamentally characterized as a machine learning paradigm wherein an autonomous agent aims to optimize decision-making processes within an environment that is not known to the agent. Central to the construct of an RL algorithm is the interplay between the agent and the environment (Figure 3.2). A set of states defines the environment, while the agent is endowed with the capacity to execute actions upon this environment.

The outcomes of these agent-initiated actions, in turn, produce either rewards or penalties computed based on the observations derived from the environment. A positive environmental alteration following an action elicits a reward, while a harmful impact triggers a penalty. Thus, the overarching objective within RL is enhancing state-action pair optimization through iterative trial-and-error exploration of the agent's actions [47].

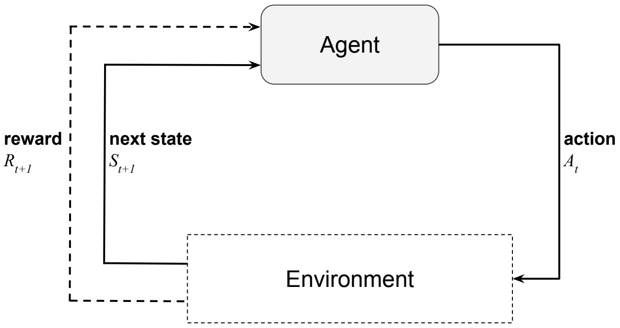


Figure 3.2: Agent-environment interaction. The diagram presented in this figure illustrates a sequential decision-making strategy called Markov Decision Process (MDP). Within this framework, an agent continually interacts with the environment by taking actions (A) at specific time steps (t) and observing subsequent states (S_{t+1}) resulting from those actions. A reward value (R_{t+1}) is generated for each interaction to assess the action effectiveness, which is expected to be maximized throughout the agent's training process. Adapted from [47].

Xiong et al. [49] extensively elucidates a plethora of applications of RL within the purview of mobile networks. These applications encompass domains such as Network Slicing, Power Control in Cellular Networks, Computation Offloading, Edge Caching, and other segments of the mobile network architecture characterized by resource optimization challenges. The adeptness of RL finds resonance, particularly in scenarios where the allocation and utilization of resources are paramount, ultimately culminating in enhanced efficiency and performance across diverse facets of mobile networks [18].

GAN and RL used jointly remain relatively underexplored within the existing literature. A comprehensive exploration highlighting the advantages gleaned from the confluence of these two methodologies is lacking. While RL has garnered substantial traction across many domains, the prospect of its integration with GAN has yet to be examined. Given RL's robustness in optimizing decisions and GAN's capacity to simulate intricate data distributions, this intersection carries considerable potential. GANs can model and generate realistic network traffic patterns, serving as training data for RL agents responsible for testing and optimizing network configurations and policies.

However, an important issue must be considered when using RL in an actual setup compared to a simulated scenario. Sim-to-real discrepancy is an important issue that must be addressed in the context of mobile network research. The concept is defined as the gap between simulation and real environments. It is not a particular problem in the field of communication, as it can also be observed in other domains, such as computer vision, natural language processing, robotic control, and autonomous driving [35]. The layered construction of computer networks increases the difficulty of accurately simulating a real configuration, complicating the discrepancy between these approaches.

Qiang et al. [29] discussed an automatic online service configuration in network slicing. The authors have proposed and implemented Atlas's solution on an end-to-end network prototype. The system aims to automate the network-slicing process to reduce the Sim-to-Real discrepancy. To do that, they created a methodology divided into three stages, from simulation to real network. The authors proposed using a new parameter-searching method based on Bayesian optimization. Complementary to that work, our study aims to show that it is possible to reduce the Sim-to-Real discrepancy using TimeGAN instead of Bayesian optimization. So, we show the feasibility of this approach by training a RL model and comparing its training using real and synthetic data.

The work presented by Hua et al. [24] introduces a strategy that merges deep RL with distributional modeling to improve the allocation of resources like bandwidth among various network slices. This technique elevates efficiency and service quality within network slicing setups. By harnessing advanced methodologies from RL and distributional modeling, this approach resolves the intricate issues of resource management within complex network-slicing contexts. It's worth noting that while the authors' approach employs a GAN to approximate the distribution of state-action values in an RL model, our method primarily focuses on generating simulated environmental data to facilitate RL training.

Falahatraftar et al. [13] also proposes an approach for addressing network slicing in heterogeneous vehicular networks using CGAN. The main idea involves partitioning a network into multiple virtual networks to cater to different service requirements. The authors suggest employing GANs to generate tailored network slices that match specific vehicular communication needs. This approach aims to optimize resource allocation, enhance communication quality, and enable efficient coexistence of various vehicular applications within the network. However, the authors trained the CGAN model using data from a simulated scenario because of the lack of data sets from real environments.

The integration of GAN and RL techniques to estimate channel coefficients is investigated in [30]. Similar to prior research, the aim is to create synthetic data through GAN and employ it to train RL algorithms, contributing to the refinement of the estimation process. By leveraging GAN's data generation capabilities and RL's adaptable learning, the proposed method enhances the precision of channel coefficient estimation. This underscores the potential of machine learning paradigms for wireless communication applications. However, it's important to note that the authors do not utilize a real experimental setup or environment. Instead, they generate the dataset using Gaussian distributions for GAN training, omitting the use of real data. Obtaining actual network data for these tasks poses significant challenges. First, network data may contain sensitive or private information of the users or organizations, which raises ethical and legal issues for sharing or publishing them. Second, network data may be scarce or outdated [41], especially for emerging or evolving network scenarios (e.g., 5G and beyond). Third, network data may be biased/incomplete [10], which limits the generalization and robustness of the network analysis models.

3.2.2. PCAP Generation

Neural networks offer a powerful solution to the challenges of obtaining and utilizing actual network data for analysis tasks. One of the primary advantages of employing neural networks in this context is their ability to learn complex patterns and relationships from existing network traces, such as PCAPs (Packet Capture files), and augment existing datasets with synthetic packet traces. In this manner, network traces are augmented without user privacy concerns – in addition to the possibility of teaching the model how and/or where to generate anomalies.

Several papers have proposed GAN-based methods for generating different aspects of network traffic, such as packet headers, packet payloads, packet lengths, packet inter-arrival times, flow features, and control-plane messages. To our knowledge, Ring et al.[42] was the first to utilize GAN models to generate network traffic data. They leveraged two GAN variants, WGAN [3] and WGAN-GP [17]. PcapGAN [12] proposes a method for generating realistic PCAP files that preserve the style and structure of real PCAP files. The method uses a style-based GAN architecture that can control the style of the generated packets at different levels of abstraction. The method can also generate packets with protocols, such as TCP, UDP, ICMP, ARP, and DNS. Shahid et al. [44] combined an autoencoder with WGAN and WGAN-GP to generate IoT traffic so it could detect rare attacks. The generated data had data distribution characteristics similar to real data. However, the model had little improvement in detecting rare attacks due to dataset imbalance. SIP-GAN [32] introduces a method for generating realistic SIP (Session Initiation Protocol) traffic that can be used for testing VoIP (Voice over IP) systems. The method uses a conditional GAN architecture that can generate SIP messages with different types, such as INVITE, ACK, BYE, CANCEL, and OPTIONS. The method can also generate SIP messages with different attributes, such as call duration, caller ID, callee ID, and session description. Similarly, proposes a new and generalized two-pass approach to evaluating the quality of samples produced by the generator to produce a filtered, higher-quality output data set.

NetShare [51] and Han et al. [20] leverage Wassertein GAN variations. The first generates synthetic IP header traces that can be used for network analysis and anomaly detection. The method uses a Wasserstein GAN with gradient penalty (WGAN-GP) architecture to generate IP headers with realistic features, such as source IP address, destination IP address, protocol type, port number, and packet length. Also, it preserves the statistical properties of the original traces, such as mean, variance, auto-correlation, and cross-correlation. Han et al. [20] also leveraged synthetic packet payloads that can be used for smart cybersecurity applications. The method uses a flow-based WGAN architecture to generate packet payloads with different lengths and contents. The method also uses an attention model based on byte embedding that can learn the correlations between different bytes in the packet payloads. Similarly, Lee et al. [27] improved the identification accuracy of sparse assaults in network intrusion detection and developed a data augmentation strategy based on the WGAN-GP model.

Soper et al. [45] introduced a two-pass approach that consists of two steps: (i) generating synthetic network traffic data sets using existing methods; (ii) refining the synthetic network traffic data sets using an error correction model based on recurrent neural networks (RNNs). The method can reduce errors between real and synthetic network traffic data sets regarding statistical properties and classification performance. On the other hand, Buhler et al. [8] generates live network traffic from code repositories. The method uses a code analysis tool to extract network-related information from code files, such as protocol type, port number, packet content, and packet frequency. The method can also use a code execution engine to run the code files and generate live network traffic based on the extracted information. Meng et al. [33] uses a Markov chain model to capture the temporal and spatial patterns of control-plane traffic, such as the state transitions, the inter-arrival times, and the location updates. The method can also use a GAN model to generate synthetic control-plane traffic based on the Markov chain model. Holland et al. [22] introduce nPrint, a tool that generates a unified packet representation amenable to representation learning and model training. They also integrate nPrint with automated machine learning (AutoML), resulting in nPrintML. This public system eliminates feature extraction and model tuning for traffic analysis tasks. The paper evaluates nPrintML on eight separate traffic analysis tasks and shows that it can achieve comparable or better performance than state-of-the-art methods with minimal human intervention. More recently, NetDiffusion [25] took a step forward and treated packets as images. It leverages the nPrint [22] the fixed-length standardized packet representation to transform packets into images that can be easily manipulated, considering the advent of generative models with a customized diffusion model.

This section delved into the fundamental concepts of GANs, providing essential insights into RL, and discussing the generation of PCAP files. Building upon this foundation, Section 3.3 complements the discussion by integrating these concepts with network programmability, examining how the principles outlined in this section apply to and enhance the domain of programmable networks.

3.3. In-band Network Telemetry and Programmable Data Planes

With the emergence of Software Defined Networking [31], the initial advances towards network programmability were taken [21]. The separation of data and control planes has brought about increased flexibility and new avenues of research in computer networks.

In this context, the OpenFlow protocol pioneered the provision of a network abstraction layer to the control element (Controller), thereby enabling the configuration and manipulation of the network's data plane through software programming. However, this model lacked sufficient flexibility for adding new headers and defining new actions after flow matching due to limitations imposed by the rigidity in the intelligence embedded in the pipeline of ASIC processors [14].

Traditionally, the data plane has taken on the role of processing packets according to the logic prescribed by the control plane. However, the game changes with the introduction of data plane programmability, which facilitates the incorporation of intelligence during packet processing at the hardware's most proximate level without the necessity for control plane intervention. Noteworthy examples of languages tailored for programming the data plane in this context include P4 and NPL. Among these, P4 is a language that has gained wide acceptance within scientific and industrial communities.

Beyond the evident flexibility, the response time of the data plane resides in the order of nanoseconds, while the control plane operates on a scale of seconds or milliseconds [21]. P4 code exhibits versatility across various architectures, each characterized by unique match-action pipeline stages and packet processing attributes. These architectural designs are fundamentally underpinned by an abstraction model, such as PISA (Protocol Independent Switch Architecture), that facilitates mapping instructions and hardware-specific features tailored to individual targets. The P4 language abstracts packet parsing and processing by providing a generalized forwarding model.

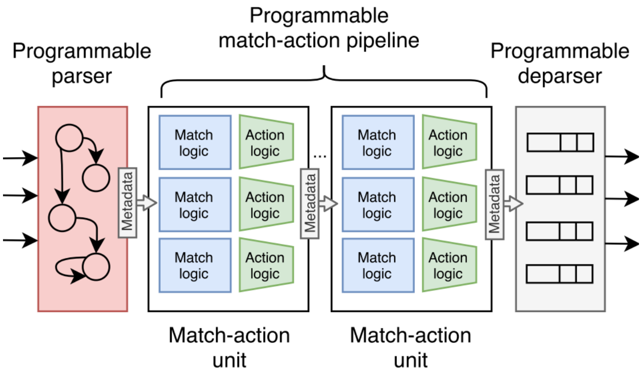


Figure 3.3: PISA abstraction model. Adapted from [21].

As illustrated in Figure 3.3, the PISA architecture consists of three fundamental components: a programmable parser, a programmable match-action pipeline, and a programmable deparser.

The programmable parser operates as a finite state machine that dictates the order of header extraction from incoming network packets. Once the programmable parser has exposed the headers and their associated fields, they can traverse multiple stages within the match-action pipeline. In this context, tables are defined to effectively manage metadata, adhering to the logic established by the programmer. For instance, in IPv4 routing, a table can be configured to perform matches against the destination address and execute actions such as i) decrementing the TTL field, ii) adjusting physical addresses, and iii) configuring the output port. The deparser represents the programmable component that specifies the packet's serialization, i.e., reassembly, for subsequent transmission.

In the context of programmable networking, the industry has recently adopted support for the P4 language. Xilinx has introduced the NetFPGA SUME, an advanced network card enabling P4 programming. Furthermore, industry leaders such as Intel and Broadcom have launched dedicated processors, namely the Tofino and Trident4, which are tailored explicitly for the programmable networking market.

A software-based switch version capable of executing P4 code is available for educational purposes. The Bmv2 (Behavioral Model Version 2) [1] is an integral component within the P4 ecosystem, designed as a tool for the study, testing, and analysis of solutions directly within the data plane.

Each equipment category in the P4 ecosystem, commonly called a "target", possesses a specific architecture. In the case of NetFPGA, the employed architecture is the

[1] `https://github.com/p4lang/behavioral-model.`

Simple Sume Architecture, whereas Tofino utilizes the Tofino Native Architecture (TNA), and the Bmv2 is rooted in the V1model architecture [21].

A comprehensive understanding of the architecture supported by the target is essential for the programmer, as it delineates the implementation particulars of the various stages. Furthermore, the availability of metadata for utilization depends on the specific architecture. These metadata can be accessed and integrated into network monitoring systems. For instance, in the V1model architecture, metadata about switch interface buffers can be retrieved and encapsulated within packets during forwarding. This technique is elaborated in the INT specification [39], designed to facilitate metadata collection within the data plane without necessitating control plane intervention or involvement.

These advancements in programmable data plane have enabled network devices to report the network's state autonomously, eliminating the need for direct control plane intervention [4]. In this scenario, packets incorporate telemetry instructions within their header fields, facilitating the fine-grained collection and recording of network data. The telemetry instructions are defined in the INT data plane specification [39].

Figure 3.4 illustrates the operation of INT within an arbitrary network. The network comprises four hosts, namely *H1, H2, H3*, and *H4*, along with four nodes equipped with P4 and INT support, denoted as *S1, S2, S3*, and *S4*. Each network node possesses a set of metadata, represented by orange (S1), magenta (S2), green (S3), and blue (S4) rectangles. This metadata contains information specific to each node, such as Node ID, Ingress Port, Egress Spec, Egress Port, Ingress Global Timestamp, Egress Global Timestamp, Enqueue Timestamp, Enqueue Queue Depth, Dequeue Timedelta, and Dequeue Queue Depth, as specified in the V1Model architecture.

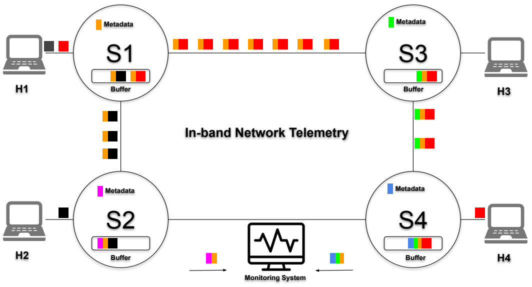


Figure 3.4: INT operation. INT metadata is appended on the packets in each hop. In the specific collection point, the monitoring system receives INT metadata.

Still, in Figure 3.4, two distinct flows are depicted: one represented by red packets and the other by black packets. The red flow is required to adhere to the prescribed network path *f1=H1, S1, S3, S4, H4*, while the black flow must traverse the designated path *f2=H1, S1, S2, H2*.

At each network hop along these paths, the data plane of the network devices employs telemetry instructions to facilitate the collection and inclusion of metadata within the packets as they traverse each node. This process is iteratively performed throughout the path, starting from the first node after the source and concluding at the last node before reaching the destination. Upon reaching the destination node, the metadata is extracted from the packet and relayed to the monitoring system. The original packet is then directed to its final destination.

One of the primary advantages of employing telemetry lies in the exceptional level of granularity it offers. Every packet traversing the network carries pertinent information directly to the monitoring system at the line rate. This level of granularity aligns with the perspective presented in [19], wherein it is recognized that a substantial volume of data can prove helpful for ML algorithms.

However, the dynamic nature of network traffic necessitates ongoing adjustments to machine learning models. RL is employed to tackle this issue, enabling agents to learn from variations in traffic characteristics through trial and error. A pivotal element in this context is the convergence time, which indicates when the intelligent agent has effectively learned a policy and started to implement it consistently. Regardless, abrupt changes in traffic dynamics can extend the time required for convergence or, in severe cases, cause the model to diverge. Using synthetically generated samples through GANs is advantageous for accelerating training and improving convergence, as RL agents can rapidly achieve the desired policy. Thus, Section 3.4 shows the challenges associated with synthetic data generation, contextualizing the operational environment and bringing the pertinent problems around it.

3.4. Synthetic Data Generation

This section outlines the dual-use cases of synthetic network data generation employed within the scope of this tutorial. Section 3.4.1 explains the characterization of an environment, providing a comprehensive description of the problem and the methodology adopted to address it. Furthermore, this section also expounds on the characterization of the dataset and the mechanism proposed for selecting the best model. On the other hand, Section 3.4.2 presents all the steps to generate PCAP traces with NetDiffusion and NetShare, the two main tools currently available for PCAP generation. First, the definition of the packet generation problem is presented. Finally, the choices of each work are presented, justifying the main coding/interpretation choices of network packets and how they deal with issues such as temporal (e.g., packet sequence) and spatial (e.g., packet fields) dependencies.

3.4.1. Generation of Telemetry Data

3.4.1.1. Problem definition

The environment where the dataset was collected in the present tutorial is depicted in Figure 3.5. In this setup, each infrastructure component is represented by an isolated virtual machine interlinked through a P4 programmable data plane network. The CDN was deployed to facilitate an MPEG-DASH service featuring live streaming of a soccer game and a playlist housing the ten most frequently accessed YouTube videos. Load management was executed using WAVE [2] [2], a versatile load generator that orchestrates instances of an application over time. A network architecture is instantiated and consists of three programmable switches where the INT telemetry is collected. Complementing this infrastructure, a Video Client is integrated to provide video metrics. At the top of this configuration, an RL agent operates within the network ecosystem, actively engag-

[2] https://github.com/ifpb/wave.

ing to enhance the user experience by orchestrating alterations to queue sizes within the switches. This RL agent's central goal resides in optimizing resource utilization across the switches, thereby elevating the overall efficiency of the network infrastructure.

