Evaluation of Conditional Recurrent Generative Adversarial Networks for Multivariate Time-Series Augmentation

Master's thesis in Engineering Mathematics and Computational Science

ANNA CARLSSON

Department of Mathematical Sciences
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2020

MASTER'S THESIS

Evaluation of Conditional Recurrent Generative Adversarial Networks for Multivariate Time-Series Augmentation

ANNA CARLSSON

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Mathematical Sciences
Division of Applied Mathematics and Statistics
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2020

Evaluation of Conditional Recurrent Generative Adversarial Networks for Multivariate Time-Series Augmentation
ANNA CARLSSON

Supervisors:
Patrik Dammert and Håkan Warston, SAAB Surveillance
Torbjörn Lundh, Chalmers University of Technology

Examiner:
Torbjörn Lundh, Chalmers University of Technology

Master's Thesis 2020
Department of Mathematical Sciences
Division of Applied Mathematics and Statistics
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Synthetic sinusoidal time-series generated using the s-BiLSTM-CNN GAN architecture developed in this thesis. See Chapter 3 for further description of the architecture and Chapter 4 for evaluation of the synthetic time-series and the training process.

Typeset in LaTeX
Gothenburg, Sweden 2020

Evaluation of Conditional Recurrent Generative Adversarial Networks for Multivariate Time-Series Augmentation
ANNA CARLSSON
Department of Mathematical Sciences
Chalmers University of Technology

Abstract

A successful application of any machine learning algorithm is dependent on a sufficiently large training dataset, preferably class-balanced and correctly labeled. However, in many applications, the collection and labeling of data is time-consuming, expensive, and might require special security precautions if the data is of a sensitive nature. Therefore, different types of augmentation methods are commonly used. For time-series data, traditional augmentation methods such as rotation, translation, and flipping are not applicable. In applications where the dataset consists of time-series data, other augmentation methods are therefore of interest.

In this thesis, the usage of generative adversarial networks (GANs) as an augmentation method for univariate and multivariate time-series data is investigated. Both recurrent and conditional recurrent GANs are examined. Apart from constructing architectures for time-series generation, the thesis focuses on finding suitable methods for evaluating the quality of the generated data. To monitor the training progress and select a suitable generator model to simulate synthetic data from, two distance-based kernel metrics are used: maximum mean discrepancy (MMD) and energy distance (ED). To evaluate the sample quality and diversity of the generated data, several experiments are performed where a classifier is trained on real, tested on synthetic data (TRTS), trained on synthetic, tested on real data (TSTR), and lastly trained and tested on a mixture of real and synthetic data (TMTM). Furthermore, experiments aiming to examine the usage of synthetic samples from conditional recurrent GANs to augment a real dataset are performed.

The results indicate that the GANs successfully generates highly realistic samples, both of simpler time-series and more complex multivariate time-series. However, the time-series seem to not aid a classifier to any large extent when added to real data, even when larger proportions of synthetic data are added. A possible explanation for this is that the synthetic data, although consisting of realistic samples, suffers from loss of in-class diversity and boundary distortion.

Keywords: deep learning, generative adversarial networks, generative models, multivariate time-series classification, maximum mean discrepancy, energy distance, covariate shift, boundary distortion

Acknowledgements

Firstly, I would like to express my sincere gratitude to my industrial supervisors, Håkan Warston and Patrik Dammert, for invaluable support and encouragement throughout the project. I would also like to thank Torbjörn Lundh, my academic supervisor, for enthusiastically supporting my work and answering my questions.

I would also like to thank my manager Ulrika Svahn and the whole New Concepts team. Thank you for giving me the opportunity to work with such an interesting project and for warmly welcoming me into your team. I have thoroughly enjoyed all morning and afternoon coffee breaks, lunch walks, and interesting discussions.

Lastly, I would like to thank my family and friends. My family, for unconditionally supporting and encouraging me no matter which project I undertake. My friends, for supporting me during those five very intensive years. Thank you for being my late-night study company, for all coffee breaks spent together, and for all pep-talks. I cannot imagine that my time at Chalmers would have been nearly as good without each and every one of you.

Anna Carlsson, Gothenburg, June 2020

Contents

List of Figures

List of Tables

1 Introduction

Over the last few years, deep learning has become one of the most promising methods for a wide range of tasks, including classification problems. However, a successful application of deep learning relies on sufficient training data: that is, a sufficiently large training set, a balance between classes, and correctly labeled observations. Low-quality data can lead to poorly fitted models and overfitting issues. However, in many real-world applications, such ideal data is not possible to collect. For this reason, several augmentation techniques have been developed and are used to extend limited datasets without collecting any new data. In the image data domain, simple methods such as rotation, translation, and flipping are used to increase the number of training samples [1]. For other types of data, augmentation is not as straightforward. This is, in particular, true when it comes to time-series data, for which the so-called *temporal dependency* needs to be conserved in the augmented samples.

A field related to data augmentation is generative modeling: given a sample from a probability distribution, the aim is to model the underlying distribution from which the sample is drawn [2]. Such methods offer the possibility to augment a dataset by estimating the underlying distribution, drawing additional samples from this estimate, and including such synthetic samples in the training dataset to extend it. A wide range of methods for generative modeling has been proposed. Such models include analytic approaches where the distribution is modeled using some fixed family of distributions, graphical models, variational approaches, and neural network approaches such as Variational Auto-Encoders (VAEs) and Generative Adversarial Networks (GANs) [2].

Another related field is the qualitative evaluation of synthetic data from generative models with computationally intractable likelihoods. The problem of how such evaluation should be performed is an open research question, and yet today, many applications rely heavily on visual inspection of the synthetic samples.

In this thesis, an interesting boundary region between those areas is explored: the usage of GANs as an augmentation method for time-series data classification and evaluation of the synthetic data by some evaluation metrics, avoiding visual inspection. In the following sections, the background and scope of the project are given as well as a summary of some related work.

1.1 Background

Saab Surveillance is a business area within Saab AB, a Swedish defense company that develops and manufactures a wide range of defense and security solutions. At the unit Surface Radar Solutions, land and naval-based systems for primarily air surveillance and weapon location are developed. A key component of a radar system is the tracker, responsible for connecting radar observations of the same target into tracks describing the movement and position of the target. To determine what kind of object the radar is tracking, some type of classification algorithm is needed.

A previous master thesis investigated the usage of deep learning to classify such tracks recorded by the tracker in a certain radar system [3]. One conclusion of the thesis was that the results suffered from the small amount of training data available. Obtaining a sufficiently large dataset in a military setting can be problematic for several reasons. One reason is that it is time-consuming and expensive to run the measurement systems, record the data, and manually label the observations. Another reason is that military data is often of sensitive nature and special security precautions are needed to store and handle such data. Furthermore, the availability of recorded data for the most interesting targets can be limited. Therefore, it is interesting to examine the possibility of utilizing generative models to create synthetic samples from collected real-world data. Such synthetic data would not just be easier to obtain compared to real data, but would also be an improvement from a security perspective since it does not hold real-world information. Specifically, it is interesting to examine the characteristics of such generative models and their applicability for classification problems in radar systems. This can be done by comparing the distribution of synthetic data to the distribution of real-world data. An application of special interest is the classification of birds and UAVs (Unmanned Aerial Vehicles, commonly called drones) from data recorded by the radar tracker, as a continuation of the mentioned master thesis.

1.2 Scope

The scope of this thesis is to investigate the usage of generative adversarial networks (GANs) to generate time-series data that can be used for augmentation in a classification task. Furthermore, the quality of the synthetic data should be evaluated using some discrepancy metrics. In the sections below, the main limitations of the thesis are described.

The Training Data

The time-series data in question are two different datasets: one simpler univariate dataset consisting of simulated sinusoidal time-series and one multivariate time-series dataset, provided by Saab Surveillance. The latter is a real-world dataset, recorded by a tracker in one of Saab's radar systems.

The Generative Model

The generative models examined in this thesis are different types of generative adversarial networks (GANs). The reason for selecting GANs is that they have been incredibly successful in a wide range of tasks, including time-series forecasting and anomaly detection. Furthermore, the most common deep learning frameworks support the implementation of such networks. There exist a large number of well-documented variations of GANs, as well as scientific reports evaluating the performance of them. Specifically, recurrent generative adversarial nets (cGANs) will be used and in the case of the multivariate tracker data, they will be conditioned on the class labels. Such GANs are referred to as conditional recurrent GANs. Naturally, further types of GANs and other generative models exist that could be suitable for generating multivariate time-series, but such models are beyond the scope of this thesis.

Evaluation of the Synthetic Data

To evaluate the synthetic data generated in this thesis, two distance-based discrepancy metrics are used: maximum mean discrepancy (MMD) and energy distance (ED). They are both examples of two-sample tests, from the statistical domain. To evaluate the classification performance of the synthetic data, a classifier is needed. The selected classifier was used in a previous thesis done on the same dataset [3], and consist of a multilayered gated recurrent unit (GRU) network. No other architectures have been investigated since this thesis is limited to investigating the possibility of incorporating GANs in the classification framework used, rather than finding a better classifier.

1.3 Related Work

Studies relevant to this thesis are works utilizing GANs for time-series data generation and for general data augmentation. Furthermore, studies investigating methods for quality evaluation of synthetic data from generative models are of interest.

Evaluating the quality of generated data by generative models, for instance GANs, is an open research field. In a successful application of generative models, the synthetic data should capture the underlying distribution of the training data well. Several studies have aimed at developing frameworks for investigating whether or not generative models manage to capture such characteristics of the training data. In a study by S. Santhurkar et. al. [4], the authors investigate a phenomenon called *covariate shift* from a classification perspective. The authors denote the true data distribution $P_T(\boldsymbol{X})$ and the synthetic data distribution $P_G(\boldsymbol{X})$, and define covariate shift as the case when $P_T(\boldsymbol{Y}|\boldsymbol{X}) = P_G(\boldsymbol{Y}|\boldsymbol{X})$ but $P_T(\boldsymbol{X}) \neq P_G(\boldsymbol{X})$, where $\boldsymbol{X}$ is some source domain and $\boldsymbol{Y}$ is some target domain. A framework for detecting two types of covariate shift: mode collapse (the generator generates samples of some modes with larger probability) and boundary distortion (the synthetic data fails to capture the distribution in the boundary regions of the support) is presented. For this thesis, the test for detection of boundary distortion is relevant. The test

is performed by training a GAN separately on the data classes, form a balanced dataset, and check the classification accuracy when tested on real data. If the performance of the classifier is worse when trained on synthetic data compared to when trained on real data, it indicates that the generator fails to capture boundary cases present in the training dataset.