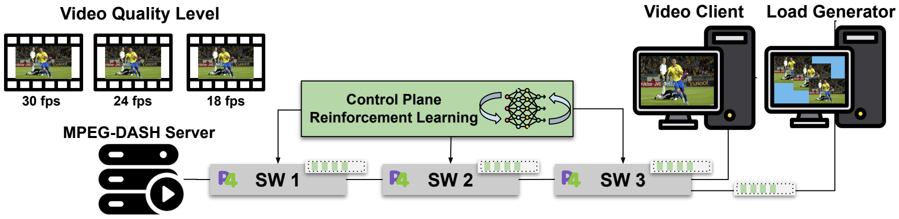


Figure 3.5: Real setup used for the evaluation composed by P4 BMv2 switches, a DASH server and video clients.

The RL model works as a data plane optimizer within this study, orchestrating the management of queue sizes across three distinct switches to enhance user experience. It is important to observe that cooperation from network operators to facilitate experiments of this nature is often challenging. In practice, the necessity arises to devise a mechanism for training the optimizer, or RL agent, without a real setup. In this case, the GAN trained on real data, serving as a simulator replacing the real setup, becomes pertinent. The present work is based on a real setup to train the RL agent while concurrently storing a dataset for employment in a GAN model. Moreover, after an on-the-fly training of the RL model, an offline training of the RL model is executed using synthetic data generated from the TimeGAN. This process facilitates a comparative assessment of the RL agent's generalization capacity across both scenarios: on-the-fly (real setup) and its simulated counterpart realized through utilizing the GAN generator.

However, a pertinent question arises: Given the availability of datasets sourced from operators, is the utilization of GAN to train Machine Learning models, including RL methodologies, necessary? Why not exclusively rely upon the original dataset to train such models?

Collecting a dataset constitutes a prerequisite for training a GAN model, necessitating data acquisition from operators or real setups. However, the inherent dataset may exhibit characteristics such as imbalance, inadequacy, missing or erroneous values, and static nature, preventing the feasibility of repeated experiments and statistical validation as demonstrated in [40]. In contrast, a GAN model, once trained, offers the capacity to generate balanced data devoid of inconsistencies or gaps. Moreover, it facilitates the generation of data volumes requisite for a diverse spectrum of experiments. Additionally, GAN empowers the training of models on different datasets, facilitating the acquisition of varied data distributions to cater to numerous scenarios. Besides, it is also important to highlight the time taken to train an RL model using a synthetic dataset compared to a real scenario. In the context of RL applications, GAN utilization assumes significance in enabling the introduction of diversified scenarios for the agent to learn from, thereby enhancing the agent's capacity to adapt to broader conditions.

Consequently, this tutorial delineates a methodology wherein the efficacy of an RL model is evaluated through its deployment in conjunction with data generated by a

TimeGAN model. The TimeGAN is trained to use telemetry data from video applications within the programmable data planes framework. The following exploration thereby offers insights into the viability and performance of RL when trained with synthetic data generated by TimeGAN.

3.4.1.2. Methodology

Broadly, the methodology adopted in this tutorial is encapsulated within Figure 3.6, and its implementation can be defined through a delineation into four distinct steps:

1. ***Creation of a Real Setup and Data Collection:*** In this preliminary phase, the establishment of a tangible real setup is undertaken, as presented in Sec. 3.4.1.1.
2. ***Dataset Recording and TimeGAN Model Training:*** An ensemble of pertinent features is stored within this stage. Among these features are switch telemetry metrics, notably encompassing switch queue sizes and packet arrival times. Simultaneously, video metrics, such as frames per second (FPS) and resolution, are captured. The ascertained dataset comprises two distinct scenarios, characterized by dissimilar queue sizes—32 and 64 packets—affording coverage of two operational settings.
3. ***Reinforcement Learning Training Utilizing Synthetic Data:*** After training the TimeGAN model, the subsequent step encompasses training the RL algorithm through synthetic data. Two discrete synthetic datasets were generated, each corresponding to the two distinct queue sizes—32 and 64 packets. The specific dataset input provided to the RL algorithm is contingent upon the actions undertaken by the agent. The agent is endowed with two distinct actions: either transitioning the queue size from 32 to 64 packets or vice versa. So, when the agent executes an action to effect a queue size alteration to 32 packets, the RL model is fed by synthetic data derived from the corresponding scenario. Conversely, when the agent executes a queue size modification to 64 packets, the RL model is furnished with synthetic data from the scenario predicated upon that specific queue size.
4. ***Perfomance Evaluation of RL Models:*** This phase is designed to stage a comprehensive evaluation of the two RL training paradigms: one executed within the confines of real setup and the other reliant on synthetic data engendered by the TimeGAN model. The principal aim here is not to ascertain optimal configuration parameters or enhanced performance for the RL model but to holistically gauge the degree of similarity exhibited by the outcomes in both scenarios. In essence, this evaluation seeks to determine whether the TimeGAN, as an alternative to real setups, can viably replicate the results obtained through RL training, thereby validating its potential to serve as a surrogate simulator without compromising the efficacy of RL training.

The continuous line illustrates an online component wherein the RL model undergoes real-time training (depicted in steps 1 and 3). Conversely, the GAN model undergoes training in an offline manner, as depicted in step 2. Concluding the sequence, step 4 embodies the comparative evaluation conducted between the RL model, trained dynamically in the real setup, and synthetic data engendered by the TimeGAN algorithm.

3.4.1.3. Dataset characterization

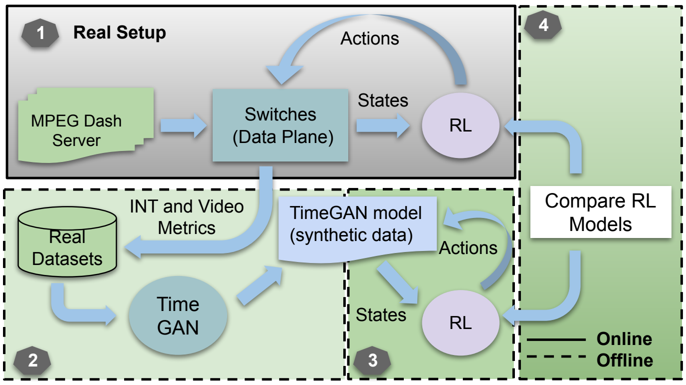


Figure 3.6: Proposed Methodology

Our dataset, centered on the Video Application within the Programmable Data Plane context, comprises two distinct categories of metrics: video metrics and network metrics, each residing in separate datasets. The video metrics encompass FPS, bitrate, and buffer size. In contrast, the network metrics consist of queue depth at packet queuing (Enq Qdepth), packet queuing duration in microseconds (Deq Timedelta), and queue depth at packet dequeuing (Deq Qdepth).

Our network configuration employed BMv2 software switches with P4 code. We employed an out-of-band approach involving specific ONT probes sent from the DASH server to the Video Client. This approach eliminates the need to modify data packets for telemetry metadata. Telemetry metadata (32 bytes) was collected at each network node. Two experiments were conducted to gather data from our real setup, as discussed in Section 3.4.1.1. The first experiment used switches configured with 32-packet buffer size, while the second utilized a 64-packet buffer size. Consequently, we obtained two datasets, each representing the real setup under differing buffer size conditions. These datasets were merged based on timestamps, with higher ‘Deq Timedelta’ samples filtered out to capture network behavior under high-load conditions.

An essential characteristic of the telemetry values within our Programmable Data Plane application is the non-stationary nature of all our features, as visually demonstrated in Figure 3.7. This data distribution poses significant challenges in comparing real data with synthetic data generated by GANs. We introduce a metric in Section 3.4.1.5 to address this complexity. This metric serves as a tool in our model selection process, aiding us in identifying the most suitable model capable of providing the closest feature representation to the real dataset.

3.4.1.4. TimeGAN Training

TimeGAN is a generative model crafted to generate synthetic time series data. It intends to replicate the statistical characteristics inherent to real-world time series datasets. Training a TimeGAN model encompasses a series of critical steps.

To commence, we initiated the process with data preprocessing, addressing concerns such as missing data imputation, outlier removal, and data normalization. The configuration of hyperparameter values was also crucial since it plays a pivotal role in

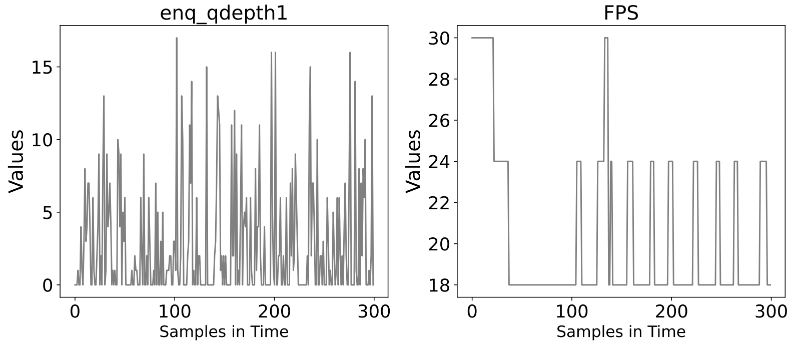


Figure 3.7: Real data plots for enq_qdepht1 and FPS features.

TimeGAN's training regimen. The determination of sequence sizes or windows was a significant consideration. These sizes are instrumental in the training, as they subdivide the dataset into multiple snapshots, each designed to capture the temporal behaviors within the time series.

Furthermore, other hyperparameters are variated, including sequence length, the number of hidden dimensions, and batch size. Hyperparameter tuning is a recognized challenge in machine learning training, with various strategies available, such as Grid Search. In our tutorial, we adopted an empirical approach, wherein we iteratively adjusted hyperparameter values based on prior training experiences and our insights into the parameters that exerted the most significant influence on the training process. After completing the training processes, we learned that the loss values across the training iterations exhibited a degree of similarity. Consequently, we recognized the need to develop a model selection methodology, the details of which are expounded upon in Section 3.4.1.5.

3.4.1.5. Models Selection

Addressing the challenge inherent to GAN, specifically the evaluation of synthetic data generated by these networks, is important. In [7] is underlined the absence of a consensus regarding the appropriate means to assess the distributions produced by GAN models. This issue is further compounded in the context of time series data, where the lack of recent publications is evident compared to more conventional GAN models. Besides, the characteristics of the dataset we collected are non-stationary. In contrast to stationary time series, non-stationary time series exhibit trends, seasonality, or other forms of systematic changes that make their statistical properties, such as the mean, variance, and autocovariance, vary with time. So, it is not possible to apply traditional tests such as Kullback–Leibler (KL) divergence. For example, the synthetic data obtained from certain trained models is constant values for some features, but real data has high variability. But, if a traditional test is applied, the results of the KL test are better for distributions with constant values, which is not desirable.

In our experimentation, we noted through empirical observations that certain GAN models we trained have superior synthetic data for distinct features. Consequently, we proposed a mechanism to select the most suitable model by analyzing each feature. To address this objective, we introduce a metric constructed through straightforward statistical measures, encompassing quartiles and medians. The central concept revolves around contrasting the distribution disparities exhibited by real and generated synthetic data. Consequently, a more favorable model for a given characteristic is indicated by a diminished dissimilarity in data distribution between these two sources. The calculation of this metric is depicted by Equation 1. To initiate this calculation, it is imperative to determine the discrepancy magnitude between the third and first quartiles for both datasets: real (X) and synthetic (y). This calculation furnishes the size of data dispersion divergence, facilitating a comparison.

Subsequently, another calculation involves assessing the variation between the sizes of these data dispersion, thereby gauging the difference between real and synthetic data. However, determining the disparity in dispersion magnitude alone might be insufficient, as the dispersion might have similar sizes while maintaining a positional offset. To address this, we calculate the difference in medians between the real and synthetic datasets and incorporate this disparity alongside the variance in dispersion sizes. Thus, a summation of these differences between quartiles and median for each feature is achieved and stored in metric (M) for all models. So, a superior model is characterized by the minimal value of M, indicative of the least dispersion in data distribution between real and synthetic data sources.

$$\begin{aligned}\mathbf{M} = & \sum_{n=1}^{n\_feats} |[Q_3(X_n) - Q_1(X_n)] - [Q_3(y_n) - Q_1(y_n)]| \\ & + |[med(X_n) - med(y_n)]|\end{aligned} \tag{1}$$

The algorithm presented in Algorithm 1 elucidates the procedure of selecting the optimal model for individual features. The algorithm takes as its inputs arrays of models about the two distinct scenarios expounded upon in Section 3.4.1.4, specifically those delineated by switch queue sizes of 32 and 64 packets.

Algorithm 1 Optimal Model Selection

```
Input: $models32, models64$
Output: $bestModel32, bestModel64$
    Initialisation :
$modelsMetrics32 \leftarrow calculateMetricForEachFeature(models32)$
$modelsMetrics64 \leftarrow calculateMetricForEachFeature(models64)$
for each $model32$, $model64$ in $modelsMetric32$ and $modelsMetric64$ do
    $models32Sum[model] \leftarrow getSumFeaturesMetrics(model, modelsMetrics32)$
    $models64Sum[model] \leftarrow getSumFeaturesMetrics(model, modelsMetrics64)$
end for
$bestModel32 \leftarrow min(models32Sum)$
$bestModel64 \leftarrow min(models64Sum)$
return $bestModel32, bestModel64$
```

As detailed earlier, the algorithm's initial operation involves the computation of the aforementioned metrics for each feature and all of the trained models. After this metric computation, an iteration wherein the algorithm determines the summation of metrics for each model is achieved. So, the minimum value of the sum of all features determines the best model ***considering the models trained in our study***.

The algorithm culminates by providing two arrays, each comprising the features of the best model. These arrays distinctly encapsulate the finest model under the respective queue size scenarios, addressing the dual scenario of queue sizes - 32 and 64 packets.

An evaluation of the model selection outlined in this section is expounded upon in Section 3.5.1.5. The primary objective is to provide a comprehensive analysis of the synthetic data generated by TimeGAN and its applicability in training an RL agent through an offline approach, contrasting it with an agent trained in an online format.

3.4.2. Generation of Synthetic Network Traces

This section discusses the NetDiffusion [25] and NetShare [51] problem definition and methodology towards synthetic packet generation.

3.4.2.1. NetDiffusion: Problem Definition

The NetDiffusion's authors proposed a two-fold strategy that leverages stable diffusion techniques. First, they convert raw packet captures into image representations - i.e., with nPrint [22]. This process allows them to use powerful image-based techniques for further analysis. Then, they fine-tune a text-to-image diffusion model on these converted packet capture images to let the model learn the patterns and relationships within network traffic data. To ensure the generated packets resemble real network traffic, they employ controlled techniques and domain knowledge-based heuristics to maintain fidelity to real-world data and ensure semantic correctness.

3.4.2.2. NetDiffusion: Methodology

The aforementioned steps are broadly summarized in Figure 3.8. The approach is structured around three primary components:

1. Conversion of raw network traffic into image-based representations using nPrint.
2. Fine-tuning a Stable Diffusion model to enable controlled text-to-traffic generation with high distributional similarity to real-world network traffic.
3. Domain knowledge-based post-processing heuristics for detailed modification of generated network traffic to ensure high protocol rule compliance.

The authors represented the packets as images leveraging print representation and noticed network traffic data exhibits high dimensionality. For instance, fields are abundant between the IP and TCP headers alone (e.g., IP addresses, ports, sequence and acknowledgment numbers, flags, etc.). In summary, nPrint has a bit-level representation that takes accounting for all [3] potential header fields (even if not present in the original packet). However, there are some limitations. While this ensures a uniform input structure for ML

[3] "All" means all considered headers by the authors so far: IPv4, IPv6 (Fixed Header), TCP, UDP, ICMP, Payloads.

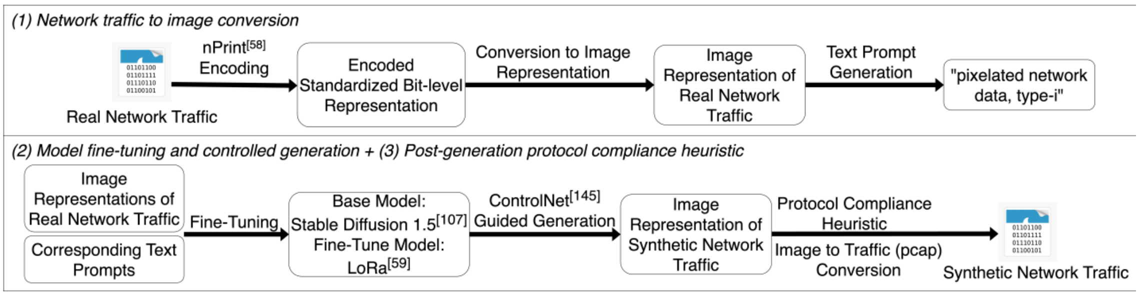

(1) Network traffic to image conversion
nPrint[58] Encoding
Real Network Traffic
Encoded Standardized Bit-level Representation
Conversion to Image Representation
Image Representation of Real Network Traffic
Text Prompt Generation
"pixelated network data, type-i"
(2) Model fine-tuning and controlled generation + (3) Post-generation protocol compliance heuristic
Image Representations of Real Network Traffic
Corresponding Text Prompts
Fine-Tuning
Base Model: Stable Diffusion 1.5[107] Fine-Tune Model: LoRa[59]
ControlNet[145] Guided Generation
Image Representation of Synthetic Network Traffic
Protocol Compliance Heuristic
Image to Traffic (pcap) Conversion
Synthetic Network Traffic

Figure 3.8: Generation Framework Overview. Source: the authors.

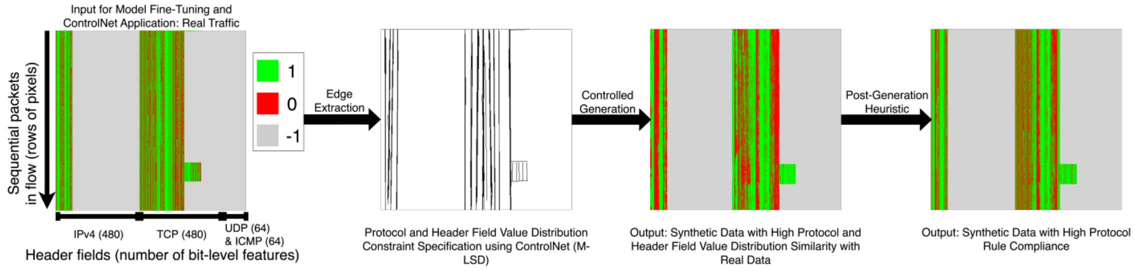

Input for Model Fine-Tuning and ControlNet Application: Real Traffic
Sequential packets in flow (rows of pixels)
🟩 1
🟥 0
⬜ -1
Edge Extraction
Controlled Generation
Post-Generation Heuristic
IPv4 (480)
TCP (480)
UDP (64) & ICMP (64)
Header fields (number of bit-level features)
Protocol and Header Field Value Distribution Constraint Specification using ControlNet (M-LSD)
Output: Synthetic Data with High Protocol and Header Field Value Distribution Similarity with Real Data
Output: Synthetic Data with High Protocol Rule Compliance

Figure 3.9: Synthetic Amazon network traffic outputs: (1) Using ControlNet, it detect regions present in the original traffic and ensure protocol and header field value distribution conformance by generating within specified regions. (2) Applying a post-generation heuristic to refine field details for protocol conformance. Source: The authors.

models, the attribute count per packet often exceeds a thousand. This high dimensionality introduces computational bottlenecks for generative models.

To arrive at image representations of network traffic, the authors first encode PCAPs using nPrint, which converts network traffic into standardized bits where each bit corresponds to a packet header field bit, as shown in Figure 3.8. This binary representation is simple yet effective, where the presence or absence of a bit in the packet header is denoted as 1 or 0, respectively, and a missing header bit is represented as -1. This encoding scheme ensures a standardized representation irrespective of the protocol in use. The payload content is not encoded since it is often encrypted. However, the size of the packet payloads can be inferred from other encoded header fields, such as the IP Total Length fields.

A sequence of PCAPs is converted into a matrix, which is then interpreted as an image. The colors green, red, and gray represent a set bit (1), an unset bit (0), and a vacant bit (-1), respectively. This color-coding schema provides a visually intuitive representation of the network traffic. Due to the limitations in the generative models' capability to handle very high dimensional data, they grouped every 1024 packets. While necessary for the current scope, this constraint could be revisited in future work to accommodate larger groups of packets. Through this process, any network traffic in PCAP format is transformed into an image with dimensions of 1088 pixels in width and 1024 pixels in height, with each row of pixels representing a packet in the network traffic flow as shown in Figure 3.9.

After transforming the packets into their corresponding image representations, the generative model is fine-tuned, specifically a diffusion model, to produce synthetic network traffic. While the out-of-the-box stable diffusion model is undeniably potent, they cannot directly use it to synthesize network traffic because it is designed to cover a broad spectrum of patterns and intricacies, causing it to lack the depth needed in specific generation tasks. For instance, a `Netflix Network Traffic` prompt might yield a generic image like a highway scene within a Netflix player. This is particularly evident when the textual description provided has multiple potential visual interpretations, causing the model to produce images that may be blurry or off-target. The authors addressed these limitations by fine-tuning Stable Diffusion on specific network datasets. They build upon Stable Diffusion 1.5 [43] and fine-tune this model on our specific network datasets as shown in Figure 3.8. To facilitate this fine-tuning, they employ Low-Rank Adaptation (LoRa) [23], a training technique tailored to fine-tune diffusion models, particularly in text-to-image diffusion models, swiftly. Its crux lies in enabling the diffusion model to learn new concepts or styles effectively while maintaining a manageable model file size. With LoRa, the resultant models are compact, balancing file size and training capability, while adpating to new data. For each image, they craft a unique encoded text prompt (e.g., `pixelated network data, type-0`) for Netflix traffic that succinctly describes its class type. The choice of their encoded prompt, though seemingly simplistic, achieves two main objectives. It offers a specific vocabulary that reduces ambiguity and ensures the model hones in on the nuances of the traffic. Subsequently, image-text pairs are fed into the fine-tuning process, where the base stable diffusion model, augmented with LoRa, learns to generate network traffic images conditioned on our prompts.

Upon fine-tuning the generation model, the next phase involves generating the desired class of synthetic network traffic. This is achieved by supplying the appropriate text prompts to the diffusion models to produce image representations of the traffic. Diffusion models operate by simulating a reverse process from a simple noise distribution to the data distribution, which enables them to capture and replicate the intricate patterns inherent in real-world data. The noise is progressively reduced over several steps, allowing the model to gradually refine the generated image until it closely resembles genuine network traffic patterns. An example of synthetic network traffic is shown in the image representation for Amazon traffic in Figure 3.9. This prompt-based generation process facilitates the creation of a synthetic nPrint-encoded network traffic dataset tailored to specific class distribution requirements. For instance, to curate a dataset with a certain class distribution and size, one would provide the corresponding quantity of text prompts per class and activate the generation process accordingly. Certain constraints are introduced during the generation process to ensure the generated traffic closely aligns with the prevalent protocol and header field value distributions observed in real traffic. If, for instance, the actual Amazon network traffic primarily consists of TCP packets, the generation process should prioritize populating header fields associated with TCP packets.

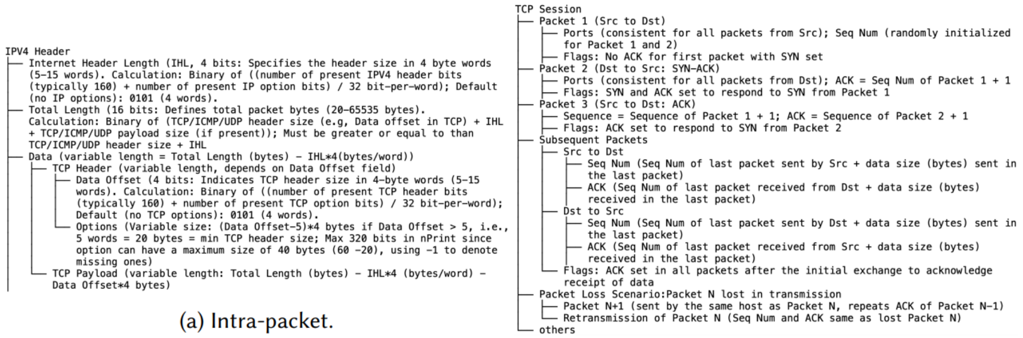
Leveraging the controllable nature of diffusion models, they incorporate ControlNet [55] into the generation process. ControlNet is a commonly used neural network architecture that adds spatial conditioning controls to large, pre-trained text-to-image diffusion models. It capitalizes on the robust encoding layers of these models, which are pre-trained with vast datasets, to learn a diverse set of conditional controls. In their specific use case of ControlNet, they leveraged M-LSD straight-line detection to find the boundaries between fields that are supposed to be populated and those that are not, as shown in Figure 3.9. Other edge detection methods, such as Canny edge detection, produce similar results. Such line or edge detection methods are effective because they align with the inherent columnar consistency present in packet traces.

Utilizing the controlled diffusion model, the generated encoded network traffic resembles the protocol and header field value distributions inherent in real-world data, capturing every feature observed in real traffic. Despite the guidance provided by ControlNet during the generation process, some network analysis and testing tasks often require raw network traffic. Converting the synthetic encoded traffic back to raw formats, such as PCAP, is not straightforward. This complexity arises from the multitude of detailed transport and network layer protocol rules at both inter and intra-packet levels. Properly formatted traffic must strictly adhere to these rules. While transport layer rules emphasize end-to-end communication and reliability, network layer protocols focus on packet routing and address assignment. Combined, these rules can be broadly divided into two categories:

1. *Inter-Packet Rules:* These rules dictate the relationships and sequencing between header fields within multiple packets in a network flow. For instance, packets need to be sequenced properly in a typical TCP connection, starting with the handshake process involving SYN and SYN-ACK flags. Data transfer integrity is ensured by aligning sequence numbers and acknowledgment numbers. Misalignment or incorrect sequencing can disrupt the connection or data transfer process.
2. *Intra-Packet Rules:* These pertain to the structure and contents within individual packets. For example, many protocol headers have a checksum field computed based on the packet's contents to detect errors during transmission. The checksum must be consistent with the packet's payload. Additionally, certain fields within a packet, such as port numbers in TCP and UDP headers, must adhere to specific formatting and value constraints to ensure the packet's validity and proper routing.

To maximize the encoded synthetic network traffic's compliance with transport and network layer protocol rules, they first discern a subset of critical header fields that mandate strict adherence to their formatting rules, e.g., sequence and acknowledgment numbers. In contrast, some fields can accommodate a degree of flexibility without compromising the integrity of the network traffic, such as TCP window size or TTL With the critical fields identified, they develop a systematic way to calculate their correct values based on other generated fields. This is achieved by constructing two dependency trees — intra-packet header field dependencies and inter-packet dependencies. These trees are built upon domain knowledge and are sourced from standard network protocol documentation [6]. They present example protocol rules and the associated dependency trees for TCP protocol in Figure 3.10. More comprehensive and detailed dependency trees can be found in the open-sourced repository. [4] Given a generated encoded network traffic, they begin the correction process by traversing the trees in an automated, bottom-up fashion.

[4] https://github.com/noise-lab/NetDiffusion_Generator



```
IPV4 Header
├── Internet Header Length (IHL, 4 bits: Specifies the header size in 4 byte words
│   (5-15 words). Calculation: Binary of ((number of present IPV4 header bits
│   (typically 160) + number of present IP option bits) / 32 bit-per-word); Default
│   (no IP options): 0101 (4 words).
├── Total Length (16 bits: Defines total packet bytes (20-65535 bytes).
│   Calculation: Binary of (TCP/ICMP/UDP header size (e.g, Data offset in TCP) + IHL
│   + TCP/ICMP/UDP payload size (if present)); Must be greater or equal to than
│   TCP/ICMP/UDP header size + IHL
├── Data (variable length = Total Length (bytes) - IHL*4(bytes/word))
│   ├── TCP Header (variable length, depends on Data Offset field)
│   │   ├── Data Offset (4 bits: Indicates TCP header size in 4-byte words (5-15
│   │   │   words). Calculation: Binary of ((number of present TCP header bits
│   │   │   (typically 160) + number of present TCP option bits) / 32 bit-per-word);
│   │   │   Default (no TCP options): 0101 (4 words).
│   │   └── Options (Variable size: (Data Offset-5)*4 bytes if Data Offset > 5, i.e.,
│   │       5 words = 20 bytes = min TCP header size; Max 320 bits in nPrint since
│   │       option can have a maximum size of 40 bytes (60 -20), using -1 to denote
│   │       missing ones)
│   └── TCP Payload (variable length: Total Length (bytes) - IHL*4 (bytes/word) -
│       Data Offset*4 bytes)
```

(a) Intra-packet.

```
TCP Session
├── Packet 1 (Src to Dst)
│   ├── Ports (consistent for all packets from Src); Seq Num (randomly initialized
│   │   for Packet 1 and 2)
│   ├── Flags: No ACK for first packet with SYN set
├── Packet 2 (Dst to Src: SYN-ACK)
│   ├── Ports (consistent for all packets from Dst); ACK = Seq Num of Packet 1 + 1
│   ├── Flags: SYN and ACK set to respond to SYN from Packet 1
├── Packet 3 (Src to Dst: ACK)
│   ├── Sequence = Sequence of Packet 1 + 1; ACK = Sequence of Packet 2 + 1
│   ├── Flags: ACK set to respond to SYN from Packet 2
├── Subsequent Packets
│   ├── Src to Dst
│   │   ├── Seq Num (Seq Num of last packet sent by Src + data size (bytes) sent in
│   │   │   the last packet)
│   │   ├── ACK (Seq Num of last packet received from Dst + data size (bytes)
│   │   │   received in the last packet)
│   ├── Dst to Src
│   │   ├── Seq Num (Seq Num of last packet sent by Dst + data size (bytes) sent in
│   │   │   the last packet)
│   │   ├── ACK (Seq Num of last packet received from Src + data size (bytes)
│   │   │   received in the last packet)
│   └── Flags: ACK set in all packets after the initial exchange to acknowledge
│       receipt of data
├── Packet Loss Scenario:Packet N lost in transmission
│   ├── Packet N+1 (sent by the same host as Packet N, repeats ACK of Packet N-1)
│   └── Retransmission of Packet N (Seq Num and ACK same as lost Packet N)
└── others
```

(b) Inter-packet.

Figure 3.10: Example TCP protocol rules/dependencies. Source: the authors.

Initially, they satisfy intra-packet dependencies, ensuring that individual packets are internally consistent. Subsequently, they address inter-packet dependencies, guaranteeing that the packets in a flow relate correctly. Certain fields necessitate uniformity across packets within the same network traffic trace—like IP addresses and ports. Others require specific initialization values, such as IP identification and TCP acknowledgment numbers. They employ a majority voting system to determine the most appropriate values for these fields by selecting the most frequently appearing value within the generated traffic. Another notable challenge is timestamp assignment for individual packets, given its intricate time series dependencies. This post-processing ensures that the encoded synthetic traffic can be seamlessly converted into raw network traffic formats (like PCAP) and subsequently be utilized for a range of non-ML tasks.

3.4.2.3. NetShare: Problem Definition

In this work, the authors explored the feasibility of ML-based synthetic packet-header (e.g., PCAP) and flow-header (e.g., Netflow) trace generation using Generative Adversarial Networks or GANs. They implement an end-to-end system, NetShare, and build a web service prototype [5] to help solve the following practical challenges in existing approaches:

- *Prior techniques*: (especially those based on tabular data GANs, which dominate the synthetic header generation literature) are unable to capture key correlations across header fields and header fields that have large ranges of values.
- *Scalability-fidelity tradeoff:* Existing techniques require significant GPU hours to train even moderately-sized traces (e.g., millions of records). Simple tabular GANs take a few hours to train but suffer in fidelity, while more complex time series GANs can take an order of magnitude more time.
- *Privacy-fidelity tradeoffs:* GANs are not well explored in the context of network header traces. Preliminary work suggests that differentially-private learning approaches are likely to yield poor fidelity for networking datasets [28].

[5] Available through `https://www.pcapshare.com`

In designing NetShare, the authors tackle these key challenges with a careful data-driven understanding of the limitations of canonical GAN-based approaches. NetShare combines the following key ideas to address the above issues:

- *Reformulation as flow time series generation:* Instead of treating header traces from each measurement epoch as an independent tabular dataset (i.e., rows of packets/flows with headers), they recast the problem for learning synthetic models for a merged flow-level trace across epochs. This reformulation allows us to capture intra and inter-epoch correlations natively.

- *Improving scalability via fine-tuning :* They identify opportunities to optimize learning time by using ideas of model fine-tuning and data-parallel learning from the ML literature [53]. Doing so naively may fail to capture dependencies across parallel instances, so they develop heuristics to preserve such correlations.

- *Practical privacy reformulations:* They adopt recent advances in differentially-private model training [54] and combine a small amount of public data with private data to improve privacy fidelity tradeoffs. To the best of our knowledge, this is the first application and empirical demonstration of header trace generation.

3.4.2.4. NetShare: Methodology

NetShare [6] is an end-to-end system leveraging a tabular representation distributed among different GANs with temporal and confinement dependencies. According to the authors, the existing approaches treat each packet or flow record independently and ignore intra- and inter-measurement epoch correlations. To systematically capture these cross-record correlations, they reformulate the header generation problem as a time series generation problem rather than a tabular generation problem, as shown in Figure 3.11. Specifically, they merge data from measurement epochs into one giant trace to capture inter-measurement epoch correlations. Given this giant trace, they split it into a set of flows. D^{flow} based on five-tuples to explicitly capture flow-level metrics (e.g., flow size/duration). Each sample in D^{flow} has a five-tuple as metadata and a record (or "measurement data") corresponding to a sequence of packets for PCAP data and flow records for NetFlow data. Specifically, for PCAP data, each sequence element (packet) includes a raw timestamp, packet size, and other IP header fields.

[6] The code is open-sourced at `https://github.com/netsharecmu/NetShare`.

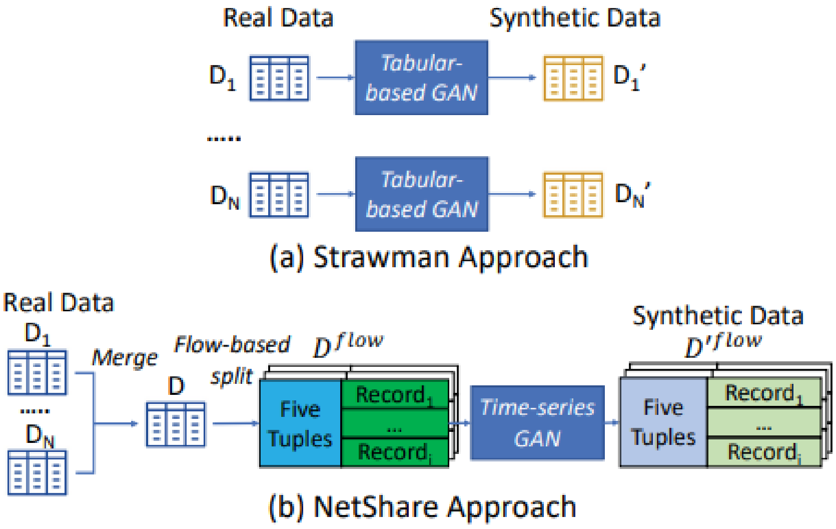


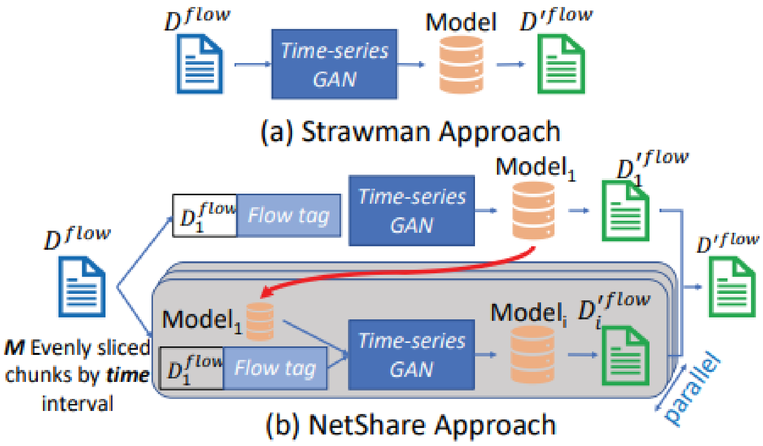
Figure 3.11: Multiple epochs D_i merged into a giant trace D, splitting the trace into flows D^{flow}, and using time series GAN. Source: the authors.