Several studies aiming at generating continuous time-series data with GANs have been performed. One of the first such studies was conducted by S. L. Hyland et. al. in 2017 [5]. The study aimed to generate multivariate continuous time-series using a GAN trained on a medical dataset from an intensive care unit (ICU). As a first step, a GAN was trained to generate simpler data such as sinusoidal time-series and a serialized version of the MNIST dataset of handwritten images [6]. All GANs in the study were recurrent, and for the ICU data, a conditional recurrent GAN was used. The authors successfully managed to generate realistic-looking univariate time-series, as well as more complex multivariate ICU time-series. A more recent study by A. M. Delaney et. al. from 2019 [7] focused on generating sinusoidal time series and electrocardiogram (ECG) signals. The study was based on a 2019 paper by F. Zhu [8], where ECG signals were generated using GANs. Both works showed promising results and managed to generate univariate time-series data successfully. Works, in which multivariate time-series data have been generated, are fewer but existent. Except for the already mentioned work by S. L. Hyland et. al. [5], two studies have been performed where GANs was used for anomaly detection in multivariate time-series [9, 10]. Both studies were based on the work by S. L. Hyland et. al. and they managed to train GANs in less than 100 epochs that could generate realistic synthetic samples.

1.4 Thesis Outline

The thesis is structured as follows. In Chapter 2, the theoretical preliminaries necessary to follow the work in this thesis are presented. The theory covers the basics behind neural networks, recurrent neural networks, generative adversarial networks, and the discrepancy metrics used in this thesis to evaluate synthetic data. In Chapter 3, the datasets, the various architectures used to generate and classify data, and the experiments performed on the synthetic data are presented. Furthermore, some practical details regarding training settings and training procedures of GANs are presented. In Chapter 4, results from the training processes of GANs and the results of the experiments made on the synthetic data are presented. In Chapter 5, the results are discussed. Lastly, in Chapter 6, the thesis is wrapped up and some methods that can be interesting to investigate further are described.

2 | Theory

In the following sections, the theoretical preliminaries needed to follow the rest of the report are presented. In Section 2.1, a brief introduction to unsupervised and supervised learning is presented together with some main challenges of supervised learning. In Section 2.2, the theory behind some relevant types of artificial neural networks is covered: mainly feed-forward networks, convolutional neural networks, and recurrent neural networks. In Section 2.3, the theoretical foundation of generative adversarial networks is covered. Lastly, in Section 2.4, some evaluation metrics that can be used to compare samples from the true dataset with samples from a synthetic dataset generated by a generative model are presented.

2.1 Supervised and Unsupervised Learning

Supervised learning is the problem of teaching an algorithm to perform a certain task based on an outcome variable, commonly referred to as the *target* variable [11]. The input variables to the algorithm are commonly referred to as *feature* variables. Examples of supervised learning methods are linear and logistic regression, random forest classification, support vector machines (SVM), and artificial neural networks (mostly). Unsupervised learning is, in contrast, the problem of learning from data without having any target variable associated with each observation. Examples of unsupervised learning methods are cluster analysis and generative modeling. In this thesis, both supervised (deep neural networks) and unsupervised learning (generative adversarial networks) will be used.

One of the main challenges with supervised learning is to obtain models with high generalizability [1]. Generalizability refers to the ability of the model to perform well on data it has not previously seen. If a trained model has poor generalizability, it has either underfitted or overfitted the training data. Underfitting refers to the case when a too simple model has been used. When overfitting, the model finds patterns in noise and fails to adapt to the true underlying features of the training data. When such an overfitted model is tested on new data, the model performs poorly. In a sense, the model can be considered to be too flexible: a too large number of parameters gives the model ability to adapt to noise in the training data. However, it is desirable that the model is flexible to some extent: a too rigid model will fail to capture the true relationship in the data since it is too simple. This relationship between under- and overfitting is usually referred to as the bias-variance tradeoff.

Many methods exist for avoiding overfitting and are widely used when training almost any supervised model. Several such methods will be discussed in a subsequent section.

2.2 Artificial Neural Networks

Artificial neural networks is a flexible and powerful framework for solving supervised learning tasks inspired by the structure and dynamics of the biological brain [12]. Although the usage of neural networks has practically exploded both within academia and a wide range of industries during the last few years, neural networks are not a new idea: the basic neural unit we use today was first proposed by McCulloch and Pitts in 1943 [12, 13]. Due to the rapid increase in hardware performance and the increase of available training datasets, neural networks are nowadays used in many different fields. Common applications include object recognition, speech recognition, and machine translation and new application areas are under constant development.

In the following sections, some theory behind artificial neural networks is presented. McCulloch and Pitts' simple neural unit is presented in Section 2.2.1. In Section 2.2.2, a simple network structure with a single hidden layer is presented. Such a simple structure serves well to build an understanding of neural networks in general. In Section 2.2.3, the training process of a neural network is described as well as some methods for reducing overfitting of the training data. In Section 2.2.4, the principles behind convolutional neural networks are presented. Such networks are commonly used for processing temporal and spatial data. Lastly, in Section 2.2.5, recurrent neural networks and some well-known developments such as long-short term memory (LSTM) and gated recurrent unit (GRU) networks are introduced.

2.2.1 The Basic Neural Unit

A simple neural unit is shown in Figure 2.1. The neural unit has index i. As can be seen in the figure, neuron i receives input from N other upstream neurons. Each output from the upstream neurons is assigned a weight, which determines how much a certain output should affect the input of neuron i. The output of neuron i is computed by summing the weighted input together and then apply an *activation function* as follows:

$$x_i = f\left(\sum_{j=1}^{N} w_{ij} x_j - b_i\right), \tag{2.1}$$

where f denotes the activation function, w_{ij} is the weight of the jth input neuron to neuron i, x_j is the output of neuron j, and b_i is the threshold (or the bias) of neuron i.

The activation function is usually a nonlinear function, needed to introduce some non-linearity to the model. A common choice is the *rectified linear unit* (ReLU) function, defined as [14]:

$$\mathrm{ReLU}(x) = \max(0, x), \tag{2.2}$$

where x corresponds to the summarized input to the neuron, sometimes referred to as the *local field* [12]. Inspired by the mammalian brain, a neuron can either be active (excitatory) or inactive (inhibitory). To be active, the neuron needs to receive a sufficiently large input from the upstream neurons. The threshold of the neuron determines how large the local field needs to be for the neuron to be active. The values of the weights can be either positive, negative, or zero. If a weight from a certain upstream neuron is set to zero, the output of this neuron will not contribute to the activation of neuron i. A negative weight means that the input from a certain upstream neuron will decrease when the output of the upstream neuron increases.

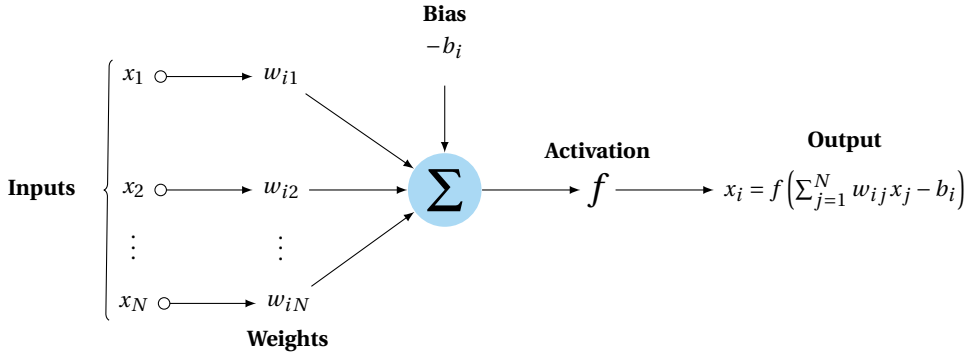


Figure 2.1: An illustration of a single neural unit, originally proposed by McCulloch and Pitts [13]. The blue neuron in the figure has index i, and the output of the ith neuron is computed by summing the weighted input $w_{ij}x_j$ together, adding the bias, and applying the nonlinear activation function f. The bias b_i is sometimes referred to as the *threshold* of neuron i.

2.2.2 A Simple Feed-Forward Network

A simple feed-forward network is shown in Figure 2.2. The name refers to the fact that information is always fed forward in the network, and no feedback loops to upstream layers or neurons within the same layer exist. Sometimes, such networks are called *multilayered perceptrons* [12]. The first layer is referred to as the *input* layer. The number of neurons in this layer is determined by the shape of the data: mostly, there are as many input neurons as features in the data. The middle layer is referred to as a *hidden* layer since the states of the neurons in this layer are not accessible [12]. The number of neurons in such a layer can vary. The last layer is called the *output* layer. This layer delivers the final result of the network, and the dimension of this last layer depends on the problem and the structure of the network. For binary classification, the number of output neurons is usually two. For multi-class classification, the number of output neurons usually corresponds to the number of target classes. Note that the number of hidden layers can vary; for some simpler problems, no hidden layers are necessary and for some more complex problems, several layers may be needed.

In Figure 2.2, some notation is introduced. The states of the input neurons are denoted x_i^1, where i denotes the number of the neuron in the layer and 1 refers to the fact that the input layer is also the first layer. The same goes for the hidden neurons and for the output neurons, but their states will be denoted h_i^l and y_i^l, respectively, where l denote the layer number. In some

cases, vector notation is preferable. The states of the neurons in the input layer are denoted $\boldsymbol{x}^{(1)}$. Similarly, the state of the hidden neuron is denoted $\boldsymbol{h}^{(l)}$. In a classification setting, the difference between the true labels of the observations, commonly denoted $\boldsymbol{y}$, and the output of the network is clarified by denoting the output of the network $\hat{\boldsymbol{y}}^{(l)}$. When confusion might occur, this notation will be used in this thesis too. The weight matrix of each layer is denoted $\boldsymbol{W}^{(l)}$, where l refers to the layer number. Note that the layer index will mostly be omitted for the input and output layers in subsequent sections.

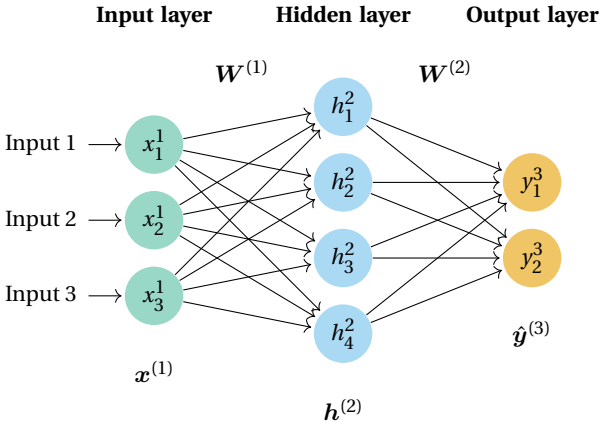


Figure 2.2: A simple feed-forward network with a single hidden layer. In the figure, both scalar and vector notation are introduced. For practical reasons, vector notation is often preferred.