Reformulating the problem as a time-series generation brings much better header/temporal correlations but increases the total CPU time. One opportunity to improve scalability is via parallelism. However, naively dividing the giant trace into chunks and parallelized training across chunks poses two limitations. First, they incur the risk of losing correlations across chunks [7] e.g., flow size distribution for flows that span multiple chunks. Second, while the wall clock time decreases, the total CPU hours consumed remain unchanged.

These limitations can be avoided as shown in Fig 3.12. First, they borrow the idea of fine-tuning from the ML literature, i.e., they use a pre-trained model as a “warm start” to seed training for future models [54]. Specifically, they use the first chunk as the “seed” chunk to give a warm start, and subsequent chunks are fine-tuned using the model trained from the first chunk. This permits parallel training across chunks. One concern remains regarding the cross-chunk correlations; fine-tuning alone cannot preserve these. To this end, they append “flow tags” to each flow header to capture the inter-chunk correlation. Specifically, they annotate each flow header with a 0-1 flag denoting whether it starts in “this” chunk. They append a 0-1 vector after the flag with a length equal to the total number of chunks, with each bit indicating whether the flow header appears in that specific chunk. When splitting the giant trace D^{flow} flow into chunks, there are two natural choices: split by fixed time interval or by number of packets per chunk. Splitting by a fixed number of packets per chunk may impact differential privacy guarantees, as any single packet could change the final trained model in an unbounded way: removing any packet could change the packet assignment of all the following trunks. Thus, they split by fixed time intervals rather than a fixed number of packets. They left the choice of M (number of chunks) as a configurable tradeoff; a higher M would give fewer total CPU hours while increasing the learning complexity across chunks. In their case, $M = 10$ was

[7] These chunks are logically independent of the measurement epochs in the original dataset; chunks are merely a construct for parallelization training.

chosen for each dataset with 1 million records.



(a) Strawman Approach

(b) NetShare Approach

Figure 3.12: They split D^{flow} into M evenly time-spaced chunks with explicit "flow tags" to capture cross-chunk correlations. They use the first chunk as a pre-trained model for parallel training of later chunks. Source: the authors.

Prior attempts to train DP synthetic network data models using deep generative models have utilized DP-SGD, which modifies stochastic gradient descent (SGD) by clipping each gradient and adding Gaussian noise [1]. For a fixed amount of added noise, the more rounds of DP-SGD they run, the greater the cost in the privacy budget. In NetShare, they exploit the observation that one can reduce the number of rounds of DP-SGD needed to achieve a fixed fidelity level by pre-training NetShare on a related public dataset; then, they take the learned parameters from the public dataset and fine-tune them using DP-SGD over the private dataset. In doing so, they reduce the required number of iterations of DP-SGD. This insight has been explored in related work from the DP community [5, 26], but it has not been utilized in the networking domain to the best of our knowledge. They also use public data to improve our privacy-fidelity tradeoff due to our IP2Vec encoding. Specifically, they train our IP2Vec mapping on a public dataset with many port/protocol pairs, which helps us learn an embedding without affecting their DP budget.

Finally, they use a time series GAN to model this data. While autoregressive models [50] also use a time series approach, they are less effective for learning implicit distributions (e.g., flow length [28]) and achieve worse fidelity. Specifically, they build on an open-source tool called DoppelGANger [28]. Note, however, that natively using a time-series GAN like DoppelGANger would run into the same issues as the tabular GANs, as each flow or packet record will be a time-series record of length 1 and will miss the key cross-record effects. Furthermore, they will also encounter other challenges regarding encoding, scalability, and privacy. The baselines struggle to learn the distribution of fields with large support accurately. Hence, instead of training a GAN on the original data representation, they use domain knowledge to transform certain fields (especially those with large support) into a more tractable format for GANs. For fields with numerical semantics like packets/bytes per flow with a large support, they use log transformation, i.e., log(1+x), to effectively reduce the range. This simple yet effective technique helps NetShare better distribute large-support fields than baselines.

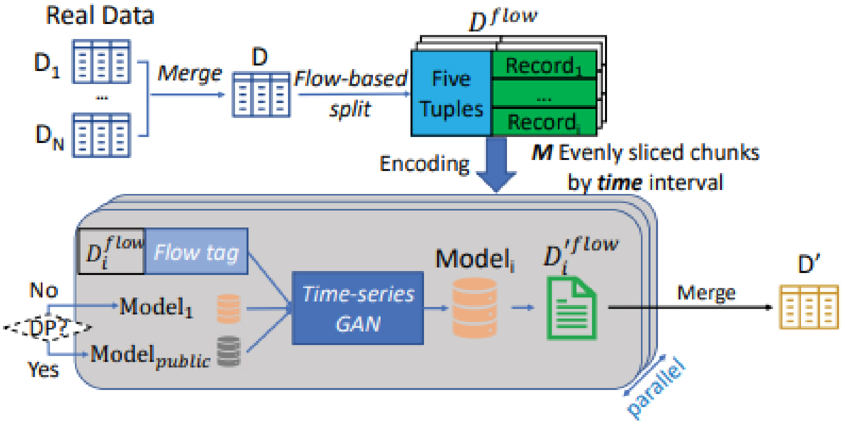


Figure 3.13: NetShare: end-to-end overview. Source: the authors.

3.4.2.5. NetShare: End-to-end view

Combining the key insights above, our end-to-end design is summarized in Figure 3.13. **Pre-processing:** Merge data from different measurement epochs D_i into one giant trace D with a flow-based split as D^{flow}. Encode header fields based on domain knowledge and fidelity-scalability-privacy tradeoffs.

Training: Evenly slice flow traces into M fixed-time chunks with explicit flow tags added. Train a time series GAN for each chunk; they use DoppelGANger [28]. If differentially-private (DP) is not required, use the model from the first chunk as the pre-trained model to improve scalability-fidelity tradeoff. If DP training is desired, use the model pre-trained on public data to fine-tune DP-SGD.

Post-processing: After generating $D_i'flow$, they map transformed fields back to their natural representations (e.g., map IP2Vec embeddings to `<port, protocol>` via nearest-neighbor search). Then, they generate derived fields (e.g., checksum) [8]. Finally, they convert to PCAP/NetFlow dataset by merging packets/NetFlow records according to the raw timestamp (for PCAP) or raw flow start time (for NetFlow).

3.5. Use Cases for Synthetic Data Application

This section presents the practical application of the two use cases outlined in Section 3.1. We aim to demonstrate the methodologies, code snippets, and tools for creating synthetic datasets. Specifically, Section 3.5.1 explores the generation of telemetry data for INT and DASH protocols, detailing the process and techniques involved. Conversely, Section 3.5.2.1 focuses on creating PCAP files, elucidating the procedure using the NetDiffusion Tool.

[8] They make an explicit design choice to exclude such derived fields, which are likely intractable to learn automatically. As such, they use a two-step generation mechanism: (1) use NetShare to generate the native fields (e.g., IP/port/timestamp) and (2) compute the checksum based on that to ensure the correctness of packets. Additionally, they did not consider the option field in the IP header, which is rarely used (and they do not observe the appearance of the IP option field in all three PCAP-related datasets).

3.5.1. Generation of Telemetry Data

This section guides training, generating, evaluating, and applying synthetic telemetry data for training an RL agent. It's important to note that this chapter does not cover the initial phase of our methodology, which involves data collection in a real-world environment. Our primary focus here is on the synthesis of data. The entire project for this tutorial can be accessed and cloned from the Github repository [9]. We have used in all of our scripts the YData Synthetic API[10], to conduct and perform the training and data generation.

The creation, training, and evaluation of synthetic data are heavily influenced by the number of models you aim to develop, each featuring a unique set of hyperparameters. To streamline this process, we provide a configuration file (**params.py**), which defines essential parameters for our workflow. In this tutorial, we introduce a variable called amount_of_models that influences the setting of hyper-parameter values, including:

- **seq_len (sequence length)**: This parameter delineates the temporal window's extent for each sequence during the training phase, meaning the count of time chunks (or lines) encapsulated within each sequence.
- **hidden_dim (hidden dimensions)**: It specifies the count of units or neurons present within each hidden layer of the network, thus shaping the model's capacity to learn and represent data complexities.
- **batch_size**: This hyperparameter defines the quantum of temporal sequences (or the number of data examples/lines) that are congregated into a singular batch for training, affecting both the computational efficiency and the learning dynamics of the model.
- **train_steps**: It denotes the aggregate count of training iterations, quantifying the extent of the model's exposure to the training data, influencing its learning trajectory and convergence behavior.

The variable amount_of_models is factored to set the upper limits for the i, j, and k variables in the scripts' nested triple-loop structure. The fatNum method in ModelUtility class is designed to compute these maximum values. For instance, with three models, the maximum limits for i, j, and k are set to 1, 1, and 3, respectively. With nine models, these limits are adjusted to 1, 3, and 3, respectively.

So, the workflow for synthetic data generation is systematically outlined across several sections. Section 3.5.1.1 describes the environment configuration. Section 3.5.1.2 details the data preprocessing steps, Section 3.5.1.3 focuses on the training process, Section 3.5.1.4 describes the synthetic data generation, and Section 3.5.1.5 examines the evaluation of the generated synthetic data. Each section is dedicated to a specific workflow phase, ensuring an understanding of the entire synthetic data generation process.

[9] https://github.com/thiagocaproni/tutorial_timegan

[10] https://github.com/ydataai/ydata-synthetic.

3.5.1.1. Dependencies and Environment Settings

Python 3.9.16 is necessary to execute the code. An environment manager like Anaconda, which includes the conda package manager, can be used to set up a dedicated environment with all the required dependencies. Installing Anaconda in your user directory ensures it does not conflict with the existing system Python installation. The dependencies are listed in the **environment.yml** file.

To create and configure the environment, execute the command provided in Code 3.1 within the repository directory, where **environment.yml** file is located. This will establish a separate environment and install the necessary dependencies. To begin using this environment, activate it by executing the command “conda activate”.

Code 3.1: Environment Settings

```
conda env create
conda activate ydata
```

3.5.1.2. Data preprocessing

The `DataPre` class (defined in **`preprocess_data.py`**) is a component in our data preprocessing pipeline, designed specifically for preparing time series data for synthetic data generation. In this part, we will delve into each method provided by this class and explain how they contribute to the preprocessing of the dataset. Before we start utilizing it, we need to import the necessary Python libraries that our class depends upon, as demonstrated in the script in the repository.

The `transformTimeStamp` method of `DataPre`, shown in Code 3.2, is used for standardizing the timestamp format across our datasets. This method converts timestamps from milliseconds to seconds and sets them as the DataFrame’s index, a common prerequisite for time series analysis.

Code 3.2: Transforming Timestamps

```
def transformTimeStamp(self, df):
    df[’timestamp’] = df[’timestamp’] / 1000
    df[’timestamp’] = df[’timestamp’].astype(int)
    df.set_index(’timestamp’, inplace=True)
```

In turn, Code 3.3 presents the `loadDataSet` method, which is responsible for loading and combining multiple data sets into a single DataFrame. This method ensures that data from different sources can be aligned and analyzed based on timestamps.

Code 3.3: Loading and Merging Datasets

```
def loadDataSet(self, path_int, path_dash):
    df_int = pd.read_csv(path_int, sep=’,’)
    df_dash = pd.read_csv(path_dash, sep=’;’)
    self.transformTimeStamp(df_dash)
    df_int = df_int.loc[df_int.groupby(’timestamp’)[’deq_timedelta1’].idxmax()]
    df_int.set_index(’timestamp’, inplace=True)
    self.dataset = pd.merge(df_int, df_dash, left_index=True, right_index=True).
    reset_index()
```

The hotEncode method (Code 3.4) applies one-hot encoding to categorical variables in the dataset, transforming them into a format that machine learning algorithms can more effectively utilize.

Code 3.4: One-Hot Encoding of Categorical Variables

```
def hotEncode(self):
    self.encoder = OneHotEncoder(handle_unknown='ignore')
    encoder_df = pd.DataFrame(self.encoder.fit_transform(self.dataset[['Resolution']]).toarray())
    self.dataset = self.dataset.join(encoder_df).copy()
    self.dataset.drop('Resolution', axis=1, inplace=True)
```

Finally, the preProcessData method orchestrates the overall preprocessing workflow, covering the imputation of missing values, encoding, and optional data shuffling.

Code 3.5: Preprocessing the Dataset

```
def preProcessData(self, num_cols, cat_cols, random):
    for i in num_cols:
        self.dataset[i].fillna(self.dataset[i].mean(), inplace=True)
    if len(cat_cols) > 0:
        self.hotEncode()
    self.processed_data = self.dataset[num_cols + cat_cols].copy()
    if random:
        idx = np.random.permutation(self.processed_data.index)
        self.processed_data = self.processed_data.reindex(idx)
```

The DataPre class serves as a tool for preparing time series data for synthetic data generation, encompassing a variety of preprocessing tasks to ensure data quality and compatibility for model training, which is discussed in Section 3.5.1.3.

3.5.1.3. TimeGAN Training

In practice, the scripts timegan32.py and timegan64.py perform data preprocessing and train a TimeGAN model for the synthetic generation of time series data. In this section, we show the code step by step, explaining each method's purpose and how they collectively form a pipeline for generating synthetic data. As in Section 3.5.1.2, we start by importing (see the script in the repository) the necessary Python modules and libraries and adjusting the system path to include our utility scripts.

The loadDp method, presented in Code 3.6, orchestrates the data loading and preprocessing, leveraging the DataPre class for various preprocessing tasks.

Code 3.6: Loading and preprocessing data

```
def loadDp(random, outliers):
    dp = DataPre()
    dp.loadDataSet(path_int='../../datasets/log_INT_TD-32_100.csv',
                   path_dash='../../datasets/dash_TD-32_100.csv')
    dp.preProcessData(params.num_cols, cat_cols=params.cat_cols, random=random)
    dp.removeSameValueAttributes()
    if not outliers:
        dp.removeOutliers()
    return dp
```

The `train` function, as shown in Code Listing 3.7, produces the setup and execution of the TimeGAN model's training process. This method utilizes the preprocessed data alongside the TimeGAN instance, which is constructed using the YData API, to train the model effectively.

Code 3.7: Training a model

```python
def train(dp, seq_len, n_seq, hidden_dim, noise_dim, dim, batch_size, model, train_steps):
    learning_rate = 5e-4
    gan_args = ModelParameters(batch_size=batch_size, lr=learning_rate, noise_dim=noise_dim, layers_dim=dim)
    processed_data = real_data_loading(dp.processed_data.values, seq_len=seq_len)
    synth = TimeGAN(model_parameters=gan_args, hidden_dim=hidden_dim, seq_len=seq_len, n_seq=n_seq, gamma=1)
    synth.train(processed_data, train_steps=train_steps)
    synth.save(model)
```

The scripts `timegan32.py` and `timegan64.py` initiate the data preprocessing and model training processes, iterating over various hyperparameter configurations to optimize model performance. In Code 3.8, we illustrate a portion common to both scripts, focusing on the model training procedure. This code part demonstrates a triple-nested loop structure, where each iteration level adjusts specific hyperparameters critical to the training dynamics. As previously discussed, this adjustment of hyperparameters is important for refining the model's ability to effectively generate realistic synthetic time series data.

Code 3.8: Executing the training process

```python
dp = loadDp(random=False, outliers=False)
iMax, jMax, kMax = ModelUtility.fatNum(params.amount_of_models)
print("Number of models", params.amount_of_models, ’iMax:’, iMax, ’jMax:’, jMax, ’kMax:’, kMax)
print("Num GPUs Available: ", len(tf.config.list_physical_devices(’GPU’)))
try:
    with tf.device(’/device:GPU:0’):
        for i in range(iMax):
            for j in range(jMax):
                for k in range(kMax):
                    train(dp, ...)
except RuntimeError as e:
    print(e)
```

So, this Section explored the complete loading, preprocessing, and training of a TimeGAN model for synthetic time series data generation. The workflow cohesively integrates data handling and machine learning training, preparing the data and training the model with varying configurations to generate high-quality synthetic time series.

3.5.1.4. Telemetry Data Generation

This section outlines the process for creating synthetic time series data with TimeGAN. It includes steps for loading and preprocessing real datasets and performing statistical analysis. These activities form the basis for assessing the quality of the synthetic data, which will be further explored in the evaluation script detailed in Section 3.5.1.5. The synthetic data generation is carried out using the script `generate_synth_data.py` available in our Git repository.

The first thing to do is import the required libraries and modules (see the repository). Adjust the system path to include directories where additional modules like `DataPre` (discussed in Section 3.5.1.2) and `ModelUtility` are located.

The `loadSynthData` method, presented in Code 3.9, is designed to generate synthetic data based on previously trained TimeGAN models. It loads the models and generates a specified number of synthetic data windows. This method generates synthetic data based on models trained in the **timegan32.py** and **timegan64.py** scripts. The object returned when generating synthetic data is a three-dimensional object (`[i][j][k]`), where the variable `i` controls the number of windows generated and defined by the variable `seq_len` (explained in the training script). In turn, the variable `j` controls the index of the lines in each window, and the variable `k` controls the columns. In other words, several time windows are returned. Therefore, it is necessary to define the number of windows to generate in the `loadSynthData` method.

Code 3.9: Loading and creating synthetic data

```python
def loadSynthData(model32, model64, number_of_windows):
    # Load TimeGAN models for 32-bit and 64-bit data
    synth_32 = TimeGAN.load(model32)
    synth_data_32 = synth_32.sample(number_of_windows)
    synth_64 = TimeGAN.load(model64)
    synth_data_64 = synth_64.sample(number_of_windows)
    # Post-process synthetic data (e.g., binarizing a portion of the data, which was hot encoded)
    synth_data_32[:,:,13:17][synth_data_32[:,:,13:17] >= 0.5] = 1
    synth_data_32[:,:,13:17][synth_data_32[:,:,13:17] < 0.5] = 0
    synth_data_64[:,:,13:17][synth_data_64[:,:,13:17] >= 0.5] = 1
    synth_data_64[:,:,13:17][synth_data_64[:,:,13:17] < 0.5] = 0
    return synth_data_32, synth_data_64
```

The `loadRealData` function is responsible for loading and preprocessing the real datasets for subsequent comparison with synthetic data.