2.2.3 Training a Neural Network

As have been seen, a neural network consists of neurons, connected to downstream neurons, and the output of upstream neurons is fed forward through the network until it reaches the output layer. But how does the network learn? How can we prevent overfitting of the training data? Such questions will be answered in the following sections.

Backpropagation and Stochastic Gradient Descent

Backpropagation refers to the process of evaluating the error of the network, and then sequentially update the weights of the network to allow the network to perform better in subsequent training iterations. This is almost always performed using a *gradient descent* algorithm. Gradient descent-type of algorithms performs optimization by iteratively stepping in the direction of the steepest descent, which corresponds to the negative gradient:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \alpha \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_k), \tag{2.3}$$

where $\boldsymbol{\theta}_{k+1}$ are the trainable parameter values at iteration $k+1$, $\boldsymbol{\theta}_k$ are the parameter values at iteration k, α is the learning rate (the step size of the algorithm), and $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_k)$ is the gradient of the cost function $J(\boldsymbol{\theta}_k)$ at iteration k with respect to the trainable parameters. The learning

rate can be constant, but more often some kind of adaptive learning rate is implemented. More details on this will be described later in this section.

The process of feeding information forward in the network to obtain an output is called *forward propagation*. The forward propagation algorithm is summarized in Algorithm 1 following [14]. As can be seen in Equation 2.3, the gradient of the cost function with respect to the trainable parameters needs to be computed to update the weights. Since the output is computed by sequentially transforming the input, the gradients of parameters belonging to upstream layers can be computed with the chain rule, using gradients of the downstream parameters. This process is usually called *backpropagation*. The backpropagation algorithm is summarized in Algorithm 2, also here following [14].

Algorithm 1: The forward propagation algorithm, following [14]. $\Omega(\theta)$ is a regularizer with some weight parameter λ. $\odot$ denotes element-wise multiplication.

input : number of layers l, weight matrices $\boldsymbol{W}^{(i)}$, $i \in \{1,\dots,l\}$, the bias parameters $\boldsymbol{b}^{(i)}$, $i \in \{1,\dots,l\}$, the features (input) $\boldsymbol{x}$, the target $\boldsymbol{y}$
output: loss function $J = L(\boldsymbol{y}, \hat{\boldsymbol{y}})$

$\boldsymbol{h}^{(0)} = \boldsymbol{x}$;
for $k = 1,\dots,l$ **do**
  $\boldsymbol{a}^{(k)} = \boldsymbol{b}^{(k)} + \boldsymbol{W}^{(k)}\boldsymbol{h}^{k-1}$; // local field
  $\boldsymbol{h}^{(k)} = f\left(\boldsymbol{a}^{(k)}\right)$; // activation
$\hat{\boldsymbol{y}} = \boldsymbol{h}^{(l)}$; // predicted output
$J = L(\boldsymbol{y}, \hat{\boldsymbol{y}}) + \lambda\Omega(\theta)$; // loss

Algorithm 2: The backpropagation algorithm, following [14]. The gradient of the output layer is first computed and then used to compute upstream gradients. Also here, $\Omega(\theta)$ corresponds to a regularizer and λ some weight parameter.

input : loss function $J = L(\boldsymbol{y}, \hat{\boldsymbol{y}})$

$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}, \boldsymbol{y})$; // gradient of output layer
for $k = l, l-1,\dots,1$ **do**
  $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f'(\boldsymbol{a}^{(k)})$; // gradient of activation
  $\nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g} + \lambda\nabla_{\boldsymbol{b}^{(k)}}\Omega(\theta)$; // gradient of biases
  $\nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g}\boldsymbol{h}^{(k-1)\intercal} + \lambda\nabla_{\boldsymbol{W}^{(k)}}\Omega(\theta)$; // gradient of weights
  $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)\intercal}\boldsymbol{g}$; // gradient of layer k-1's activation

Activation Function and Loss Function

The activation function introduces non-linearity to the network. Without the activation function, the network would simply perform a series of linear transformations, which could be insufficient to learn the pattern of more complex data. By using nonlinear transformations $\phi(\boldsymbol{x})$, where $\boldsymbol{x}$ is some local field, we can apply a linear model to the transformed data and thus model nonlinear relationships [14]. Several activation functions are commonly used in neural network architectures and are suitable in different situations.

The most commonly occurring activation function is the already mentioned rectified linear unit (ReLU) function (see Section 2.2.1). Although simple, ReLU has some large advantages, making it the recommended choice for feed-forward and convolutional neural networks [14]. Firstly, the gradients of the active neurons are always one. Other activation functions, such as sigmoid and the hyperbolic tangent function, tend to cause very small gradients when the local fields are either very large or very small. This is not the case for ReLU, and therefore this activation function is considered more stable for learning. One disadvantage is the risk of so-called *dead neurons*: neurons that independently of the input always outputs zero. Why such neurons arises vary, but since the gradients of such neurons are always zero, they cannot be recovered. In some applications, this is a desired property since it makes the network sparser. The mentioned sigmoid and hyperbolic tangent functions are defined as:

$$\text{sigm}(x) = \frac{1}{1+e^{-x}} \tag{2.4}$$

and

$$\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}. \tag{2.5}$$

The sigmoid function normalizes the output between 0 and 1 and is therefore often used to normalize the output to probabilities. Since the GAN discriminator should output the probability that a sample is real, a sigmoid activation function is often used in the output layer of a discriminator. A disadvantage of the sigmoid function is that it easily saturates if the input to the activation is very small or very large [14]. Therefore, it is important to make sure that the outputs of the layers are kept relatively close to zero since the sigmoid is most sensitive there: otherwise, using sigmoid units in a network can make the learning process difficult. For this reason, it is discouraged to use the sigmoid function in-between network layers.

The loss function is central in all gradient descent-based learning algorithms and measures the inconsistency between the network output and the true labels of the data. The loss function is the objective function minimized by the gradient descent algorithm. There exists many different loss functions, and which one to use depends on the problem at hand. In the context of GANs, a common choice of loss function is the binary cross-entropy loss. In Section 2.3.1, the specific loss function for GANs will be further discussed, but here we will state the formal definition of binary cross-entropy loss:

$$L = L(p_{\boldsymbol{\theta}}) = -\big(y_i \log\big(p_{\boldsymbol{\theta}}(y_i)\big) + (1-y_i)\log\big(1-p_{\boldsymbol{\theta}}(y_i)\big)\big), \tag{2.6}$$

where y_i is the true class of the observation (either 0 or 1), $p_{\boldsymbol{\theta}}(y_i)$ is the probability of observation y_i belonging to the first class, and $1-p_{\boldsymbol{\theta}}(y_i)$ the probability of y_i belonging to the second class.

Adaptive Learning Rate

Gradient descent-based learning algorithms require a learning rate (step size), to perform optimization. How this parameter should be chosen is not straightforward, and it has proven to be a hyperparameter that can greatly affect the performance of the trained model. How sensitive the cost is to changes in parameter values can substantially vary between different parameters. For this reason, different types of algorithms with *adaptive learning rates* have been proposed.

One of the most popular adaptive learning rate optimization algorithms is the Adam (Adaptive Moment estimator) algorithm [15]. The original stochastic gradient descent algorithm has a fixed learning rate for all parameters. The Adam algorithm, on the other hand, computes adaptive learning rates for each parameter. The idea is to adapt the learning rate to how the gradients have changed in the past: the algorithm saves the exponentially decaying average of past gradients (denoted m_t for time t, the gradients themselves are denoted g_t) and the past squared gradients (denoted v_t for time t). The exponentially decaying averages are defined as:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2. \end{aligned} \tag{2.7}$$

Since the averages are initialized as zero vectors, they are biased towards zero. This is especially prominent during early iterations according to the original paper [15]. Therefore, a bias correction is needed and is defined as:

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{(1-\beta_1^t)} \\ \hat{v}_t &= \frac{v_t}{(1-\beta_2^t)}. \end{aligned} \tag{2.8}$$

If the gradients in the past have changed much, it is reasonable to slow down training a bit to prevent taking too large steps and hence miss an optimal solution. If the past gradients have changed slowly, then the step size should be increased. This yields the Adam update rule:

$$\theta_t = \theta_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon}. \tag{2.9}$$

This is repeated until convergence is reached. The default values of β_1 and β_2 is usually 0.9 and 0.999, respectively, but can require modifications in different applications.

Regularization

Regularization is a method common in both statistics and machine learning that can be used to prevent overfitting by shrinking coefficient estimates towards zero [16]. Effectively, this

corresponds to reducing the number of trainable parameters. The two most used regularization methods are the *lasso* (L1-regularization) and *ridge regression* (L2-regularization). There also exist some methods more specific to deep learning, such as dropout. All three methods will be briefly described in the following paragraphs.

In the case of deep learning, the coefficient estimates are simply the trainable weights and biases. As was briefly introduced in Section 2.1, overfitting in machine learning usually refers to when a model performs well on some particular training data partition but fails to capture the general structure of the data and hence generalizes badly when tested on other partitions. Regularization is applied to prevent the model from becoming too complex, by reducing the number of trainable parameters and/or punishing large parameter values. Both the lasso and ridge regression adds a term to the loss function, which punishes models with a large number of weights. In the case of the lasso, the loss function is given by [14]:

$$J_{L_1} = L(\boldsymbol{y}, \hat{\boldsymbol{y}}) + \lambda\Omega(\boldsymbol{\theta}) = L(\boldsymbol{y}, \hat{\boldsymbol{y}}) + \lambda\|\boldsymbol{w}\|_1 = L(\boldsymbol{y}, \hat{\boldsymbol{y}}) + \lambda\sum_i |w_i|, \tag{2.10}$$

meaning that the regularization term corresponds to the sum of the absolute values of each individual weight. Here, J_{L_1} is the total loss function, $L(\boldsymbol{y}, \hat{\boldsymbol{y}})$ is the loss computed from the labels $\boldsymbol{y}$ and the network output $\hat{\boldsymbol{y}}$, λ is a fixed parameter, and $\Omega(\theta)$ is the regularization term dependent on the trainable parameters, in this case the weights $\boldsymbol{w}$. The lasso gives in general more *sparse* solutions compared to the non-regularized solution by reducing some weights to zero. Ridge regression, on the other hand, does too reduce the size of the weights but tends to keep all the parameters in the model. The loss function in the case of ridge regression is given by [14]:

$$J_{L_2} = L(\boldsymbol{y}, \hat{\boldsymbol{y}}) + \lambda\Omega(\boldsymbol{\theta}) = L(\boldsymbol{y}, \hat{\boldsymbol{y}}) + \lambda\|\boldsymbol{w}\|_2^2 = L(\boldsymbol{y}, \hat{\boldsymbol{y}}) + \lambda\sum_i w_i^2, \tag{2.11}$$

meaning that the squares of all weights in a layer are added to the loss function. While the lasso sets parameters to zero, ridge regression never shrinks parameters to zero. Practically, this gives models that have a smaller variance but a slightly larger bias [16]. The fact that the squared weights are added to the loss function also punishes large weights, which could stabilize the training process since large weights can lead to exploding gradients.