Code 3.10: Reading real data

```python
def loadRealData(dsint32, dsint64, dsdash32, dsdash64, num_cols, cat_cols, sample_size, randon, outliers):
    # Instantiate DataPre for each dataset and process
    dp32 = DataPre()
    dp32.loadDataSet(path_int=dsint32, path_dash=dsdash32)
    dp32.preProcessData(num_cols, cat_cols=cat_cols, random=randon)
    if not outliers:
        dp32.removeOutliers()
    real_data_32 = dp32.processed_data
    real_data_32 = real_data_32[0:sample_size].copy()
    real_data_32 = real_data_32.values
    # Repeat for 64-bit data
    dp64 = DataPre()
    dp64.loadDataSet(path_int=dsint64, path_dash=dsdash64)
    dp64.preProcessData(num_cols, cat_cols=cat_cols, random=randon)
    if not outliers:
        dp64.removeOutliers()
    real_data_64 = dp64.processed_data
    real_data_64 = real_data_64[0:sample_size].copy()
    real_data_64 = real_data_64.values
    return real_data_32, real_data_64
```

Calculate statistical metrics such as quartiles and median to assess the quality of the synthetic data, as demonstrated in Code 3.11. It is important to highlight that the `genStatisctics` method returns a dictionary for a given set of real and synthetic models comprising lists of column statistics (features). Each list contains statistical values in positions 0, 1, 2, and 3, corresponding to real data from 32-bit real, 32-bit synthetic, 64-bit real, and 64-bit synthetic, respectively, presenting percentiles and medians.

Code 3.11: Statistical methods

```
def getStatistics(data):
    # Compute and return percentiles and median
    median = np.median(data)
    percentile_25 = np.percentile(data, 25)
    percentile_75 = np.percentile(data, 75)
    return [percentile_25, median, percentile_75]

def genStatisctics(real_32, synth_32, real_64, synth_64, sample_size, num_cols):
    # Generate statistics for each column in the dataset
    statistics = {}
    for col in num_cols:
        statistics[col] = [getStatistics(real_32[:, col][:sample_size]),
                           getStatistics(synth_32[:, col][:sample_size]),
                           getStatistics(real_64[:, col][:sample_size]),
                           getStatistics(synth_64[:, col][:sample_size])]
    return statistics
```

So, the Code 3.12 is defined to convert the three-dimensional synthetic data objects into a two-dimensional dataset, flattening the windows into rows and columns.

Code 3.12: Creating dataset in two dimensions

```
def createDataSet(seq_len, data):
    lines = int(params.synth_sample_size / seq_len)
    dataset = np.zeros(lines * seq_len * params.merged_columns_len).reshape(lines *
    seq_len, params.merged_columns_len)
    for i in range(lines):
        for j in range(seq_len):
            dataset[i * seq_len + j] = data[i][j][:]
    return dataset
```

Code 3.13, especially in `get_allfeatures_metrics`, presents the metric generation based on Equation 1, which is derived from the discrepancy in the interquartile ranges between each feature of the real and synthetic data. Additionally, the variance in the medians of these features is incorporated into this computation. Consequently, an object named 'metrics' is instantiated, encapsulating the results of this equation across all models and their corresponding characteristics. Subsequently, these metrics are used in the evaluation script to determine the optimal model. It should be emphasized that the total of the metrics for all features of a given model, denoted by the variable M in the equation, is calculated in the `analyze_data_models.ipynb` script in the method `getFeaturesBestMetricsOfModels`. This method identifies the best and worst models from the trained models.

Code 3.13: Method of creating metrics

```
def getMetrics(statistic_data):
    # Calculate discrepancy metrics for 32 and 64-bit data
    metric32 = abs(statistic_data[0][2] - statistic_data[0][0]) - abs(statistic_data
    [1][2] - statistic_data[1][0]) + abs(statistic_data[0][1] - statistic_data[1][1])
    metric64 = abs(statistic_data[2][2] - statistic_data[2][0]) - abs(statistic_data
    [3][2] - statistic_data[3][0]) + abs(statistic_data[2][1] - statistic_data[3][1])
    return metric32, metric64

def get_allfeatures_metrics(metrics, model_index, statistic_data):
    # Aggregate metrics for each feature across all models
    for col in params.num_cols:
        metrics[0][model_index][col], metrics[1][model_index][col] = getMetrics(
    statistic_data[col])
```

As detailed in Section 3.5.1.3 on the training script, the generation script employs a triple-nested loop to process all trained models, generating data and computing statistical metrics for each one. To maintain clarity in this document, we have opted not to include this specific code segment. For those interested in examining the full code, it is accessible in our Git repository.

This section methodically illustrated the process of generating synthetic time series data with TimeGAN. It covers everything from preparing real datasets and generating synthetic data to conducting statistical analyses to evaluate the synthetic data's quality. Section 3.5.1.5 delves into the script responsible for identifying the optimal model from all the trained models.

3.5.1.5. Model Selection

This section assesses synthetic time series data generated by TimeGAN, focusing on identifying the best and worst-performing models through statistical analysis. The Python notebook script, `analyze_data_models.ipynb`, available in our repository, contains the code for model selection. The initial step involves importing necessary Python libraries and setting up the environment.

So, The `getFeaturesBestMetricsOfModels` method in the script (Code 3.14) evaluates synthetic data generated by different models to identify the best and worst-performing models based on their statistical metrics. The method aims to compare the synthetic data generated by different models with the real data, identifying which models produce the most and least similar models to the real data based on the aggregated metrics. It starts by calling `sumFeatureMetricsOfModels` to get the aggregate metrics for each model. These metrics represent the total summation of the synthetic data from the real data across all features. Two lists are returned: `sum32` and `sum64`, respectively, representing the summed metrics for 32-bit and 64-bit model data. The function then identifies the best models (i.e., those with the lowest sum value in their metrics, indicating the least deviation from the real data) for both 32-bit and 64-bit data. It uses `np.argmin` to find the index of the lowest value in `sum32` and `sum64`, indicating the best models. The `getModelNameByIndex` method (see our git) retrieves the model name based on this index, and the model's synthetic data is accessed using this name.

Similarly, it identifies the worst models (those with the highest sum value in their metrics, indicating the greatest deviation from the real data) for both 32-bit and 64-bit data. It uses `np.argmax` to find the highest value index in `sum32` and `sum64` and retrieves the corresponding model's name and data. Finally, the function returns the synthetic data for the best and worst models for 32-bit and 64-bit buffer sizes.

Code 3.14: Evaluating the best and word models

```
def sumFeatureMetricsOfModels(models, data_metrics):
    # Initialize arrays to store sum of metrics for each model
    sum32 = np.zeros(len(models))
    sum64 = np.zeros(len(models))
    # Sum the metrics for each feature in 32 and 64-bit models
    for i in range(len(models)):
        sum32[i] = sum(data_metrics[0,i,:])
        sum64[i] = sum(data_metrics[1,i,:])
    return sum32, sum64

def getFeaturesBestMetricsOfModels(models, metrics):
    sum32, sum64 = sumFeatureMetricsOfModels(models, metrics)
    index = np.argmin(sum32)
    model = getModelNameByIndex(index)
    best_32 = models.get(model)[0]
    index = np.argmin(sum64)
    model = getModelNameByIndex(index)
    best_64 = models.get(model)[0]
    index = np.argmax(sum32)
    model = getModelNameByIndex(index)
    worst_32 = models.get(model)[0]
    index = np.argmax(sum64)
    model = getModelNameByIndex(index)
    worst_64 = models.get(model)[0]
    return best_32, worst_32, best_64, worst_64
```

The subsequent portion of the script, following the method to select the best and worst models, is focused on producing graphical representations to facilitate the analysis of the models. This includes generating box plots, violin plots, cumulative distribution function (CDF) plots, and correlation matrices. These visualizations are integral in understanding the performance nuances of the models identified by the selection method, highlighting the differences in data distribution, variability, and correlation patterns between the synthetic data generated by the best and worst models. Additionally, the script enables the exportation of the best model's dataset in a CSV format through `createSaveDataSetModel` method. This dataset can then be imported into the scripts designed for training the RL agent.

Also, it is important to be clear that in [52], various methods are outlined for assessing the quality of data produced by a TimeGAN model. These evaluation techniques include visualization tools like t-SNE and PCA, alongside regression models to analyze samples generated by TimeGAN models. The visualization method, however, necessitates human involvement for subjective assessment, making it challenging to determine the most appropriate model definitively. Conversely, the regression-based evaluation strategy was found to be less effective in our context due to the non-stationary nature of our data, complicating the task of developing a reliable regressor.

Despite these obstacles, we implemented a basic RNN via the Keras framework for regression analysis to compare real and synthetic data. Utilizing the Adam optimizer and Mean Absolute Error (MAE) as the loss metric, the RNN was configured to forecast the final entry in a sequence from its preceding elements. Nonetheless, the outcomes for both the real and synthetic datasets were unsatisfactory. The code for this regression model is accessible in our Git repository.

Due to the aforementioned issues, our approach to model selection was primarily grounded in statistical analysis. Moreover, we pursued an alternative method for evaluating our synthetic data by developing an RL agent, trained using both real and synthetic datasets, to evaluate its performance in each scenario, as detailed in Section 3.5.1.6.

3.5.1.6. Applying synthetic data to an RL agent

This section will present the implementation details regarding the queue management problem in video streaming applications described in section 3.4.1.1. We will start by describing how to model the agent-environment interaction illustrated in Figure 3.2. We will briefly explain how the agent learns, and lastly, we will show how to use the agent and environment classes in an application context.

In a video streaming application scenario, the `Environment` class will receive the synthetic INT metadata, execute an action, and calculate a reward value for such an action (Code 3.15). We modeled the agent-environment interaction by considering the DASH Server video streaming chunk size as a time step. Thus, the agent should take action to increase or decrease the switch's queue size every 4 seconds.

Code 3.15: Take action method from the Environment class

```
1  def take_action(self, action, intState, dashState):
2      self.current_state = intState
3      self.received_intStates.append(intState)
4      self.received_dashStates.append(dashState)
5      self.actions_history.append(action)
6      self.fps_history.append(dashState[FOURTH_SECOND][FPS_INDEX])
7  
8      if action == BUFFER_SIZE_64:
9          self.buffer_size.append(64)
10     elif action == BUFFER_SIZE_32:
11         self.buffer_size.append(32)
12 
13     if len(self.received_intStates) == 2:
14         # state observed when the action was taken
15         self.current_state = self.received_intStates[0]
16         # resulting state after the taken action
17         self.next_state = self.received_intStates[-1]
18         # dash state when the action was taken
19         current_dash = self.received_dashStates[0]
20         # dash state after the taken action
21         next_dash = self.received_dashStates[-1]
22         self.calculate_reward(current_dash[FOURTH_SECOND][BUFFER_INDEX], next_dash[
       FOURTH_SECOND][BUFFER_INDEX], next_dash[FOURTH_SECOND][FPS_INDEX])
23         self.received_intStates = []
24         self.received_dashStates = []
25 
26     return self.current_state, self.next_state, self.reward, self.done, {}
```

In this sense, it should be noted that to induce the agent to learn how to orchestrate the switches queue sizes in a way to enhance the application Quality of Service (QoS), the reward score cannot be calculated immediately after the action taken in the current state, since the consequence of such an action would only be noticeable in the next state [11]. This phenomenon occurs because both TCP and the Adaptive Bitrate Streaming (ABR) algorithm have control mechanisms to alleviate the congestion on the network [46]. Hence, the reward calculation should be delayed until the next state observation. The implementation of this strategy can be observed in the conditional clause depicted in line 13 of Code 3.15.

In this context, the agent is rewarded for improving the video streaming QoS, characterized by FPS and the Local Buffer Occupancy (LBO). These metrics have an intrinsic correlation, such that as the LBO increases, there is a tendency for the FPS to increase as well. However, such a correlation is not always straightforward due to the complex dynamics between network congestion and video streaming. Therefore, to calculate a reward (R_{t+1}) for a specific action (A_t), we can first evaluate whether the LBO from the next state (LBO_{t+1}) improved in comparison to the LBO observed when the action was executed (LBO_t) [11].

Then, a reward score is assigned considering the effects of this action on both LBO and FPS (FPS_{t+1}) next states. In this sense, the agent should receive a maximum reward whenever the action taken leads to the maximization of LBO_{t+1}, and is penalized in an inversely proportional manner if the video streaming present stalls (Code 3.16). This approach ensures that the agent's reward is conditioned by its ability to optimize both LBO and FPS, reducing the video streaming tradeoffs in dynamic network conditions [11]. Hence, when developing your agent, you should carefully design the reward function to reflect your application's intrinsic characteristics. This is important since the reward function affects the agent's learning capacity as much as the model hyperparameters.

Code 3.16: Calculate reward method from the Environment class

```python
def calculate_reward(self, buffer_action_step, buffer_reward_step, fps_reward_step):
    # Check whether the LBO improved after the action taken
    if buffer_reward_step > buffer_action_step:
        if buffer_reward_step > 30:
            self.reward = 2
        # Check the next state’s FPS and assign a reward score accordingly
        elif buffer_reward_step < 30:
            if fps_reward_step == 30:
                self.reward = 1
            elif fps_reward_step == 24:
                self.reward = .5
            else:
                self.reward = .1
    # Checking whether the LBO retarded after the action taken
    if buffer_reward_step < buffer_action_step:
        if buffer_reward_step > 30:
            self.reward = 2
        # Check the next state’s FPS and assign a reward score accordingly
        elif buffer_reward_step < 30:
            if fps_reward_step == 30:
                self.reward = 1
            elif fps_reward_step == 24:
                self.reward = .5
            else:
                self.reward = -2
    # Append the calculated reward to the reward history
    self.reward_history.append(self.reward)
```

The *agent* in question is a Deep Q-Network (DQN) [36] designed with the Multi-Layer Perceptron (MLP) architecture, comprising an input layer with units for each INT feature, 2 hidden layers with 24 Rectified Linear Units (ReLU) each, and an output layer with 2 units - one for each action the agent could take (increase queue size to 64 or decrease it to 32). We trained an agent on the real setup throughout the video transmission to evaluate the feasibility of employing TimeGAN as a network traffic simulator for video applications. Afterward, we trained a second agent using only the synthetic data generated by TimeGAN. We followed the same methodology in both contexts. Still, in the latter one, instead of using the real setup, we fed the agent with the synthetic INT metadata corresponding to the action taken, thus simulating the real network behavior.

During the agent training, the actions to increase or decrease the switch's queue size are executed in accordance with the ε-greedy strategy, in which the agent starts exploring the environment by selecting random actions from the action space and exponentially decreases the probability to take such actions (ε) throughout the training steps to exploit the maximizing reward actions. Nonetheless, it should be noted that there must be a balance between exploration and exploitation to enable the agent to learn how to map actions to states continually. A well-adopted strategy is to start ε with 1.0 and exponentially decrease it to 0.01. Hence, we allow the agent to exploit its existing knowledge and explore new actions by intermittently choosing a random action from the action space [47]. Code 3.17 describes the exploration-exploitation strategy implementation.

Code 3.17: Epsilon-greedy policy method from the Agent class

```
def epsilon_greedy_policy(self, state):
    self.total_steps += 1
    # Is the probability of selecting a random action less than or equal to epsilon?
    if np.random.rand() <= self.epsilon:
        # If so, select a random action (exploration)
        return np.random.choice(self.num_actions)
    # Otherwise, select the action with the highest Q-value (exploitation)
    # Normalize the input state representation
    normalized_states = preprocessing.normalize(state)
    # Predict Q-values using the online network
    q = self.online_network.predict(normalized_states)
    # Return the action matching the highest Q-value
    return np.argmax(q, axis=1).squeeze()
```

In this context, we linearly decreased ε over a span of 250 time steps in order to favor exploration. Equation 2 formalizes the ε linear decay rate (D_{linear}) algorithm, which represents the ratio between the difference in starting (ε_s) and ending (ε_e) ε values, and the number of decay steps (T). Then, the probability of selecting random actions was exponentially reduced by a rate of 0.99 to a minimal value (0.01) in order to advance exploitation over exploration. Lines ranging from 9 to 15 of Code 3.18 describe the epsilon decay strategy implementation.

$$D_{linear} = (\varepsilon_s - \varepsilon_e)\,/\,T \tag{2}$$

Code 3.18: Memorize transition method from the Agent class

```
1  def memorize_transition(self, s, a, r, s_prime, not_done):
2      # Check whether the episode has terminated
3      if not_done == 0:
4          # If the episode has terminated, update episode-related attributes
5          self.episode_reward += r
6          self.episode_length += 1
7      # Otherwise, update the exploration rate based on training progress
8      else:
9          if self.train:
10             if self.episodes < self.epsilon_decay_steps:
11                 # Decrease epsilon linearly
12                 self.epsilon -= self.epsilon_decay
13             else:
14                 # Decrease epsilon exponentially
15                 self.epsilon *= self.epsilon_exponential_decay
16     # Update the episode counter
17     self.episodes += 1
18     # Update the reward history
19     self.rewards_history.append(self.episode_reward)
20     # Update the episode length
21     self.steps_per_episode.append(self.episode_length)
22     # Reset episode-specific attributes
23     self.episode_reward, self.episode_length = 0, 0
```

```
    # Store the transition in the experience replay buffer
    self.experience.append((s, a, r, s_prime, not_done))
```

The experiences resultant from agent-environment interactions are represented by a tuple of the current state, the action taken, the reward score, and the next state (`s`, `a`, `r`, and `s_prime` in Code 3.18). Once the agent parameters update condition is met, the DQN's online network is trained every time a new experience is stored in the experience replay memory buffer (lines ranging from 7 to 39 of Code 3.19). Conversely, the target network weights are updated at each τ step (line 49 from Code 3.19). For more information regarding the DQN training flow and the motivations supporting this methodology, please refer to the work by [36], since such content is beyond the scope of the current tutorial.

Code 3.19: Experience replay method from the Agent class

```
 1 def experience_replay(self):
 2     # If the experience replay does not meet the minimum requirement, no update occurs
 3     if self.minimum_experience_memory > len(self.experience):
 4         return
 5     # Sample a minibatch of transitions from the experience replay buffer
 6     minibatch = map(np.array, zip(*sample(self.experience, self.batch_size)))
 7     states, actions, rewards, next_states, not_done = minibatch
 8     # Normalize the states and next states
 9     normalized_states = preprocessing.normalize(states)
10     normalized_next_states = preprocessing.normalize(next_states)
11     # Compute Q-values for next states using the online network
12     next_q_values = self.online_network.predict_on_batch(normalized_next_states)
13     # Select the best actions for next states
14     best_actions = tf.argmax(next_q_values, axis=1)
15     # Compute target Q-values using the target network
16     next_q_values_target = self.target_network.predict_on_batch(normalized_next_states)
17     # Gather target Q-values from the best actions considering the batch size
18     target_q_values = tf.gather_nd(next_q_values_target,
19                                    tf.stack((self.idx, tf.cast(best_actions, tf.int32)),
20                                    axis=1))
21     # Computes target Q-values based on the Bellman equation
22     targets = rewards + not_done * self.gamma * target_q_values
23     # Record the mean Q-value for monitoring
24     self.q_values.append(np.mean(targets))
25     # Predict the Q-values for the sampled batch of transitions
26     q_values = self.online_network.predict_on_batch(normalized_states)
27     # Update the Q-values for the sampled actions
28     q_values[self.idx, actions] = targets
29     # Train the online network using the updated Q-values
30     loss = self.online_network.train_on_batch(x=normalized_states, y=q_values)
31     self.losses.append(loss)
32     # Update the target network weights periodically
33     if self.total_steps % self.tau == 0:
34         self.update_target()
```

With that, we covered the first two objectives of this section: describing how to model an agent-environment interaction following the MDP strategy (Figure 3.2) and explaining how such an agent can learn to make assertive decisions in the context of queue management for video streaming applications. Now, we will describe how to employ the aforementioned methods. Thus, this use case is based on 3 Python scripts: `receiveMetricsGan.py`, which acts as our main script, by reading the INT metadata and controlling the agent training pipeline; `environmentGan.py`, which contains the functions to take actions and calculate their respective reward scores; and `agent.py`, which defines all components of the DQN workflow.

Hence, to apply the synthetic data generated by TimeGAN to train an RL-based agent, you should follow the following steps:

- Read INT metadata regarding the 32-bit and 64-bit queue sizes from their respective CSV files. In this step, you should instantiate a thread for each function in order to be able to read the data from both files simultaneously.
- Join the data obtained in the previous step into a global dictionary, this will enable us to change the data source in accordance with the agent actions, thus simulating the real setup's network behavior.
- Send data to the RL Environment. In this step, you should first verify whether the agent has taken actions and retrieve a data frame from the global dictionary based on the action taken or the switch's initial queue size. Then, you will need to slice this data frame and get the columns related to INT and DASH, and subsequently convert them to numpy arrays. This method will enable us to use these features as input for the DQN. As mentioned throughout this section, the INT metadata will be used by the agent for action-taking, while the DASH metrics will be employed in the agent's reward calculation.

The aforementioned steps should be repeated throughout the video streaming, which is the length of the synthetic data files. All code mentioned throughout this section is well documented and will be available in a GitHub repository[11].

3.5.2. Generation of Synthetic Network Traces

3.5.2.1. NetDiffusion

In this section, we will present a step-by-step guide[12] for synthetic traffic generation with a third-party tool namely NetDiffusion [25]. NetDiffusion is a state-of-the-art open-source tool for the community to generate traffic considering different applications. It leverages a controlled variant of a Stable Diffusion [43] model to generate synthetic network traffic that boasts high fidelity and adheres to protocol specifications. First of all, access your machine with GPU support, performing a port-forwarding on port 7860 and clone the `kohya_ss_fork` and `sd_webui_fork`. Then, activate the Python 3.10.13 virtual environment in `kohya_ss_fork` folder as follows:

Code 3.20: NetDiffusion: Clone the mandatory git forks

```
# SSH to Linux server via designated port (see following for example)
ssh -L 7860:LocalHost:7860 username@server_address
# Clone the repository
git clone git@github.com:noise-lab/NetDiffusion_Generator.git
# Navigate to the project directory
cd NetDiffusion_Generator
# Access the `fine_tune` folder
cd fine_tune
# Remove the empty `kohya_ss_fork @ 8a39d4d` and clone it as follows
git clone https://github.com/Chasexj/kohya_ss_fork.git
```

[11] https://github.com/thiagocaproni/tutorial_timegan/tree/master/code

[12] NetDiffusion tutorial: https://hackmd.io/@goes-ariel/rkVlaxTR6

```
# Clone `sd-webui-fork @ 2533cf8` (also inside `fine_tune` folder)
git clone https://github.com/Chasexj/stable-diffusion-webui-fork.git
cd <NetDiffusion main folder>/fine_tune/kohya_ss_fork/
source venv/bin/activate
```

So, We must ensure the variable `LD_LIBRARY_PATH` is correctly exported. Otherwise, a mandatory library, namely *bitsandbytes*, for CUDA custom functions - in particular 8-bit optimizers, matrix multiplication (LLM.int8()), and 8 & 4-bit quantization functions, will not work. First, we need to ensure what CUDA version is necessary for the virtual Python environment. To do that, just follow the steps in code 3.21:

Code 3.21: NetDiffusion: Checking Python's virtual environment CUDA version needed

```
1  #### Checking CUDA version ####
2  (venv) (base) thiago@ifsuldeminas-Z390-M-GAMING:~/git_ariel/NetDiffusion_Generator/
       fine_tune/kohya_ss_fork$ python
3  Python 3.10.13 (main, Aug 25 2023, 13:20:03) [GCC 9.4.0] on linux
4  Type "help", "copyright", "credits" or "license" for more information.
5  >>> import torch
6  >>> torch.version.cuda
7  '11.8'
8  >>> exit()
9  (venv) (base) thiago@ifsuldeminas-Z390-M-GAMING:~/git_ariel/NetDiffusion_Generator/
       fine_tune/kohya_ss_fork$
10 #### Find libcudart.so library ####
11 (venv) (base) thiago@ifsuldeminas-Z390-M-GAMING:~/git_ariel/NetDiffusion_Generator/
       fine_tune/kohya_ss_fork$ find / -name libcudart.so 2>/dev/null
12 ... (omitted) ...
13 /home/thiago/anaconda3/envs/ydata/lib/libcudart.so
14 /home/thiago/anaconda3/pkgs/cuda-cudart-dev-12.1.105-0/lib/libcudart.so
15 /home/thiago/anaconda3/pkgs/cudatoolkit-11.3.1-h2bc3f7f_2/lib/libcudart.so
16 /home/thiago/anaconda3/pkgs/cudatoolkit-11.8.0-h6a678d5_0/lib/libcudart.so
17 ... (omitted) ...
18 (venv) (base) thiago@ifsuldeminas-Z390-M-GAMING:~/git_ariel/NetDiffusion_Generator/
       fine_tune/kohya_ss_fork$
19 #### Set the correct one (in this case, CUDA 11.8) ####
20 (venv) (base) thiago@ifsuldeminas-Z390-M-GAMING:~/git_ariel/NetDiffusion_Generator/
       fine_tune/kohya_ss_fork$ export LD_LIBRARY_PATH=/home/thiago/anaconda3/pkgs/
       cudatoolkit-11.8.0-h6a678d5_0/lib
```

At lines 3-9, we verify the demanded version of CUDA for the Python's environment. Following, lines 13-22 depict the output of the existing CUDA versions and its respective `libcudart.so` libraries. In this case, we correctly set and export `LD_LIBRARY_PATH` (line 26).

Code 3.22: NetDiffusion: Checking whether bitsandbytes is running correctly

```
(venv) (base) thiago@ifsuldeminas-Z390-M-GAMING:~/git_ariel/NetDiffusion_Generator/
    fine_tune/kohya_ss_fork$ python -m bitsandbytes
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++ BUG REPORT INFORMATION ++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

+++++++++++++++++++ ANACONDA CUDA PATHS ++++++++++++++++++++
/home/thiago/anaconda3/lib/libicudata.so
/home/thiago/anaconda3/envs/ydata/lib/python3.9/site-packages/torch/lib/
    libtorch_cuda_linalg.so
... (ommited) ...

++++++++++++++++++ /usr/local CUDA PATHS +++++++++++++++++++
/usr/local/lib/python3.8/dist-packages/torch/lib/libc10_cuda.so
/usr/local/lib/python3.8/dist-packages/torch/lib/libtorch_cuda_linalg.so
/usr/local/lib/python3.8/dist-packages/torch/lib/libtorch_cuda.so

```

```
+++++++++++++++ WORKING DIRECTORY CUDA PATHS +++++++++++++++
/home/thiago/git_ariel/NetDiffusion_Generator/fine_tune/kohya_ss_fork/venv/lib/python3.10/site-packages/onnxruntime/capi/libonnxruntime_providers_cuda.so
/home/thiago/git_ariel/NetDiffusion_Generator/fine_tune/kohya_ss_fork/venv/lib/python3.10/site-packages/torch/lib/libtorch_cuda_linalg.so
/home/thiago/git_ariel/NetDiffusion_Generator/fine_tune/kohya_ss_fork/venv/lib/python3.10/site-packages/torch/lib/libtorch_cuda.so
... (ommited) ...
++++++++++++++++++ LD_LIBRARY CUDA PATHS +++++++++++++++++++
 /home/thiago/anaconda3/pkgs/cudatoolkit-11.8.0-h6a678d5_0/lib CUDA PATHS
/home/thiago/anaconda3/pkgs/cudatoolkit-11.8.0-h6a678d5_0/lib/libcudart.so

++++++++++++++++++++++++++ OTHER +++++++++++++++++++++++++++
COMPILED_WITH_CUDA = True
COMPUTE_CAPABILITIES_PER_GPU = ['7.5']
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++ DEBUG INFO END ++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

Running a quick check that:
    + library is importable
    + CUDA function is callable

WARNING: Please be sure to sanitize sensible info from any such env vars!

SUCCESS!
Installation was successful!
```

In code block 3.22, we run the *bitsandbytes* module (line 2) and achieved the desired output (lines 66-74) where the import of the library is correct and the module is set. Otherwise, the output of this execution would indicate the need to export the LD_LIBRARY_PATH variable again.

To generate synthetic application data, we must convert application PCAPs/traces (e.g., Youtube, Skype) or the default provided Netflix traces into the nPrint [22] format, a standardized packet representation for Machine Learning model learning. To do that, we should store raw PCAPs used for fine-tuning into NetDiffusion_Generator/data/fine_tune_pcaps with the application/service labels as the filenames (e.g., netflix_01.pcap).

Code 3.23: NetDiffusion: Changing Caption

```
# For example, 'pixelated network data, type-0' refers to NetFlix pcap,
# Adjust the script based on fine-tuning task.
cd NetDiffusion_Generator/fine_tune && python3 caption_changing.py test_task/image/20_network
```

Code 3.24: NetDiffusion: Import Data

```
# Navigate to preprocessing dir
cd data_preprocessing/
# Run preprocessing conversions
python3 pcap_to_img.py
# Navigate to fine-tune dir and the khoya subdir for task creation
# (replace the number in 20_network with the average number of pcaps per traffic type used for fine-tuning)
cd ../fine_tune/kohya_ss_fork/model_training/
mkdir -p example_task/{image/20_network,log,model}
# Leverage Stable Diffusion WebUI for initial caption creation
cd ../../sd-webui-fork/stable-diffusion-webui/
# Lunch WebUI
bash webui.sh
```

The code section 3.24 summarizes to convert the nPrint data into an image representation (lines 2-5). Then, we create placeholder folders that will be used later. Also, we access a GUI (line 15) with the Python virtual environment and configure the fine-tuning parameters to train the model, as shown in the enumerated steps below. But first, we cannot forget the need to create a corresponding text file (e.g., “netflix_01.txt”) with a customized traffic label for the preprocessed data (e.g., “netflix_01.png”).

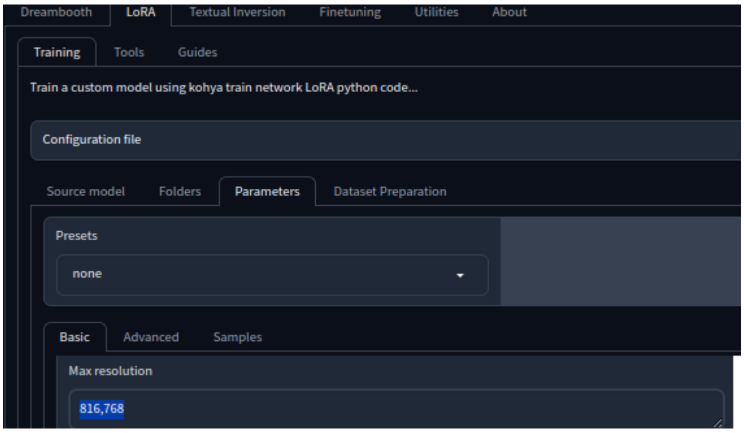
1. Open the WebUI via the ssh port on the preferred browser, example address: `http://localhost:7860/`
2. Under *Extras/Batch From Directory*, enter the absolute path for */NetDiffusion_Generator/data/preprocessed_fine_tune_imgs* and */NetDiffusion_Generator/fine_tune/kohya_ss_fork/model_training/test_task/image/20_network* as the *input/output* directories.
3. Under *Extras/batch_from_directory*, set the ‘scale to‘ parameter to `width = 816` and *height* = *768* for resource-friendly fine-tuning (adjust based on resource availability).
4. Enable the `caption` parameter under *extras/batch_from_directory* and click `generate`.
5. Terminate `webui.sh`

If you already run the setup.sh file inside `kohya_ss_fork` folder, do the following steps in code section 3.25 and check the installed requirements:

Code 3.25: NetDiffusion: Fine-tuning

```
# Navigate to fine-tuning directory
cd kohya_ss_fork
# Set up accelerate environment (gpu and fp16 recommended)
accelerate config
# Check the installed requirements with pip
pip list
# Fine-tune interface initialization
bash gui.sh
```

1. Open the fine-tuning interface via the ssh port on the preferred browser, example address: `http://localhost:7860/`
2. Under *LoRA/Training*, load the configuration file via the absolute path for */NetDiffusion_Generator/fine_tune/LoraLowVRAMSettings.json*
3. Under *LoRA/Training/Folders*, enter the absolute paths for */NetDiffusion_Generator/fine_tune/kohya_ss_fork/model_training/test_task/image*, */NetDiffusion_Generator/fine_tune/kohya_ss_fork/model_training/test_task/model*, and */NetDiffusion_Generator/fine_tune/kohya_ss_fork/model_training/test_task/log* for the Image/Output/Logging folders respectively, and adjust the model name if needed.
4. Under *LoRA/Training/Parameters/Basic*, adjust the Max Resolution (Figure 3.15) to match the resolution from data preprocessing, e.g., 816,768 (i.e., width, height).



Train a custom model using kohya train network LoRA python code...

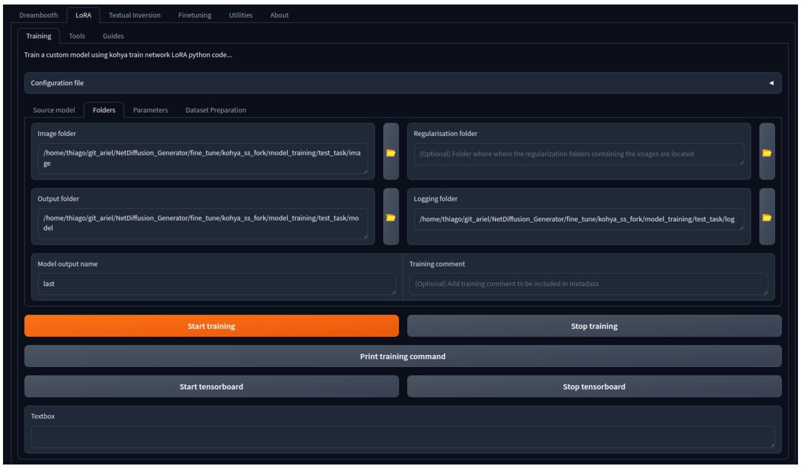
Presets: none

Max resolution: 816,768

Figure 3.15: NetDiffusion GUI: fine-tuning max resolution

5. Click on Start Training to begin the fine-tuning. Adjust the fine-tuning parameters as needed due to different generation tasks that may have different parameter requirements to yield better synthetic data quality.
6. After the "model saved" message (check the terminal), close the server (Ctrl + C).

Figure 3.14 shows the GUI to configure the model paths and parameters. By default, the `LoraLowVRAMSettings.json` file, contains Windows-based file paths for *Image/Output/Logging* files. However, we can manually configure the correct absolute paths and the Max Resolution in this file and load it again.



Train a custom model using kohya train network LoRA python code...

Image folder: /home/thiago/git_ariel/NetDiffusion_Generator/fine_tune/kohya_ss_fork/model_training/test_task/image

Regularisation folder: (Optional) Folder where where the regularization folders containing the images are located

Output folder: /home/thiago/git_ariel/NetDiffusion_Generator/fine_tune/kohya_ss_fork/model_training/test_task/model

Logging folder: /home/thiago/git_ariel/NetDiffusion_Generator/fine_tune/kohya_ss_fork/model_training/test_task/log

Model output name: last

Training comment: (Optional) Add training comment to be included in metadata

Textbox

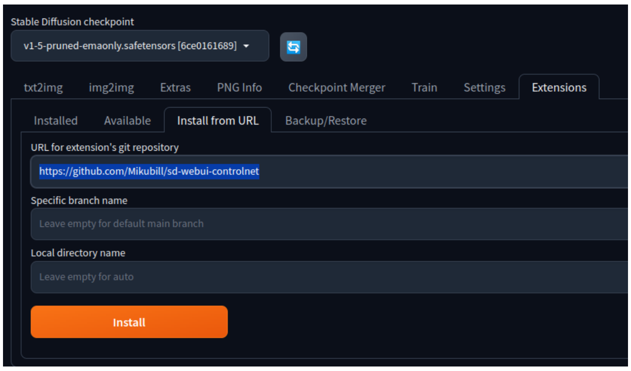
Figure 3.14: NetDiffusion GUI: fine-tuning folder paths.

In the previous steps we generated a file in the model folder provided (i.e., *finetune/kohya_ss_fork/model_training/test_task/model*). The default filename is `Addams.safetensors`. If not, replace "Addams" in the first step in the code block 3.26 by the correct name:

Code 3.26: NetDiffusion: Generation Step

```
# Copy the fine-tuned LoRA model (adjust path namings as needed) to Stable Diffusion
    WebUI
cp model_training/test_task/model/Addams.safetensors ../sd-webui-fork/stable-diffusion-
    webui/models/Lora/
# Navigate to the generation directory
cd ../sd-webui-fork/stable-diffusion-webui/
# Initialize Stable Diffusion WebUI
bash webui.sh
```

1. Open the WebUI via the ssh port on the preferred browser, example address: `http://localhost:7860/`
2. Install ControlNet extension for the WebUI and restart the WebUI: `https://github.com/Mikubill/sd-webui-controlnet` (see Figure 3.16). Go to *Extensions/* and place the URL in the appropriate field to install it.
3. To generate an image representation of a network trace, enter the corresponding caption prompt with the LoRA model extension under *txt2img* section. For example `pixelated network data, type-0 <lora:Addams:1>` for Netflix data.
4. Adjust the generation resolution to match the resolution from data preprocessing, e.g., `816,768`.
5. Adjust the seed to match the seed used in fine-tuning, default is `1234`.
6. Enable Hires.fix to scale to `1088, 1024`, upscaling the image (see Figure 3.18).
7. From training data, sample a real PCAP image (that belongs to the same category as the desired synthetic traffic) as input to the ControlNet interface, and set the Control Type (we recommend canny) - see Figure 3.17.
8. Click on `Generate` to complete the generation. Note that extensive adjustments on the generation and ControlNet parameters may be needed to yield the best generation result as the generation tasks and training data differ from each other.