Dropout is a different type of regularization method specific to neural networks. The idea is that for each training batch, a binary mask is randomly sampled and then applied to the input and hidden layers [14]. Since multiplying the output and input of a neuron with zero practically removes the neuron from the network, each such randomly sampled binary mask will for each batch remove different units from the network. The probability of generating a mask value one at a certain position in the mask is a hyperparameter set before the training phase. The result is that the model is slightly different in each training step, which has shown to significantly reduce the risk of overfitting on the training data and giving models that better generalize to testing data [17].

Cross-Validation

An important part of the training process of any statistical model is to evaluate how well the model performs. Ideally, data collection is simple and a *test dataset* can be obtained. The performance of the model can then be evaluated by predicting the target of the unseen observations in the test dataset. However, when further data collection is not possible, cross-validation is commonly used. A simple version of cross-validation is to split the dataset into a training dataset and a test dataset, and then train the model on the training dataset and evaluate the model on the test dataset.

Sometimes, for instance if the dataset is small, withholding a proportion of the dataset purely for evaluation might not be suitable, since the fitted model might generalize poorly due to lack of training data. In such cases, *k-fold cross-validation* (k-fold CV) is often utilized. The dataset is firstly split into k folds. Then, the model is trained k times, each time using $k-1$ folds as training data and one fold as the test set. This means, that after k training rounds, each fold has been used for training $k-1$ times and as test fold once. The total error of the model is computed by averaging over the individual cross-validation errors obtained when testing the withheld test-fold in each iteration.

An alternative approach to cross-validation is *Monte Carlo cross-validation* (MCCV) [18]. The approach is similar to k-fold cross-validation, but instead of splitting the data into fixed folds, the data is at each iteration randomly split into a training and test set. Also here, the total error of the model is obtained by averaging over the errors obtained in each iteration. An important difference between MCCV and k-fold CV is that in the latter, all observations are used for training and testing at some point. In MCCV, it is not guaranteed that all samples will be in the training or test sets at some time. However, MCCV explores a larger number of possible partitions into training and test sets, which is not the case in k-fold CV since the splits are fixed.

2.2.4 Convolutional Neural Networks

Convolutional neural networks (CNNs) are networks suitable for data with some known spatial or temporal structure. Examples of such structures are time-series (one-dimensional temporal data) or images (two-dimensional spatial data) [14]. One of the main advantages of CNNs is that they have relatively few trainable parameters compared to fully-connected (dense) networks and are therefore more computationally efficient. The “convolutional” part of the name stems from the mathematical operation *convolution*, which is the key feature of convolutional neural networks. In Figure 2.3, an example of a simple CNN is shown. In this case, the CNN processes 28×28 pixel images from the MNIST dataset of handwritten images [6] and aims at classifying an image into one of the classes 0-9. In the upcoming sections, each type of layer will be briefly described.

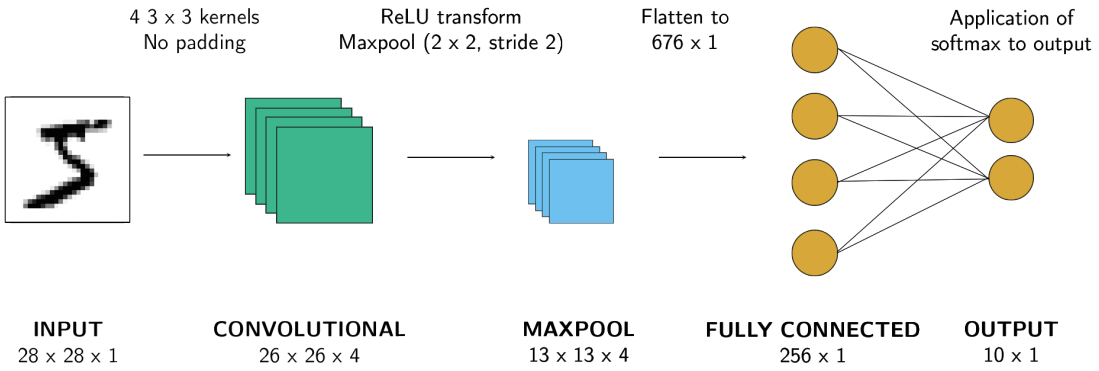


Figure 2.3: Illustration of a simple convolutional neural network (CNN) processing a single image from the MNIST dataset of handwritten images [6].

Input Layer

The input layer consists of neurons, fed with the input data. In the case of images, the layer accepts input in three dimensions: height (H), width (W), and depth (D) where depth is the number of color channels in the image. For instance, an RGB image has three color channels (red, green, and blue) and a grayscale image has a single color channel. In the general case, when the data is fed in *batches* to the network, the input layer accepts a fourth dimension corresponding to the batch size. Batch training is used in most applications and means that the trainable parameters in the network are updated after evaluating the errors from a batch rather than after a single sample. In Figure 2.3, the dimension of the input layer is $28 \times 28 \times 1$ since an MNIST image has dimensions 28×28, is grayscale, and no batch training is used.

Convolution Layer

In the convolution layer, the mathematical operation *convolution* is central. Convolution is an operation applied to two functions, f and g, and results in a function $(f * g)$, which describes the amount of overlap between f and g as f is shifted over g [19]. In the one-dimensional discrete case, the convolution between f and g is defined as:

$$(f * g)(t) = \sum_{k \in K} f(k)g(t-k). \tag{2.12}$$

In machine learning contexts, the first component usually corresponds to the input and the second component to the *convolution kernel*. The resulting component is sometimes called the *feature map* [14].

Practically, the kernel is a tensor with a predefined size. Similar to the input layer, the convolution kernel has width and height dimensions ($W \times H$) but also a depth dimension corresponding to the number of channels. This last dimension is often omitted in notation. The kernel is repeatedly applied to the input by "sliding" the kernel over the input image, and at each position,

the convolution between the kernel and the covered part of the image is computed. The step size of the sliding operation is called *stride*. A common value of the stride is one. Unless some padding is done with the input, some combinations of kernel size and stride may cause the output image to shrink in its spatial dimensions. Sometimes, this effect it desired since it downsamples the input data. The depth of the output is *not* related to the number of input channels: convolution is performed over all the input dimensions (W, H, and D) so each applied kernel results in an image with depth one. The total output depth is determined by the number of kernels used.

In Figure 2.3, four kernels of size 3×3 is used with no padding of the input. This results in a reduction of width and height of the output of the convolution layer and the four kernels result in four output channels.

Pooling Layer

The pooling layer is included in a convolutional network to reduce the spatial dimensions of the data, which results in less trainable parameters and hence a reduced risk of overfitting [20]. A pooling layer also makes the output less sensitive to translations of the input [14]. A pooling operation is applied to each depth dimension of the input separately and to sections of the data with predefined sizes. The idea is to replace each such section, or neighborhood, with a summary of the values in the neighborhood. The most common pooling layer is the max-pooling layer. In this case, each neighborhood of the input data is in the output represented by the maximum value of the neighborhood [14]. Another example of a common pooling function is average pooling, where each neighborhood is represented by its average value.

In Figure 2.3, the kernel size of the max-pooling layer is 2×2 and the stride is 2. This results in a size reduction of the output (13×13). Since the pooling operation is applied to each depth dimension separately, the depth of the output is not affected.

Fully Connected Layer

In most convolutional networks, the layer or layers before the output layers are fully connected. It is not necessary to include such dense layers before the output layers: one could replace them with convolutional layers or simply omit them, but it is common practice to include them since it is a simple way of transforming the dimensions of the convolutional layer output to the required shape of the output. The input to the fully connected layer is flattened and then fed forward through the layer similar to a simple fully-connected network.

In Figure 2.3, the output of the max-pooling layer is flattened to dimensions 676×1 and then fed into a fully connected layer. Here, the fully connected layer has 256 neurons, but this is an arbitrary choice and should be tuned as a hyperparameter. Also, note that not all neurons are shown in the figure.

2.2.5 Recurrent Neural Networks

So far, all networks in this chapter have been on a feed-forward layout, meaning that information is always transferred forward in the network. However, there also exist networks with *recurrent* connections. Such networks are suitable to use with sequential data, including time-series data, since their structure allows for the re-usage of weights across time-steps [14]. This type of network is called *recurrent neural networks* (RNNs). Their usage with sequential data can be motivated with a simple example. For a feed-forward network to process a time-series, it would need specific input nodes with different parameters for each time-step of the sequence. If the sequence is a sentence and text recognition should be performed, the network would need to learn the rules of the language for each position in the sentence since the weights are not shared between the nodes that process the different parts of the sentence. In a recurrent network, however, the weights are shared across the full sequence, and a much smaller number of trainable parameters is needed. Sharing in this context means that present network output is fed back into the network and influences future output. This is sometimes referred to as a recurrent network's "memory". Since information is fed back in the network from a later stage, the network "remembers" old information and lets this information influence the current output.

Sequential data does not necessarily have to be dependent on time. In machine translation, for instance, the input is sentences. A sentence is an example of sequential data independent of time but ordered by for instance some index. Since this thesis deals with time-series data, the notation will be adapted to time-dependent data but the results generalize to other types of sequential data as well. We denote *past* input $\boldsymbol{x}^{(0)}, \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t-1)}$ and the present input $\boldsymbol{x}^{(t)}$. The output at time t is denoted $\boldsymbol{y}^{(t)}$.

Due to the feedback connections in a recurrent network, it is not possible to directly train the network using backpropagation. However, by unfolding the network in time, using the backpropagation algorithm is possible. An example of a simple RNN is shown in Figure 2.4. The input x_t is fed into a hidden neuron, which also receives feedback in the form of the previous hidden state h_{t-1}. This network can be unfolded in time to simplify the structure. This is also shown in Figure 2.4. Here, the state of the hidden neuron is dependent on both the input x_t at time step t and the state h_{t-1} at time step $t-1$. With this network structure, the network can be trained in practically the same way as a normal feed-forward network using some gradient descent-based algorithm. After unfolding the network it corresponds to a multilayer network where each time step corresponds to a network layer.