Stable Diffusion checkpoint: v1-5-pruned-emaonly.safetensors [6ce0161689]

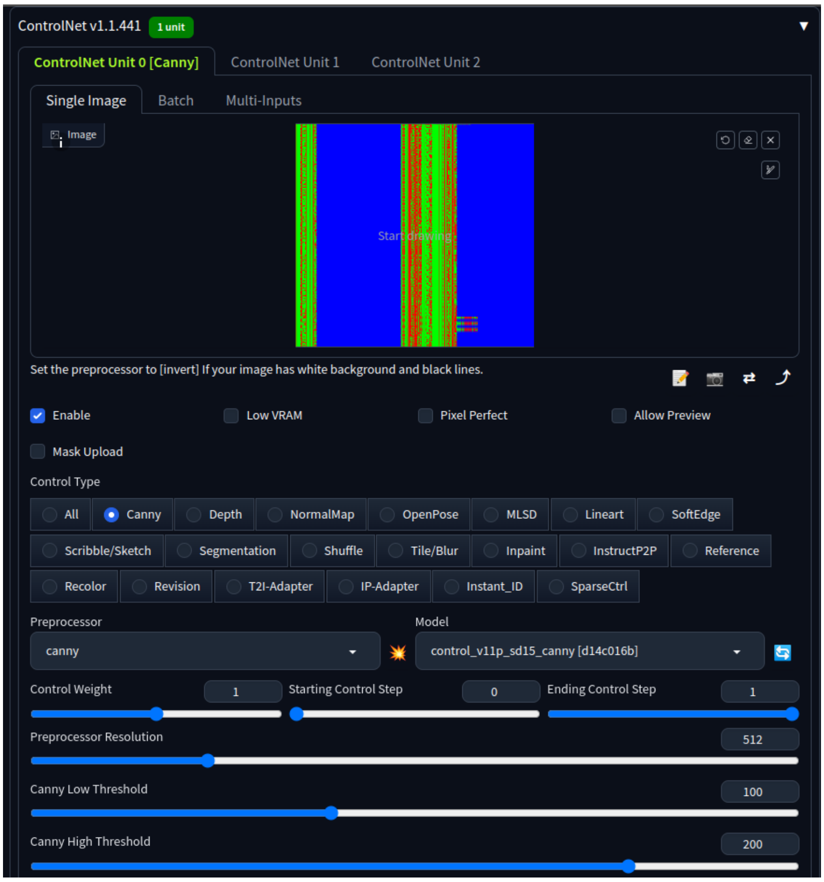
URL for extension's git repository: https://github.com/Mikubill/sd-webui-controlnet

Specific branch name: Leave empty for default main branch

Local directory name: Leave empty for auto

Figure 3.16: NetDiffusion: Installing ControlNet extension.

Remembering that the model name (with `.safetensors` extension) follows the same rule as in the previous steps. You can train different models with different names, but you must inform which one you are using in the prompt. For instance, consider we have a model with 3 applications (types 0 to 2), and we want to generate an



ControlNet v1.1.441 1 unit

ControlNet Unit 0 [Canny] | ControlNet Unit 1 | ControlNet Unit 2

Single Image | Batch | Multi-Inputs

Image

Start drawing

Set the preprocessor to [invert] If your image has white background and black lines.

☑ Enable □ Low VRAM □ Pixel Perfect □ Allow Preview

□ Mask Upload

Control Type

○ All ● Canny ○ Depth ○ NormalMap ○ OpenPose ○ MLSD ○ Lineart ○ SoftEdge

○ Scribble/Sketch ○ Segmentation ○ Shuffle ○ Tile/Blur ○ Inpaint ○ InstructP2P ○ Reference

○ Recolor ○ Revision ○ T2I-Adapter ○ IP-Adapter ○ Instant_ID ○ SparseCtrl

Preprocessor: canny

Model: control_v11p_sd15_canny [d14c016b]

Control Weight: 1

Starting Control Step: 0

Ending Control Step: 1

Preprocessor Resolution: 512

Canny Low Threshold: 100

Canny High Threshold: 200

Figure 3.17: NetDiffusion: Configuring Canny filter parameters on ControlNet extension.

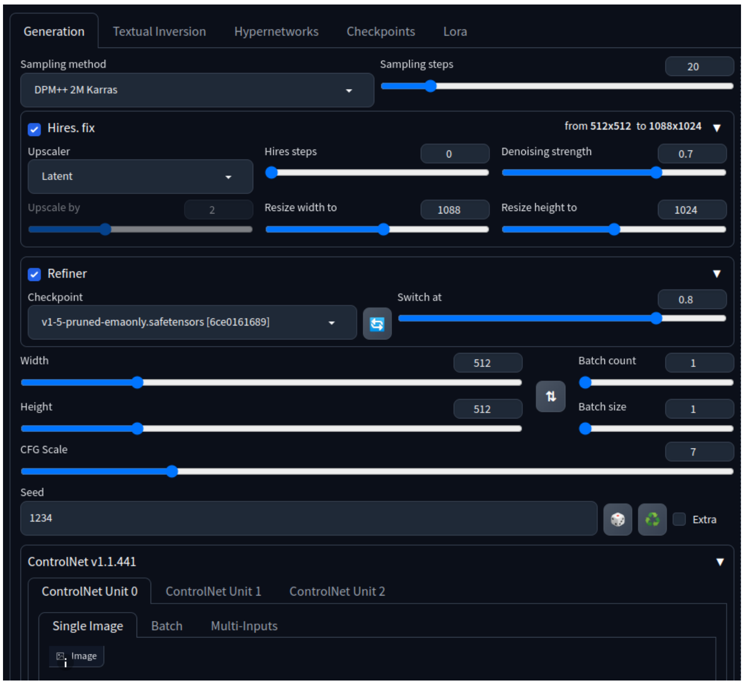
image representing a set of packets with the second class of applications. Then, we could set the prompt input as: `pixelated network data, type-1 <lora:my_model:1>`. To enable ControlNet, we must download it[13] and place it into *stable-diffusion-webui/extensions-webui-controlnet/models*

At this point, we already have a generated image. However, the generated image itself is not enough, as depicted in Figure 3.19, to reproduce a "replayable" PCAP (e.g., tcpreplay), since there is no guarantee the inter- and intra-packet dependencies are already being satisfied. In this case, the authors provided a set of post-heuristic scripts to convert the image back to nprint format and generate the final PCAP. First, we need to place the generated images into the correct folder - i.e., under */NetDiffusion_Generator/data/generated_imgs* – and navigate to the `/NetDiffusion_Generator/post-generation/` folder and run the post-generation scripts.

Figure 3.20 shows the execution of the post-heuristic processing. These steps complete the post-generation pipeline with the final nprints and PCAPs stored in */NetDiffusion_Generator/data/replayable_generated_nprints* and */NetDiffusion_Generator/data/replayable_generated_pcaps*, respectively.

To test the generated PCAPs, we can try to resend them through the local interface:

[13] Canny model (".pth" file)

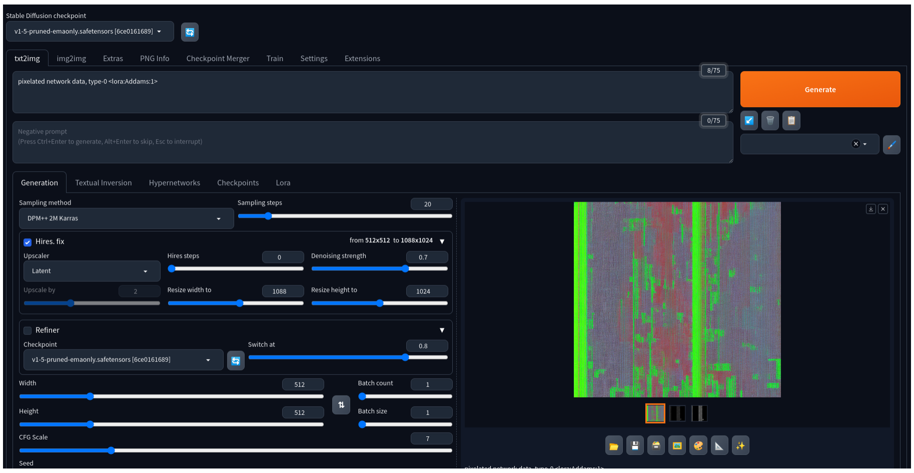


Sampling method: DPM++ 2M Karras
Sampling steps: 20
☑ Hires. fix — from 512x512 to 1088x1024
Upscaler: Latent
Hires steps: 0
Denoising strength: 0.7
Upscale by: 2
Resize width to: 1088
Resize height to: 1024
☑ Refiner
Checkpoint: v1-5-pruned-emaonly.safetensors [6ce0161689]
Switch at: 0.8
Width: 512
Height: 512
Batch count: 1
Batch size: 1
CFG Scale: 7
Seed: 1234
□ Extra

Figure 3.18: NetDiffusion: Upscaling the image to be generated.

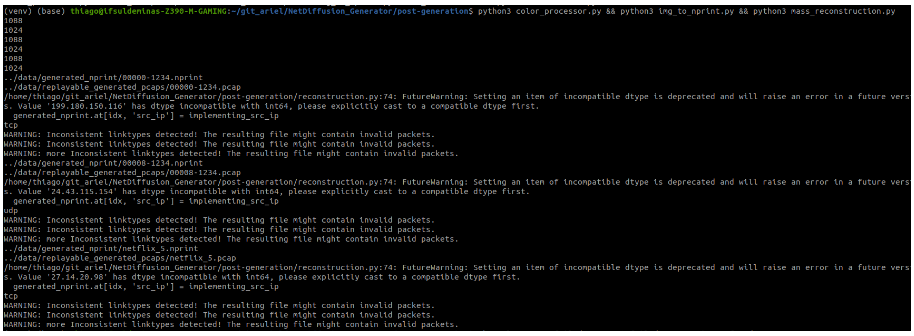
Code 3.27: NetDiffusion: Post-Generation Heuristic Correction

```
# Install tcpreplay
sudo apt update
sudo apt install tcpreplay
# Grab the local interface's name
ip a
# Navidate to the generated PCAPs folder
cd NetDiffusion_Generator/data/replayable_generated_pcaps
# Run tcpreplay with a generated PCAP file
sudo tcpreplay --loop=0 --verbose -i eno1 00000-1234.pcap
```



Stable Diffusion checkpoint: v1-5-pruned-emaonly.safetensors [6ce0161689]
8/75
pixelated network data, type-0 <lora:Addams:1>
0/75
Negative prompt
(Press Ctrl+Enter to generate, Alt+Enter to skip, Esc to interrupt)
Sampling method: DPM++ 2M Karras
Sampling steps: 20
☑ Hires. fix — from 512x512 to 1088x1024
Upscaler: Latent
Hires steps: 0
Denoising strength: 0.7
Upscale by: 2
Resize width to: 1088
Resize height to: 1024
□ Refiner
Checkpoint: v1-5-pruned-emaonly.safetensors [6ce0161689]
Switch at: 0.8
Width: 512
Height: 512
Batch count: 1
Batch size: 1
CFG Scale: 7
Seed

Figure 3.19: Generated image (without post-processing).



```
(venv) (base) thiago@ifsuldeminas-Z390-M-GAMING:~/git_ariel/NetDiffusion_Generator/post-generation$ python3 color_processor.py && python3 img_to_nprint.py && python3 mass_reconstruction.py
1088
1024
1088
1024
1088
1024
../data/generated_nprint/00000-1234.nprint
../data/replayable_generated_pcaps/00000-1234.pcap
/home/thiago/git_ariel/NetDiffusion_Generator/post-generation/reconstruction.py:74: FutureWarning: Setting an item of incompatible dtype is deprecated and will raise an error in a future vers
s. Value '199.180.150.116' has dtype incompatible with int64, please explicitly cast to a compatible dtype first.
  generated_nprint.at[idx, 'src_ip'] = implementing_src_ip
tcp
WARNING: Inconsistent linktypes detected! The resulting file might contain invalid packets.
WARNING: Inconsistent linktypes detected! The resulting file might contain invalid packets.
WARNING: more Inconsistent linktypes detected! The resulting file might contain invalid packets.
../data/generated_nprint/00008-1234.nprint
../data/replayable_generated_pcaps/00008-1234.pcap
/home/thiago/git_ariel/NetDiffusion_Generator/post-generation/reconstruction.py:74: FutureWarning: Setting an item of incompatible dtype is deprecated and will raise an error in a future vers
s. Value '24.43.115.154' has dtype incompatible with int64, please explicitly cast to a compatible dtype first.
  generated_nprint.at[idx, 'src_ip'] = implementing_src_ip
udp
WARNING: Inconsistent linktypes detected! The resulting file might contain invalid packets.
WARNING: Inconsistent linktypes detected! The resulting file might contain invalid packets.
WARNING: more Inconsistent linktypes detected! The resulting file might contain invalid packets.
../data/generated_nprint/netflix_5.nprint
../data/replayable_generated_pcaps/netflix_5.pcap
/home/thiago/git_ariel/NetDiffusion_Generator/post-generation/reconstruction.py:74: FutureWarning: Setting an item of incompatible dtype is deprecated and will raise an error in a future vers
s. Value '27.14.20.98' has dtype incompatible with int64, please explicitly cast to a compatible dtype first.
  generated_nprint.at[idx, 'src_ip'] = implementing_src_ip
tcp
WARNING: Inconsistent linktypes detected! The resulting file might contain invalid packets.
WARNING: Inconsistent linktypes detected! The resulting file might contain invalid packets.
WARNING: more Inconsistent linktypes detected! The resulting file might contain invalid packets.
```

Figure 3.20: Running the post-processing heuristics.

3.5.2.2. NetShare

Similarly to NetDiffusion, NetShare [51] is a recent work exploring the feasibility of using GANs to automatically learn generative models to generate synthetic packet- and flow header traces for network-ing tasks (e.g., telemetry anomaly detection, provisioning). However, they do not use image representation or nPrint; rather, they use a tabular representation distributed among different GANs with temporal and confinement dependencies. The input packet/flow trace is split into a set of flows based on their 5-tuples, and each of these flows is further split into N chunks based on the time duration of the flow. NetShare divides each flow into 10 chunks. The packet/flow fields of these chunks are encoded using continuous and categorical encoding functions, and these ecoded chunks are fed into NetShare's DoppleGANger time series GANs, a specific variant of time series GANs that work well with time series data. Once trained, these models are used to generate synthetic (encoded) chunks, which are decoded and assembled into a full trace by sorting all the packets/flows by their timestamps

The work is well documented in terms of reproducibility. However, a direct way to convert the CSV output of key packet fields to PCAP and verify the usability of network packets with statistical similarities to real data to increase data availability in network experiments is not publicly available.

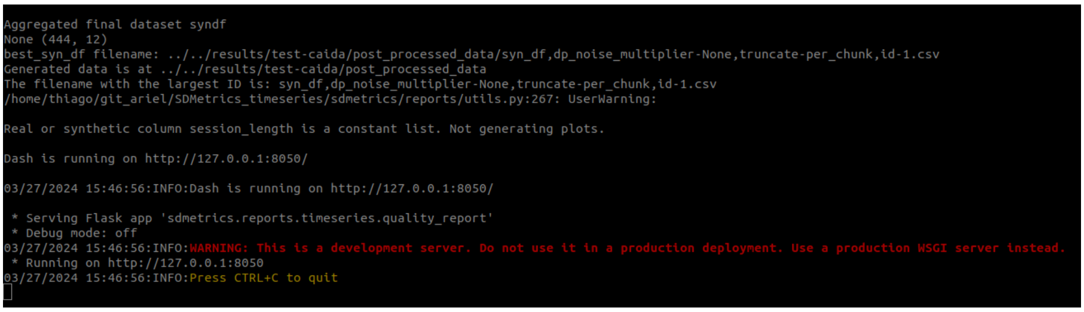
To install it, you may follow the step-guide installation tutorial on the official repository[14] or follow the next steps. First, we install the `libpcap-dev` library in the system - assuming we are on Ubuntu/Debian. Then, we create a Conda Python 3.9 virtual environment (line 6) and activate it (line 9). We then clone and install NetShare's dependencies (lines 12-13) and SDMetrics to help us plot and show the statistics after running the model (lines 16-17).

Code 3.28: **NetShare:** Setup Install

```
1 # Install libpcap depdency (Optional) - On Debian-based systems
2 sudo apt install libpcap-dev
3 # Assume Anaconda is installed
4 # Create virtual environment if not exists
5 conda create --name NetShare python=3.9
6 # Activate virtual env
7 conda activate NetShare
8 # Install NetShare package
```

[14] NetShare GitHub Documentation: https://github.com/netsharecmu/NetShare

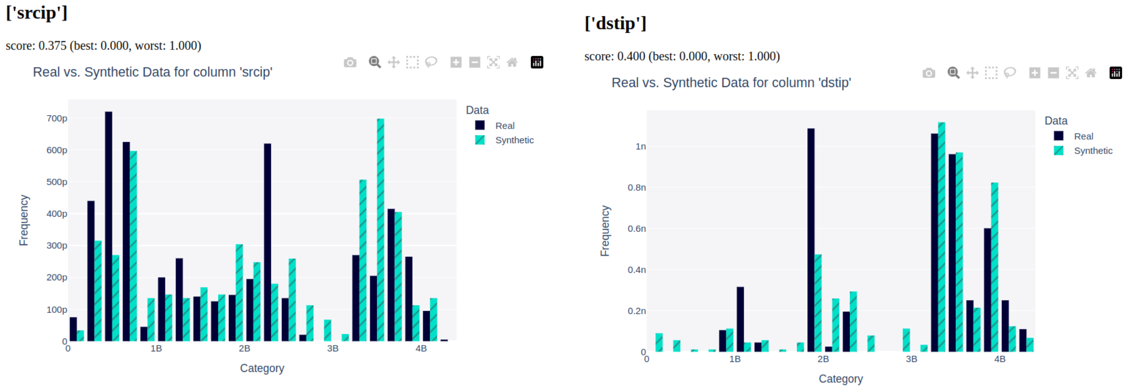
```
git clone https://github.com/netsharecmu/NetShare.git
pip3 install -e NetShare/
# Install SDMetrics package
git clone https://github.com/netsharecmu/SDMetrics_timeseries
pip3 install -e SDMetrics_timeseries/
```



```
Aggregated final dataset syndf
None (444, 12)
best_syn_df filename: ../../results/test-caida/post_processed_data/syn_df,dp_noise_multiplier-None,truncate-per_chunk,id-1.csv
Generated data is at ../../results/test-caida/post_processed_data
The filename with the largest ID is: syn_df,dp_noise_multiplier-None,truncate-per_chunk,id-1.csv
/home/thiago/git_ariel/SDMetrics_timeseries/sdmetrics/reports/utils.py:267: UserWarning:

Real or synthetic column session_length is a constant list. Not generating plots.

Dash is running on http://127.0.0.1:8050/

03/27/2024 15:46:56:INFO:Dash is running on http://127.0.0.1:8050/

 * Serving Flask app 'sdmetrics.reports.timeseries.quality_report'
 * Debug mode: off
03/27/2024 15:46:56:INFO:WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:8050
03/27/2024 15:46:56:INFO:Press CTRL+C to quit
```

Figure 3.21: NetShare: Running the example - PCAP generation mode.