In practice, simple RNNs such as the one in Figure 2.4 are seldom used. The reason for this is that they suffer from vanishing and exploding gradients, which makes the training process unstable. Several more complex recurrent networks have been suggested and are widely used in for instance natural language processing and machine translation. In the upcoming section, special cases of recurrent networks are presented. Firstly, some theory of bidirectional recurrent

networks is given. Such networks have shown impressive results in tasks where the prediction of the output $\boldsymbol{y}^{(t)}$ of the network is dependent on the whole sequence, such as speech recognition and bioinformatics [14]. Bidirectional RNNs combine a network that moves forward in time with a network that moves backward in time, and the result is that the network has access to both backward and forward information at each time step. Next, two examples of gated RNN cells called long short-term memory networks (LSTMs) and gated recurrent unit networks (GRUs) are presented. Both networks are constructed to be more stable during training and to suffer less from vanishing and exploding gradients.

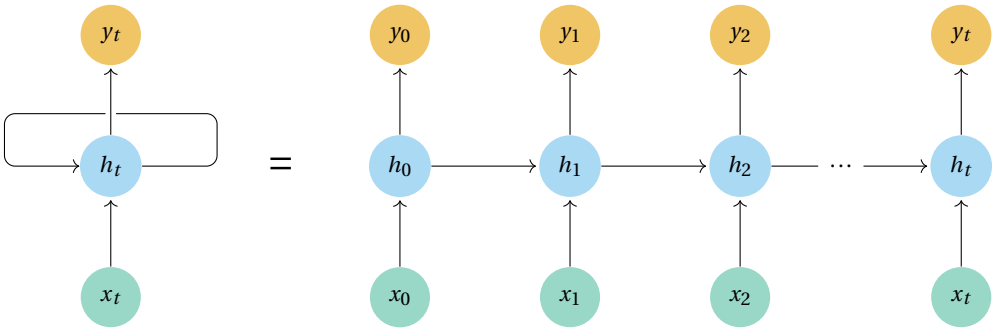


Figure 2.4: A simple recurrent neural network (RNN) unfolded in time. The current state h_t of the network depends on both the input x_t and the state at time $t-1$, h_{t-1}. To be able to train the recurrent network using backpropagation, the network is unfolded in time.

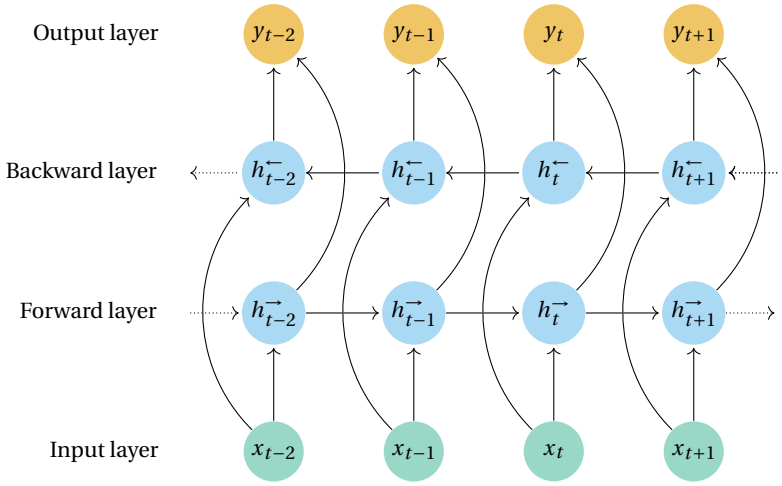


Figure 2.5: A simple example of a bidirectional recurrent network. The input $\boldsymbol{x}_t$ is fed to the corresponding neuron in both the forward and the backward RNN and thus affects states $\boldsymbol{h}_t^{\rightarrow}$ and $\boldsymbol{h}_t^{\leftarrow}$. The forward state $\boldsymbol{h}_t^{\rightarrow}$ is also dependent on information from the past, $\boldsymbol{h}_{t-1}^{\rightarrow}$, and the backward state $\boldsymbol{h}_t^{\leftarrow}$ is also dependent on future information, $\boldsymbol{h}_{t+1}^{\leftarrow}$. The output is thus dependent on the input of the current time step, but also information from past and future states.

Bidirectional Recurrent Networks

As have been previously mentioned, bidirectional recurrent networks use two recurrent networks: one transversing the data forward in time (from the beginning of the sequence) and

one transversing the data backward in time (from the end of the sequence). In Figure 2.5, a simple example of a bidirectional RNN is shown. As can be seen, the input $\boldsymbol{x}_t$ is fed to the corresponding neuron in both the forward and the backward RNN and thus affects states $\boldsymbol{h}_t^{\rightarrow}$ and $\boldsymbol{h}_t^{\leftarrow}$. The forward state $\boldsymbol{h}_t^{\rightarrow}$ is also dependent on information from the past, $\boldsymbol{h}_{t-1}^{\rightarrow}$, and the backward state $\boldsymbol{h}_t^{\leftarrow}$ is also dependent on future information, $\boldsymbol{h}_{t+1}^{\leftarrow}$. The output is thus dependent on the input of the current time step, but also information from past and future states.

The idea of bidirectional RNNs can easily be generalized to higher dimensions. In the case of images, a bidirectional network would need *four* different recurrent networks, each processing the data in one of the four possible directions: up, down, left, and right [14].

Long-Short Term Memory Network

The most widely used RNN cell in sequence modeling is the LSTM cell. A schematic illustration of an LSTM cell can be found in Figure 2.6. One cell corresponds to the unit shown in the figure, and there are usually several such cells connected in a row, similar to the hidden units in the previous recurrent networks shown. It is also possible to stack such LSTM layers, and thus create a multilayered LSTM network. The gray box represents the cell, the blue parts are neural networks, and the orange ellipses represent pointwise operations (addition or multiplication). Each LSTM cell has three important components. The first component is the cell state, which in the figure is represented by the upper horizontal line. The cell state is passed from the previous cell to the current cell and further on to the next cell. As can be seen in the figure, information from the cell can be added and removed to the cell state. This change of the cell state is regulated by the so-called *gates*. Gates are, as the name suggests, units that can let information pass but not necessarily do. The gates correspond to the blue sigmoid components in the figure and are hence neural networks. Sigmoid units output a value on the interval $[0, 1]$, and this number determines how much information that will be passed through the gate. The leftmost gate in Figure 2.6 is called the *forget gate* and it's output is denoted f_t. The forget gate considers the previous *hidden state* h_{t-1} and the current input x_t and determines how much of the old information of the cell state c_{t-1} to keep. The hidden state is, despite its name, formally the output of the cell. This state is passed between units in the same layer, but also to the layer above. In Figure 2.6, this is illustrated using two h_t outputs: one fed to the next cell, one fed to the next layer. This also means, that if the previous layer was also an LSTM layer, then the input x_t is the output h_t of the previous layer. Formally, the output f_t of the forget gate is given by:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f), \tag{2.13}$$

where W_f is the trainable weights of the forget gate and b_f is the trainable bias. This factor is then multiplied with c_{t-1}, and the result is either that c_{t-1} is kept as it is (then f_t equals one) or that c_{t-1} is reduced. The next gate in Figure 2.6 is the *input gate*. The input gate considers h_{t-1} and x_t and determines the importance of the input. If the input is considered important, it

should influence the new cell state. This is once again done using a sigmoid function:

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i), \tag{2.14}$$

where W_i is trainable weights and b_i trainable bias of the input gate. In parallel to the input gate, the tanh layer creates a vector $\tilde{c}_t$ of new proposed values to the cell state. The values are given by:

$$\tilde{c}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c), \tag{2.15}$$

where W_c is the trainable weights of the tanh layer and b_c is the trainable bias. Those values are multiplied with the input state i_t, and the result is added to the cell state. The total cell state is thus given by:

$$c_t = f_t \cdot c_{t-1} + i_t \cdot \tilde{c}_t. \tag{2.16}$$

The last gate is the *output gate*. The output gate decides the output of the cell, h_t. The output is a filtered version of the cell state c_t. The filtering is made by a sigmoid layer, which uses h_{t-1} and x_t to determine which information in c_t that should be outputted. The cell state is pushed through a tanh function and then multiplied with the filter. The resulting output is h_t. Formally, this corresponds to:

$$h_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \cdot \tanh(c_t), \tag{2.17}$$

where W_o is the trainable weights of the output gate and b_o is the trainable bias.

The intuition behind LSTMs success in many tasks is the fact that they can dynamically vary the length of its memory (hence the name, long-short term memory) [14]. Hence, some information is accumulated over large periods, and some information that is not considered important is forgotten.

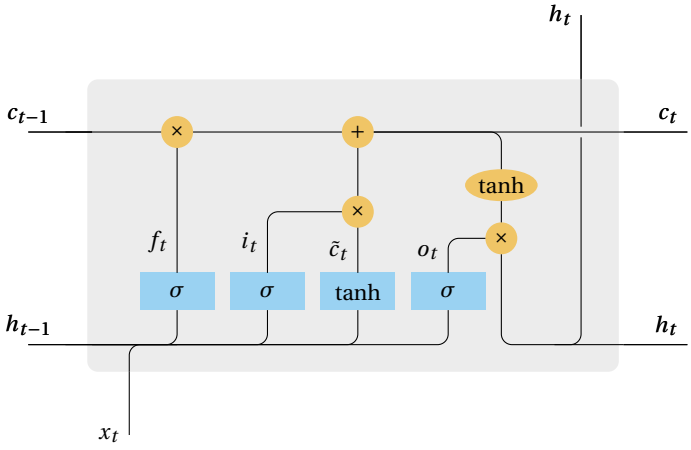


Figure 2.6: Illustration of a single long-short term memory (LSTM) unit. The three gates are the forget gate, which determines how much of the old information from the old cell state to keep, the input gate, which determines how much of the new information to include in the new cell state, and the output gate, which determines the output of the gated cell. The figure is based on a similar figure from [21].

Gated Recurrent Unit

The GRU cell is a newer type of recurrent cell and has a slightly simpler structure compared to the LSTM cell. The cell lacks a cell state, but consists of a hidden state and two gates: the *reset* and the *update* gates. An illustration of a GRU cell can be seen in Figure 2.7. The contribution to the hidden state from the reset gate is formally given by:

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t] + b_r), \tag{2.18}$$

where W_r and b_r are the trainable weights and bias of the reset gate. The other gate, the update gate, is similar to the reset gate but as will be seen, the output of the gate is used differently. The output of the update gate is given by:

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t] + b_z), \tag{2.19}$$

where W_r and b_r are the trainable weights and bias of the update gate. The new hidden state of the cell is given by:

$$h_t = (1 - z_t) \cdot h_{t-1} + z_t \tilde{h}_t, \tag{2.20}$$

where $\tilde{h}_t$ is the *proposed new hidden state* given by:

$$\tilde{h}_t = \tanh(W \cdot [r_t \cdot h_{t-1}, x_t]). \tag{2.21}$$

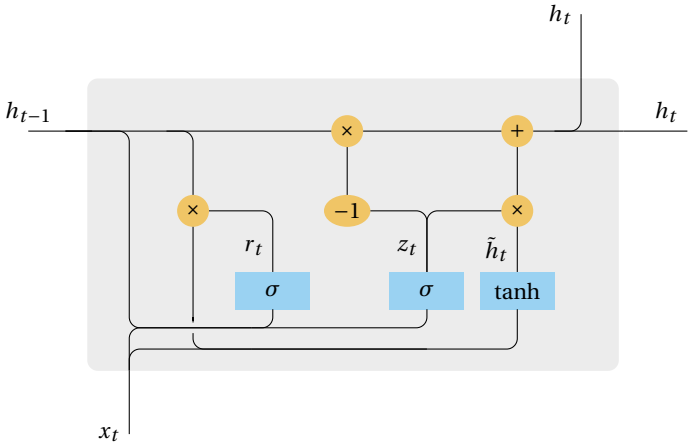


Figure 2.7: Illustration of a single gated recurrent unit (GRU). The cell has a hidden state and two gates: the *reset* and the *update gates*. The reset gate computes a factor determining how much of the old hidden state to keep. The update gate computes a factor z_t that determines the proportion of the old hidden state and the proposed hidden state in the output. The figure is based on a similar image from [21].

The output of the reset gate, r_t, is used to determine how much of the old hidden state to keep in the proposed new hidden state $\tilde{h}_t$. If r_t equals zero, then the reset gate opts for "resetting" the hidden state to the proposed state $\tilde{h}_t$. However, it is the update gate that determines the proportion of the old hidden state and the proposed hidden state in the output. The output

of the update gate, z_t, corresponds to the proportion of the proposed hidden state $\tilde{h}_t$ to be let through and $(1 - z_t)$ corresponds to the proportion of the old hidden state to be let through. This is summarized in Equation 2.20.

2.3 Generative Adversarial Networks

Generative Adversarial Networks (GANs) were first proposed in a 2014 article by Goodfellow et. al. [22] and has gained a tremendous research interest during the last few years. GANs are examples of generative models: models that take training data following some distribution p_{data}, and learns to estimate samples from this distribution [23]. GANs consists of two separate neural networks: the *generator* and the *discriminator*, trained using conflicting objectives. The generator produces samples, intending to make them as similar as possible to the training data. The discriminator is fed either a real or a fake sample and then estimates the probability of the sample being real rather than produced by the generator.

Formally, both the generator and the discriminator are functions differentiable with respect to their inputs and to their parameters [23]. They are commonly denoted G and D, respectively. The function D takes $\boldsymbol{x}$ (the observed variables) as input and its parameters are usually denoted $\boldsymbol{\theta}^{(D)}$. The generator G takes $\boldsymbol{z}$ (the latent variables) as input and its parameters are usually denoted $\boldsymbol{\theta}^{(G)}$. The objective of the discriminator, $J^{(D)}\left(\boldsymbol{\theta}^{(D)}, \boldsymbol{\theta}^{(G)}\right)$, is dependent on the parameters of both the discriminator and the generator, but it can only access its own parameters $\boldsymbol{\theta}^{(D)}$. The same goes for the generator; its objective function $J^{(G)}\left(\boldsymbol{\theta}^{(D)}, \boldsymbol{\theta}^{(G)}\right)$ is dependent not only on its own parameters, but also the discriminator's parameters, but it can only control its own parameters $\boldsymbol{\theta}^{(G)}$. Both the generator and the discriminator aims at minimizing their own objective functions, but since they cannot control all parameters, it is commonly described as a *game* rather than an optimization problem. The aim of the whole training process is to reach a *Nash equilibrium* of the game. A Nash equilibrium can be described by a tuple $\left(\boldsymbol{\theta}^{(D)}, \boldsymbol{\theta}^{(G)}\right)$ where $\boldsymbol{\theta}^{(D)}$ locally minimizes $J^{(D)}$ and $\boldsymbol{\theta}^{(G)}$ locally minimizes $J^{(G)}$.

2.3.1 The Training Objective and the Loss Function

As mentioned, the two networks are simultaneously trained using conflicting objectives. There exist several different versions of objectives for GAN training, but in all versions, it is only the objective of the generator, $J^{(G)}$, that differs. In the following sections, the different objectives associated with the discriminator and generator will be presented. The choice of loss function for the generator results in different games played by the generator and discriminator.

The Discriminator Objective

The cost function used for training the discriminator is [23]:

$$J^{(D)}\left(\boldsymbol{\theta}^{(D)},\boldsymbol{\theta}^{(G)}\right) = -\frac{1}{2}\mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}}\left[\log D(\boldsymbol{x})\right] - \frac{1}{2}\mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}}\left[\log\left(1-D(G(\boldsymbol{z}))\right)\right], \tag{2.22}$$

where $p_{\boldsymbol{z}}$ is the prior distribution of $\boldsymbol{z}$, commonly $\mathrm{U}(0,1)$ or $\mathcal{N}(0,1)$. We may recall the definition of the cross-entropy loss function defined in Equation 2.6 in Section 2.2.3. The difference is that Equation 2.6 is written for individual observations, while Equation 2.22 is written in terms of expected values over a number of observations. Also, here the classifier is trained on two different sets of data: one set consisting of real samples, labelled 1, and one set consisting of fake samples from the generator labelled 0. Hence, the objective defined in Equation 2.22 is the binary cross-entropy loss. In simple terms, minimizing this cost function is equivalent to maximizing the probability that D assigns 1s to real samples, and 0s to fake samples.

The Minimax Generator Objective

A simple version of the game played by the generator and discriminator is the *zero-sum game*. In this game, the total cost of all players equals zero at all times. In the context of GANs, this corresponds to [22]:

$$J^{(G)} = -J^{(D)}. \tag{2.23}$$

The total training objective is usually denoted $V(D,G)$. Here, V stands for *value function*. Since the cost functions of G and D are the same except for the sign, the total objective can simply be taken to be $J^{(D)}\left(\boldsymbol{\theta}^{(D)},\boldsymbol{\theta}^{(G)}\right)$. Since the function should be maximized with respect to the discriminator parameters $\boldsymbol{\theta}^{(D)}$ and minimized with respect to the generator parameters $\boldsymbol{\theta}^{(G)}$, the objective is maximized in an inner loop and minimized in an outer. Therefore, the game is sometimes referred to as a minimax game:

$$\begin{gathered}\min_G \max_D V(D,G) = \min_G \max_D \left(-J^{(D)}\left(\boldsymbol{\theta}^{(D)},\boldsymbol{\theta}^{(G)}\right)\right) = \\ \min_G \max_D \left(\frac{1}{2}\mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}}\left[\log D(\boldsymbol{x})\right] + \frac{1}{2}\mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}}\left[\log\left(1-D(G(\boldsymbol{z}))\right)\right]\right).\end{gathered} \tag{2.24}$$

The cost function in Equation 2.24 was presented in the original GAN article in 2014 and has some neat properties for theoretical analysis. It has however turned out to perform badly in practical applications [23]. This is because the generator maximizes the same binary-cross entropy as the discriminator minimizes. For the generator, this means that when the discriminator has managed to assign correct labels to the real images with high probability, the generator receives less cost [23]. Hence, the gradients of the generator vanish. Since the discriminator often assigns correct labels to real images with high probability in the beginning of the training process when the generator still outputs nonsense, using this cost function may prevent the generator from learning anything at all.

The Heuristic Non-Saturating Generator Objective

An alternative loss function for the generator, commonly referred to as the heuristic non-saturating objective, is defined as:

$$J^{(G)}\left(\boldsymbol{\theta}^{(D)}, \boldsymbol{\theta}^{(G)}\right) = -\frac{1}{2}\mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}}\left[\log D(G(\boldsymbol{z}))\right]. \tag{2.25}$$

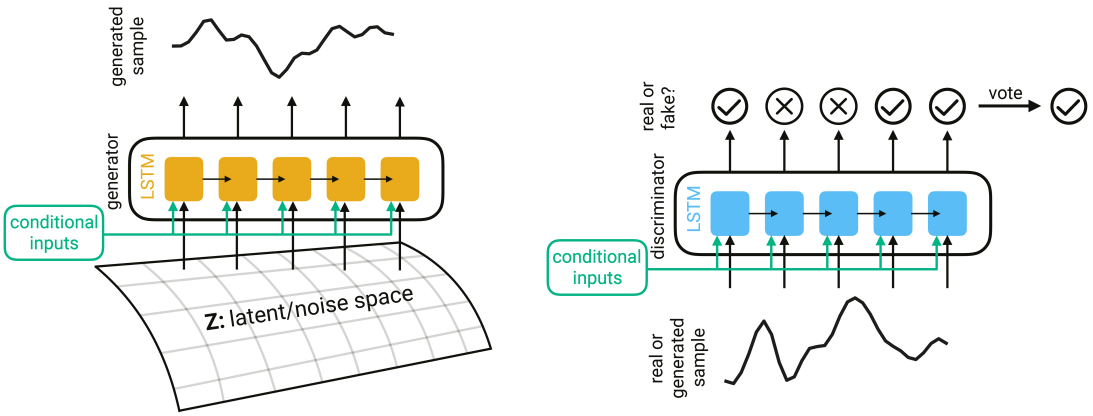
Two important things should be noted. Firstly, the generator cost does only consist of one term. This is based on the idea that the first term in Equation 2.24 does not contribute to the gradients of the generator. Secondly, the labels are flipped. Instead of minimizing the probability of the discriminator being correct, the generator is now trained by maximizing the probability of the discriminator being wrong. Note that if this loss function is used, the game is no longer a zero-sum game and cannot be described by a single value function.

2.3.2 Conditional Generative Adversarial Networks

In an ordinary GAN, there is no way of controlling what data is generated by the model, for instance with respect to class labels. In 2014, an extension to the GAN framework called *conditional* generative adversarial networks (cGANS) was proposed [24]. The idea is to provide both the generator and the discriminator with additional information, for instance class labels, to direct which modes of the data that is generated.

In most applications, both the generator and the discriminator is provided with the additional information $\boldsymbol{y}$. $\boldsymbol{y}$ is not necessarily class labels but could be any information unique for the data class. Other studies have provided numerical vectors describing the surrounding environment in the case of sensor modeling [25], facial attributes in the case of facial image generation [26], and scene attributes for outdoor scene generation [27], just to mention a few examples.