To run the example code with PCAP generation[15], go to `NetShare/examples/pcap` and run `driver.py`. As shown in Figure 3.21, NetShare opens a local connection with a GUI, where the results are visualized. There are several metrics (Figure 3.22) comparing how statistically similar the data is in various fields of the packet (e.g., source IP, destination IP).



(a) NetShare: Real vs Synthetic Source IP.

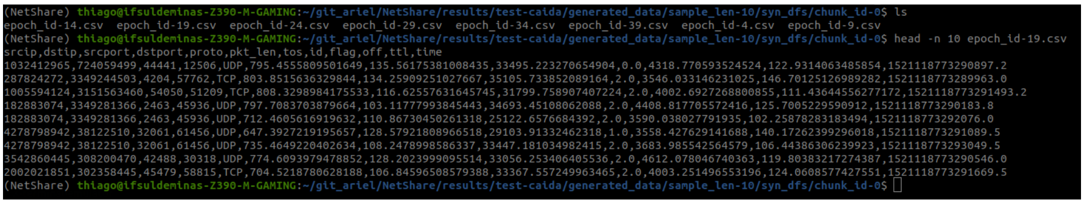
(b) NetShare: Real vs Synthetic Destiny IP.

Figure 3.22: NetShare: Some of the statistical results.

The results presented in Figure 3.22 are based on a sample CAIDA PCAP with only `76 KB` and do not represent the full potential of this work. The authors do provide a link to download the datasets[16] analyzed in the full paper for replicability. Although the code was available, the approach scales with compute resources, and the original models in the paper were trained on a 200 CPU + 200 GB Memory cluster, which are resources that neither we nor the authors have available at hand.

[15] NetShare also supports NetFlow data as input

[16] NetShare full-paper datasets: `https://drive.google.com/drive/folders/1FOl1VMr0tXhzKEOupxnJE9YQ2GwfX2FD?usp=sharing`



```
(NetShare) thiago@ifsuldeminas-Z390-M-GAMING:~/git_ariel/NetShare/results/test-caida/generated_data/sample_len-10/syn_dfs/chunk_id-0$ ls
epoch_id-14.csv  epoch_id-19.csv  epoch_id-24.csv  epoch_id-29.csv  epoch_id-34.csv  epoch_id-39.csv  epoch_id-4.csv  epoch_id-9.csv
(NetShare) thiago@ifsuldeminas-Z390-M-GAMING:~/git_ariel/NetShare/results/test-caida/generated_data/sample_len-10/syn_dfs/chunk_id-0$ head -n 10 epoch_id-19.csv
srcip,dstip,srcport,dstport,proto,pkt_len,tos,id,flag,off,ttl,time
1032412965,724059499,44441,12506,UDP,795.4555809501649,135.56175381008435,33495.223270654904,0.0,4318.770593524524,122.9314063485854,1521118773290897.2
287824272,3349244503,4204,57762,TCP,803.8515636329844,134.25909251027667,35105.733852089164,2.0,3546.033146231025,146.70125126989282,1521118773289963.0
1005594124,3151563460,54050,51209,TCP,808.3298984175533,116.62557631645745,31799.758907407224,2.0,4002.6927268800855,111.43644556277172,1521118773291493.2
182883074,3349281366,2463,45936,UDP,797.7083703879664,103.11777993845443,34693.45108062088,2.0,4408.817705572416,125.7005229590912,1521118773290183.8
182883074,3349281366,2463,45936,UDP,712.4605616919632,110.86730450261318,25122.6576684392,2.0,3590.038027791935,102.25878283183494,1521118773292076.0
4278798942,38122510,32061,61456,UDP,647.3927219195657,128.57921808966518,29103.91332462318,1.0,3558.427629141688,140.17262399296018,1521118773291089.5
4278798942,38122510,32061,61456,UDP,735.4649220402634,108.2478998586337,33447.181034982415,2.0,3683.985542564579,106.44386306239923,1521118773293049.5
3542860445,308200470,42488,30318,UDP,774.6093979478852,128.2023999095514,33056.253406405536,2.0,4612.078046740363,119.80383217274387,1521118773290546.0
2002021851,302358445,45479,58815,TCP,704.5218780628188,106.84596508579388,33367.557249963465,2.0,4003.251496553196,124.0608577427551,1521118773291669.5
(NetShare) thiago@ifsuldeminas-Z390-M-GAMING:~/git_ariel/NetShare/results/test-caida/generated_data/sample_len-10/syn_dfs/chunk_id-0$
```

Figure 3.23: NetShare: CSV results.

Finally, the results shown in Fig. 3.23 demonstrate that the output is a CSV file, not PCAP, as expected. It is unclear whether any script from the authors has not been publicly available to convert the packet fields into CSV format. However, tools like PCAP Generator [9] can be used to convert the CSV, as long as each column is correctly formatted.

3.6. Conclusions and future perspectives

This tutorial book chapter has explored the multifaceted role of GANs in the context of Computer Networks, underscoring their potential to simulate complex network environments and generate synthetic data. Through analysis and practical applications, it has been shown how GANs can be employed to create realistic network data, aiding in training RL algorithms and enhancing network management operations.

We discussed the evolution of generative models, focusing on GANs and their application in producing high-fidelity synthetic network traffic, including PCAP files. We showcased GANs' ability to model and generate data that nearly mimics real network traffic, highlighting their significance in data augmentation, privacy preservation, and the development of robust network management solutions.

The tutorial contribution of this book chapter provided a hands-on approach to understanding the generation of synthetic time series data using GANs and its subsequent application in network telemetry within PDPs. The discussions and examples illustrated the practicality of GANs in creating synthetic datasets that can be leveraged for training ML models, particularly in environments where obtaining real, labeled datasets is challenging or privacy-sensitive.

Emerging trends in the field indicate a growing integration of generative AI models like GANs with network management and optimization tasks. Continuous advancement in this area promises innovative solutions to advance network systems' capabilities in handling dynamic, complex, and resource-intensive tasks.

In conclusion, this chapter's exploration of GANs is a step toward understanding impact opportunities in the field of Computer Networks. Multiple challenges are open for research and development of synthetic data generation, network simulation, and the holistic integration of AI in network management, aiming to achieve more intelligent, autonomous, and efficient network systems.

Acknowledgments

This work has been supported by the following Brazilian research agencies: Federal Institute of Education, Science, and Technology of South of Minas Gerais - IFSULDEMINAS, FAPESP, and CAPES. This study was partially funded by CAPES, Brazil - Finance Code 001. This work was partially supported by the Innovation Center, Ericsson S.A., and by the Sao Paulo Research Foundation (FAPESP), grant 2021/00199-8, CPE SMARTNESS.

References

[1] Martin Abadi, Andy Chu, Ian Goodfellow, H Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep learning with differential privacy. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 308–318, 2016.

[2] Leandro Almeida, Jefferson Silva, Ricardo Lins, Paulo Maciel Jr., Rafael Pasquini, and Fábio Verdi. Wave - um gerador de cargas múltiplas para experimentação em redes de computadores. In *Anais Estendidos do XLI Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, pages 9–16, Porto Alegre, RS, Brasil, 2023. SBC.

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

[4] Serhat Arslan and Nick McKeown. Switches know the exact amount of congestion, 2019.

[5] Raef Bassily, Albert Cheu, Shay Moran, Aleksandar Nikolov, Jonathan Ullman, and Steven Wu. Private query release assisted by public data. In *International Conference on Machine Learning*, pages 695–703. PMLR, 2020.

[6] Jay Beale, Angela Orebaugh, and Gilbert Ramirez. *Wireshark & Ethereal network protocol analyzer toolkit*. Elsevier, 2006.

[7] Eoin Brophy, Zhengwei Wang, Qi She, and Tomás Ward. Generative adversarial networks in time series: A systematic literature review. *ACM Comput. Surv.*, 55(10), feb 2023.

[8] Tobias Bühler, Roland Schmid, Sandro Lutz, and Laurent Vanbever. Generating representative, live network traffic out of millions of code repositories, 2022.

[9] Levente Csikor. Pcap generator. `https://github.com/cslev/pcap_generator`, 2017. Accessed: 2024/03/10.

[10] Enyan Dai and Suhang Wang. Say no to the discrimination: Learning fair graph neural networks with limited sensitive attribute information, 2021.

[11] Leandro C de Almeida et al. Desired - dynamic, enhanced, and smart ired: A p4-aqm with deep reinforcement learning and in-band network telemetry. *Computer Networks*, 244(110326):1–19, 2024.

[12] Baik Dowoo, Yujin Jung, and Changhee Choi. Pcapgan: Packet capture file generator by style-based generative adversarial networks. In *2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA)*, pages 1149–1154. IEEE, 2019.

[13] Farnoush Falahatraftar, Samuel Pierre, and Steven Chamberland. A conditional generative adversarial network based approach for network slicing in heterogeneous vehicular networks. *Telecom*, 2(1):141–154, 2021.

[14] Luis Fernando Uria Garcia, Rodolfo S. Villaça, Moisés R. N. Ribeiro, Regis Francisco Teles Martins, Fábio Luciano Verdi, and Cesar Marcondes. Introdução à linguagem p4 - teoria e prática. *Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC) - Minicursos*, 2018.

[15] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets, 2014.

[16] Carroll Gray-Preston. Ai network applications, Sep 2023.

[17] Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron C Courville. Improved training of wasserstein gans. *Advances in neural information processing systems*, 30, 2017.

[18] Rohit Kumar Gupta, Shashwat Mahajan, and Rajiv Misra. Resource orchestration in network slicing using gan-based distributional deep q-network for industrial applications. *The Journal of Supercomputing*, 79(5):5109–5138, 2023.

[19] Ariel Góes de Castro, Fábio Rossi, Arthur Lorenzon, Marcelo Caggiani Luizelli, Francisco Vogt, Rumenigue Hohemberger, and Rodrigo Mansilha. Orchestrating in-band data plane telemetry with machine learning. *IEEE Communications Letters*, 23:20, 10 2019.

[20] Luchao Han, Yiqiang Sheng, and Xuewen Zeng. A packet-length-adjustable attention model based on bytes embedding using flow-wgan for smart cybersecurity. *IEEE Access*, 7:82913–82926, 2019.

[21] Frederik Hauser, Marco Häberle, Daniel Merling, Steffen Lindner, Vladimir Gurevich, Florian Zeiger, Reinhard Frank, and Michael Menth. A survey on data plane programming with p4: Fundamentals, advances, and applied research. *Journal of Network and Computer Applications*, 212:103561, 2023.

[22] Jordan Holland, Paul Schmitt, Nick Feamster, and Prateek Mittal. New directions in automated traffic analysis. CCS '21, page 3366–3383, New York, NY, USA, 2021. Association for Computing Machinery.

[23] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

[24] Yuxiu Hua, Rongpeng Li, Zhifeng Zhao, Honggang Zhang, and Xianfu Chen. Gan-based deep distributional reinforcement learning for resource management in network slicing, 2019.

[25] Xi Jiang, Shinan Liu, Aaron Gember-Jacobson, Arjun Nitin Bhagoji, Paul Schmitt, Francesco Bronzino, and Nick Feamster. Netdiffusion: Network data augmentation through protocol-constrained traffic generation. *arXiv preprint arXiv:2310.08543*, 2023.

[26] Alexey Kurakin, Shuang Song, Steve Chien, Roxana Geambasu, Andreas Terzis, and Abhradeep Thakurta. Toward training at imagenet scale with differential privacy. *arXiv preprint arXiv:2201.12328*, 2022.

[27] Gwo-Chuan Lee, Jyun-Hong Li, and Zi-Yang Li. A wasserstein generative adversarial network–gradient penalty-based model with imbalanced data enhancement for network intrusion detection. *Applied Sciences*, 13(14):8132, 2023.

[28] Zinan Lin, Alankar Jain, Chen Wang, Giulia Fanti, and Vyas Sekar. Using gans for sharing networked time series data: Challenges, initial promise, and open questions, 2020.

[29] Qiang Liu, Nakjung Choi, and Tao Han. Atlas: Automate online service configuration in network slicing. In *Proceedings of the 18th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '22, page 140–155, New York, NY, USA, 2022. Association for Computing Machinery.

[30] Pranav Mani, E. S. Gopi, Hrishikesh Shekhar, and Sharan Chandra. Generative adversarial network and reinforcement learning to estimate channel coefficients. In E. S. Gopi, editor, *Machine Learning, Deep Learning and Computational Intelligence for Wireless Communication*, pages 49–58, Singapore, 2021. Springer Singapore.

[31] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[32] Amar Meddahi, Hassen Drira, and Ahmed Meddahi. Sip-gan: Generative adversarial networks for sip traffic generation. In *2021 International Symposium on Networks, Computers and Communications (ISNCC)*, pages 1–6. IEEE, 2021.

[33] Jiayi Meng, Jingqi Huang, Y Charlie Hu, Yaron Koral, Xiaojun Lin, Muhammad Shahbaz, and Abhigyan Sharma. Characterizing and modeling control-plane traffic for mobile core network. *arXiv preprint arXiv:2212.13248*, 2022.

[34] Jesse G Meyer, Ryan J Urbanowicz, Patrick CN Martin, Karen O'Connor, Ruowang Li, Pei-Chen Peng, Tiffani J Bright, Nicholas Tatonetti, Kyoung Jae Won, Graciela Gonzalez-Hernandez, et al. Chatgpt and large language models in academia: opportunities and challenges. *BioData Mining*, 16(1):20, 2023.

[35] Qinghai Miao, Yisheng Lv, Min Huang, Xiao Wang, and Fei-Yue Wang. Parallel learning: Overview and perspective for computational learning across syn2real and sim2real. *IEEE/CAA Journal of Automatica Sinica*, 10(3):603–631, 2023.

[36] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning, 2015.

[37] Muhammad Haris Naveed, Umair Sajid Hashmi, Nayab Tajved, Neha Sultan, and Ali Imran. Assessing deep generative models on time series network data. *IEEE Access*, 10:64601–64617, 2022.

[38] Hojjat Navidan, Parisa Fard Moshiri, Mohammad Nabati, Reza Shahbazian, Seyed Ali Ghorashi, Vahid Shah-Mansouri, and David Windridge. Generative adversarial networks (gans) in networking: A comprehensive survey & evaluation, 2021.

[39] P4. In-band network telemetry (int) dataplane specification. Technical report, P4 Consortium, 2021.

[40] Cheng Qian, Wei Yu, Chao Lu, David Griffith, and Nada Golmie. Toward generative adversarial networks for the industrial internet of things. *IEEE Internet of Things Journal*, 9(19):19147–19159, 2022.

[41] Haneya Naeem Qureshi, Usama Masood, Marvin Manalastas, Syed Muhammad Asad Zaidi, Hasan Farooq, Julien Forgeat, Maxime Bouton, Shruti Bothe, Per Karlsson, Ali Rizwan, et al. Towards addressing training data scarcity challenge in emerging radio access networks: A survey and framework. *IEEE Communications Surveys & Tutorials*, 2023.

[42] Markus Ring, Daniel Schlör, Dieter Landes, and Andreas Hotho. Flow-based network traffic generation using generative adversarial networks. *Computers & Security*, 82:156–172, 2019.

[43] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 10684–10695, 2022.

[44] Mustafizur R Shahid, Gregory Blanc, Houda Jmila, Zonghua Zhang, and Hervé Debar. Generative deep learning for internet of things network traffic generation. In *2020 IEEE 25th Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 70–79. IEEE, 2020.

[45] Jacob Soper, Yue Xu, Kien Nguyen, Ernest Foo, and Zahra Jadidi. A two-pass approach for minimising error in synthetically generated network traffic data sets. In *Proceedings of the 2023 Australasian Computer Science Week*, pages 18–27. 2023.

[46] Kevin Spiteri, Ramesh Sitaraman, and Daniel Sparacio. From theory to practice: improving bitrate adaptation in the dash reference player. In *ACM Transactions on Multimedia Computing, Communications, and ApplicationsVolume 15Issue 2s*, pages 123–137, 06 2018.

[47] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[48] Mohammad Mustafa Taye. Understanding of machine learning with deep learning: Architectures, workflow, applications and future directions. *Computers*, 12(5), 2023.

[49] Zehui Xiong, Yang Zhang, Dusit Niyato, Ruilong Deng, Ping Wang, and Li-Chun Wang. Deep reinforcement learning for mobile 5g and beyond: Fundamentals, applications, and challenges, 2019.

[50] Shengzhe Xu, Manish Marwah, Martin Arlitt, and Naren Ramakrishnan. Stan: Synthetic network traffic generation with generative neural models. *arXiv preprint arXiv:2009.12740*, 2020.

[51] Yucheng Yin, Zinan Lin, Minhao Jin, Giulia Fanti, and Vyas Sekar. Practical gan-based synthetic ip header trace generation using netshare. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 458–472, New York, NY, USA, 2022. Association for Computing Machinery.

[52] Jinsung Yoon, Daniel Jarrett, and Mihaela van der Schaar. Time-series generative adversarial networks, 2019.

[53] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? *Advances in neural information processing systems*, 27, 2014.

[54] Da Yu, Saurabh Naik, Arturs Backurs, Sivakanth Gopi, Huseyin A Inan, Gautam Kamath, Janardhan Kulkarni, Yin Tat Lee, Andre Manoel, Lukas Wutschitz, et al. Differentially private fine-tuning of language models. *arXiv preprint arXiv:2110.06500*, 2021.

[55] Lvmin Zhang, Anyi Rao, and Maneesh Agrawala. Adding conditional control to text-to-image diffusion models. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 3836–3847, 2023.

[56] Cong Zou, Fang Yang, Jian Song, and Zhu Han. Generative adversarial network for wireless communication: Principle, application, and trends. *IEEE Communications Magazine*, pages 1–7, 2023.