In this project, the additionally provided information will be class labels. We let $\boldsymbol{y}$ denote the corresponding class labels to the input $\boldsymbol{x}$. $\boldsymbol{y}$ is provided to both the generator and the discriminator as a one-hot-encoded matrix, where each column corresponds to a class and each row to an observation. Thus, the row sum should equal to one since each observation only belongs to one class. There exist several methods to feed the conditional input to the networks in a GAN. In this thesis, methods relevant to time-series data are of interest. A common method in the context of sequential data is to concatenate the input to the network with the conditional input at each time step. With this approach, we can ensure that the network cannot forget the conditional input. An example of a conditional recurrent GAN can be seen in Figure 2.8. Here, LSTM units are used in both the generator and discriminator. At each time step, the input is concatenated with the conditional information. For each time step, the discriminator determines if the sample time step is real or fake, and then performs a majority vote to determine if the full sequence is real or fake.



(a) Generator **(b)** Discriminator

Figure 2.8: An example of a conditional recurrent GAN for time-series generation. This network consists of LSTM units in both generator and discriminator to process the input time-series. At each time step, the network is conditioned with additional information to prevent the network from forgetting the conditional information. The discriminator determines at each time step if the inputted sequence is real or fake, and then bases the total judgment by a majority vote over the full sequence. Courtesy of [5].

2.3.3 Improving Training Stability

Ever since the introduction of generative adversarial networks in 2014, it has been known that the training process of GANs can be difficult to succeed with. Except for problems common among all neural network training processes, such as vanishing and exploding gradients, there exist several issues with GANs related to their ability (or disability) of generating data that captures the true diversity of the training data. Two such examples are mode collapse and boundary distortion, which were introduced in Section 1.3. Since gradient descent-based algorithms aim at minimizing a loss function and not directly find the Nash equilibrium, a common problem is that the network fails to converge [28]. In the following sections, two common methods for stabilizing the training progress and encourage training convergence are presented.

Minibatch Discrimination

A common problem when training GANs is that the generator collapses into always generating very similar or identical samples. This is a form of severe mode collapse: the model is unable to generate anything but a specific mode. The problem is that the discriminator handles each sample in a batch independently, and may therefore have a gradient that points in the same direction for many iterations [28]. Since the discriminator is unable to tell that the samples are nearly identical, the gradients are independently determined and since the loss of the generator is indirectly based on the gradients of the discriminator loss, no feedback is given to the generator that it should produce more diverse samples. The idea behind minibatch discrimination is to let the discriminator compare the similarity between samples within a batch and assign each batch a similarity score. The score is appended to the batch and can be used to guide the discriminator

in whether or not the samples in the batch are real or fake. If the samples are too similar, the similarity score signals that the batch is fake, and to be able to keep improving, the generator is forced to produce more diverse samples.

Formally, we may denote an input to the discriminator $\boldsymbol{x}_i$. In some intermediate layer in the discriminator, the feature vector of this input can be denoted $f(\boldsymbol{x}_i) \in \mathbb{R}^A$, where A is the dimension of the feature vector [28]. To perform minibatch discrimination, this feature vector is multiplied with a tensor $T \in \mathbb{R}^{A \times B \times C}$. Both B and C can be considered to be hyperparameters: as we will see, B corresponds to the length of the similarity vector and C is sometimes referred to as the *hidden dimension*. All elements in the tensor should be trainable parameters. The result of the multiplication is a matrix $M_i \in \mathbb{R}^{B \times C}$ for each of the feature vectors. Now, the L_1-distance between rows of *different* matrices M_i is computed, and then a negative exponential is applied. This means that we obtain a real number: $c_b(\boldsymbol{x}_i, \boldsymbol{x}_j) = \exp\left(-\|M_{i,b} - M_{j,b})\|_{L_1}\right) \in \mathbb{R}$, where b denotes that this is one of B such real numbers (one per similarity element in the similarity vector). The similiary score vector for sample $\boldsymbol{x}_i$ is then obtained by summing over j [28]:

$$o(\boldsymbol{x}_i)_b = \sum_j c_b(\boldsymbol{x}_i, \boldsymbol{x}_j). \tag{2.26}$$

The similarity vector is then given by:

$$o(\boldsymbol{x}_i) = [o(\boldsymbol{x}_i)_1, o(\boldsymbol{x}_i)_2, \ldots, o(\boldsymbol{x}_i)_B]\,. \tag{2.27}$$

This vector is then appended to the flattened input data and can then be used by the discriminator to determine the similarity between samples in a batch.

Excluding Max-Pooling Layers in CNNs

In convolutional neural networks, pooling layers are frequently used to downscale the input data. In some studies, for instance [7, 8], pooling layers have been used to downsample the data between the convolutional layers. Another study, however, suggests that downsampling in GANs should be done using strided convolution layers since this allows the generator or discriminator to learn their downsampling on their own [29]. There is no consensus in the literature on which of those approaches that should be used, but it is believed that excluding pooling layers in GANs can lead to a more stable training process. Therefore, it is often recommended to perform downsampling using strided convolution instead of max-pooling.

2.4 Evaluation of Generative Models

To this day, there is no consensus on how data synthesized by generative models should be evaluated [7]. Still, the most common method to evaluate the success of GANs is to use a human annotator to judge if the generated samples are sufficiently realistic [4]. This is feasible in the case of images, but not entirely unquestionable: the motivation between annotators may

vary and the quality between the judgments may differ. For instance, it has been shown that annotators given feedback about their mistakes tend to classify a larger number of images as fake [28]. Other methods to evaluate synthetic images exist, such as computing the so-called *inception score*, where a neural network called Inception [30] is used to classify the images. The predicted class probabilities of an observation are summarized into a score, which can be used as a measure of quality: a high-quality image should be classified with high probability, and this is reflected in the inception score. For non-image data, the task is more difficult. It has been proposed that a combination of different evaluation metrics should be used to evaluate the synthesized data [4].

In the following sections, two evaluation metrics that can be used to evaluate synthetic data from a GAN are presented. The metrics are two-samples tests: tests used to determine if two samples are drawn from the same statistical distribution. The evaluation metrics presented here are furthermore differentiable, meaning that they in addition to two-sample tests can be used as loss functions to train generative models.

2.4.1 Maximum Mean Discrepancy

Maximum Mean Discrepancy (MMD) is an integral probability metric proposed in a 2012 article [31]. The metric has proven to be well-suited to evaluate the quality of generated GAN samples in high-dimensional data [32]. The method can be used to test whether samples from two probability distributions, p and q, are drawn from the same distribution. In the case of MMD, this is done by finding a smooth function on which the observations from p and q differ so that the differences in the probability mass functions can be visualized [31]. The test statistic is the difference between the mean values of the function on the two samples: if this difference is large, the samples are likely drawn from different distributions and when it is small, they are more likely drawn from the same distribution.

Formally, maximum mean discrepancy is defined as follows [31]: let $(\mathscr{X}, d)$ be a metric space and let p and q be two Borel probability measures defined on $\mathscr{X}$. Furthermore, let $\mathscr{F}$ be a class of functions such that $f : \mathscr{X} \to \mathbb{R}$. MMD is then defined as:

$$\mathrm{MMD}[\mathscr{F}, p, q] = \sup_{f \in \mathscr{F}} \left(\mathbb{E}_x f(x) - \mathbb{E}_y f(y) \right). \tag{2.28}$$

A biased estimate of MMD is obtained by replacing the expectations in Equation 2.28 with the population means computed from two samples X and Y:

$$\mathrm{MMD}_b[\mathscr{F}, p, q] = \sup_{f \in \mathscr{F}} \left(\frac{1}{m} \sum_{i=1}^{m} f(x_i) - \frac{1}{n} \sum_{i=1}^{n} f(y_i) \right). \tag{2.29}$$

The choice of the function class $\mathscr{F}$ is the unit ball in a reproducing kernel Hilbert space (RKHS) $\mathscr{H}$. The theoretical details will not be covered in this section, but the interested reader is recommended to read the full theoretical motivation behind the choice of function class in

the original paper [31]. However, it should be noted that the class do have some interesting properties that can intuitively motivate the choice as a function class for the MMD metrics. Let $\mathscr{H}$ be a Hilbert space of functions mapping a non-empty set $\mathscr{X}$ to $\mathbb{R}$. An interesting property of an RKHS is that if two functions, $f \in \mathscr{H}$ and $g \in \mathscr{H}$, are close in the norm of $\mathscr{H}$, then $f(x)$ and $g(x)$ are close for all $x \in \mathscr{X}$ [33]. Using for instance a Gaussian kernel, it happens that MMD is zero if and only if the distributions are identical.

For practical purposes, it is only necessary to know how to compute the test statistic. If x and x' are independent random variables with distribution p, y and y' are independent random variables with distribution q, and $k(\cdot,\cdot)$ a continuous kernel, the squared population MMD is defined as [31]:

$$\mathrm{MMD}^2[\mathscr{F}, p, q] = \mathbb{E}_{x,x'} k(x, x') - 2\mathbb{E}_{x,y} k(x, y) + \mathbb{E}_{y,y'} k(y, y'). \tag{2.30}$$

The corresponding unbiased empirical estimate, which can be computed from two samples X and Y, is defined as:

$$\mathrm{MMD}_u^2[\mathscr{F}, X, Y] = \frac{1}{m(m-1)} \sum_{i=1}^{m} \sum_{j \neq i}^{m} k(x_i, x_j) + \frac{1}{n(n-1)} \sum_{i=1}^{n} \sum_{j \neq i} k(y_i, y_j) - \frac{2}{mn} \sum_{i=1}^{m} \sum_{j=1}^{n} k(x_i, y_j). \tag{2.31}$$

A simple choice of kernel is the Gaussian radial basis function (RBF) kernel. The kernel is a popular choice in various kernelized learning algorithms and defined as follows:

$$k(x, x') = e^{-\frac{1}{2\sigma^2}\|x - x'\|_2^2}, \tag{2.32}$$

where σ is the standard deviation of the kernel. There exists various strategies to select σ. The most common choice in practical applications is to let σ equal the median pairwise distance between the joint samples X and Y [32].

Since MMD is estimated using a finite sample with some variance, the estimated value is not necessarily zero although the samples come from the same distribution. Furthermore, the estimate is not necessarily non-negative, although it estimates a squared metric. The reason for this is that, as can be seen in Equation 2.31, the terms where $i = j$ are removed. According to the original MMD paper, the reason for removing those terms is to remove spurious correlations between observations [31]. Furthermore, the authors state that this removal is necessary to make the estimator unbiased. Practically, this means that slightly negative values of the MMD estimate can be obtained.

2.4.2 Energy Distance

Energy statistics, commonly called $\mathscr{E}$-statistics, is a class of discrepancy measures based on the idea of Newton's gravitational potential energy between two heavenly bodies [34]. Statistical observations should have zero potential energy only if they originate from the same underlying distribution. The distance measure has been successful in testing high-dimensional multivariate samples for equal distribution. There exist several similar measures based on the idea of potential energy, but the most popular version was first introduced by G. J. Székely [35]. Let X and X' be two independent random vectors with the same cumulative distribution (cmf) P and Y and Y' two independent random vectors with the same cmf Q. The squared energy distance between the random vectors is then defined as [36]:

$$D^2(P,Q) = 2\mathbb{E}\|X-Y\|_2 - \mathbb{E}\|X-X'\|_2 - \mathbb{E}\|Y-Y'\|_2 \geq 0, \tag{2.33}$$

where equality holds only if X and Y are identically distributed. This inequality, the *basic energy inequality*, was formally proved in a technical report by G. J. Székely [37], and the details are omitted here. The so-called $\mathscr{E}$-statistics is computed as follows:

$$\mathscr{E}_{n,m}(X,Y) = 2A - B - C, \tag{2.34}$$

where A, B, and C are defined as:

$$\begin{aligned} A &= \frac{1}{nm}\sum_{i=1}^{n}\sum_{j=1}^{m}\|x_i - y_j\|_2 \\ B &= \frac{1}{n^2}\sum_{i=1}^{n}\sum_{j=1}^{n}\|x_i - x_j\|_2 \\ C &= \frac{1}{m^2}\sum_{i=1}^{m}\sum_{j=1}^{m}\|y_i - y_j\|_2. \end{aligned} \tag{2.35}$$

Also for the $\mathscr{E}$-statistic, it holds that $\mathscr{E}_{n,m}(X,Y)$ tends to zero if and only if the samples come from equal distributions.

Similar to maximum mean discrepancy, the energy distance measure is an integral probability metric. Although developed completely independent of each other, the definitions of MMD and energy distance are strikingly similar. For several years, it remained a question whether or not energy distance could be seen as a kernel statistics with a L^2 kernel instead of a Gaussian RBF kernel. In 2012, a study showed that both the energy distance and MMD are members of a much broader family of kernel-based metrics [38].

3 | Methodology

In the following chapter, the methods used to preprocess the data, construct the models, evaluate the synthetic data, and perform the experiments on the synthetic data are presented. In Section 3.1, an overview of how the project has been conducted is given. In Section 3.2, the datasets used to train the generative adversarial networks in this project are described. In Section 3.3, the architectures of all networks constructed in this project are described. This includes both the GAN architectures and the architecture of the recurrent neural network used to classify the multivariate radar time-series. In Section 3.4, the computing platform on which the experiments were run is described as well as some of the external libraries used to build the network and training scripts. Furthermore, the settings used to train the described network architectures are given. Finally, in Section 3.5, the experiments performed on the synthetic multivariate radar time-series data are described.

3.1 Overview

The project was conducted in several steps, to ultimately reach the point where synthetic multivariate time-series data could be evaluated in a classification setting. As a pre-experiment, two different network architectures were developed and trained to generate univariate sinusoidal time-series. Such a simple dataset is interesting to consider for several reasons. Firstly, the networks should easily learn the distribution of such simple data. Therefore this choice of dataset is motivated by the possibility of gaining some intuition for network architectures suitable for time-series data. Secondly, it is easy to visually confirm if the networks manage to generate realistic samples, and if the result of this visual inspection of the samples is in agreement with the result of the two discrepancy metrics to be evaluated: maximum mean discrepancy (MMD) and energy distance (ED). Thirdly, several other studies [5, 7, 8] have trained GANs to generate sinusoidal waves, which allows for a comparison of architectures and results.

The main part of this project consisted of generating synthetic multivariate time-series radar data and evaluate the quality of this synthetic data. This work included finding suitable GAN architectures for the multivariate time-series data, training the models, monitoring the discrepancy metrics during training, and performing several experiments to in-depth examine some aspects of time-series augmentation using synthetic samples. At this stage, when the samples

could not be visually evaluated directly, the understanding of the discrepancy metrics from the pre-experiment phase was indeed useful.

3.2 Description and Preprocessing of Datasets

In the following sections, the datasets will be presented. In the case of the sinusoidal dataset, the relevant parameters needed for recreating such a simulated dataset are given and in the case of the multivariate radar time-series dataset, the preprocessing of the dataset is described.

3.2.1 Sinusoidal Time-Series

The dataset was generated by randomly selecting parameters for amplitude (a), period time (T), and phase shift (φ). The samples were generated using the following equation:

$$\boldsymbol{y} = a \sin\left(\frac{2\pi}{T}\boldsymbol{x} + \varphi\right). \tag{3.1}$$

Note that the time steps $\boldsymbol{x}$ is a vector and so is the resulting vector $\boldsymbol{y}$. The amplitude a was uniformly selected from the interval $[0.1, 0.9]$, the period time T from the interval $[2, 8]$, and the phase shift φ from the interval $[-\pi, \pi]$. The sampling frequency was set to 4 samples per second and the sampling interval 10 seconds, yielding generated sequences of in total 40 time steps.

The generated training dataset consists of $10,000$ samples and the test dataset used for evaluation consists of $3,000$ samples.

In Figure 3.1, some examples of generated sinusoidal waves can be found. As can be seen, due to the relatively low sampling frequency, some waves appear deformed. For a generative model, this deformation should not pose any problems since the model (ideally) should be able to reproduce any distribution. For evaluation purposes, however, smaller period times than $T = 2$ was not included in any of the datasets since it can be difficult to distinguish the periodical properties of such sinusoidal waves due to the low sampling frequency.

3.2.2 Radar Tracker Time-Series

The radar tracker dataset was provided by Saab AB and is a multivariate time-series dataset. All observations are recorded using the same type of radar system, but not necessarily recorded at the same location and time. The time-series are of varied length, ranging from about 1 to 600 time steps. The raw dataset is heavily unbalanced, containing a much larger number of bird tracks compared to UAV tracks.

Each multivariate time-series correspond to a target tracked by the radar. Each such target has been classified as either “bird” or “UAV”. Each sample consists of about 20 numerical features, describing the movement of the tracked object. Examples of features present in the dataset

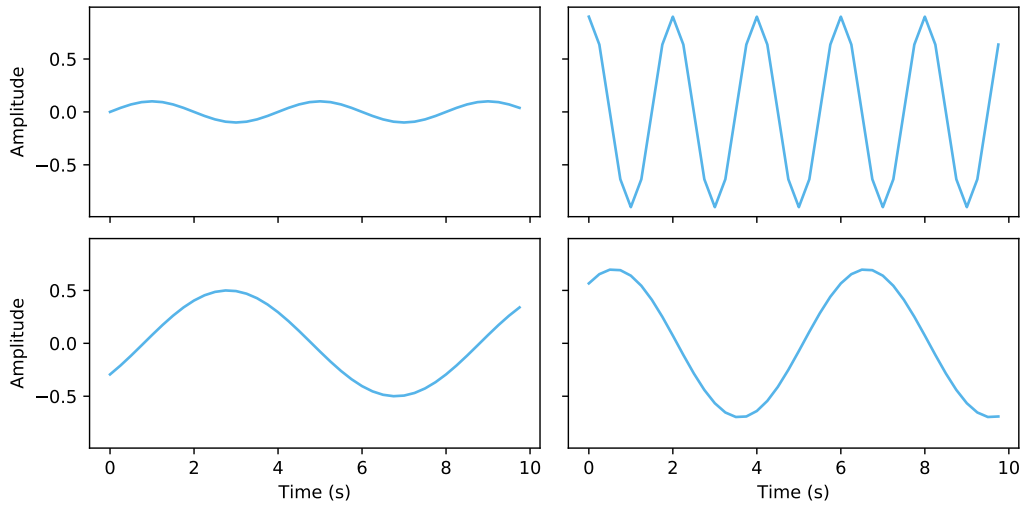


Figure 3.1: Examples of observations from the simulated sinusoidal time-series dataset used as training data for two different GAN architectures. The sinusoidal waves were simulated by randomly selecting amplitudes from the interval $[0.1, 0.9]$, period times from the interval $[2, 8]$, and phase shifts from the interval $[-\pi, \pi]$. The sampling frequency was set to 4 samples per second and the sampling interval was 10 seconds, meaning that each generated sequence was 40 time steps in length.

are three-dimensional velocity, radar cross-section (RCS), and different maneuver parameters. In the original dataset, fixed-wing (FW) UAVs and rotary wing (RW) UAVs are separated. In a previous thesis, aiming to perform classification of bird and UAV tracks using recurrent neural networks, no improvements in accuracy was observed when treating those types of UAVs as separate classes [3]. Therefore, the FW and RW samples were treated as one class in the previous thesis. Although the FW samples did not seem to affect the classification accuracy in the previous thesis, it cannot be ruled out that the performance of the trained GAN would be affected by the fact that different types of UAVs with slightly different properties are treated as a single class. Furthermore, the number of FW samples in the raw dataset is small compared to the number of RW samples. For this reason, the decision was made to exclude the FW samples from the dataset, resulting in a dataset with two classes.

The preprocessing of the dataset included removing time-series shorter than 20 time steps and sequences recorded at a too large distance. A common problem when training neural networks on time-series is that the sequences preferably should be of equal length. The reason for this is that batch training, used in most applications, requires all samples in each batch to be of equal length. For many real-world datasets, the sequence lengths vary. A solution to this is to pad the sequences with for instance zeros so they become of equal length. Since the length of the remaining time-series at this point ranged from 20 time steps to about 600 time steps, with a median of 43 time steps for birds and 87 time steps for UAVs, padding the sequences to match the length of the longest sequence would mean that most sequences would be heavily padded. Instead, the tracks were split into shorter sequences and treated as independent samples. For the bird tracks, only a single such subtrack was extracted from each raw time-series since there was a surplus of bird tracks. For the UAV tracks, however, as many subtracks as possible were

extracted from each track. The length of those subtracks was set to 20 time steps. The reason for keeping the sequences short is that most of the UAV samples were recorded to tune the signal processing system in the radar, and hence the UAVs are flown in similar patterns in the recorded data. To avoid overtraining the networks on the flight patterns, shorter sequences are preferable, and hence the length 20 time steps was selected. After this preprocessing step, 7,177 bird tracks and 3,177 UAV tracks remained and were available as training data for the models.

When a balanced dataset was needed, a random subset of the bird samples was selected to match the number of UAV samples in the dataset.

3.3 Neural Network Architectures

In the following sections, the different neural network architectures used in this project are presented. The architectures include the GAN architectures used to generate both sinusoidal data and tracker data and the architecture of the RNN classifier used to perform binary classification of birds and UAVs.

3.3.1 Generation of Sinusoidal Time-Series

To generate sinusoidal time-series, two different architectures were constructed. The first one is similar to an architecture used in a previous study [7] and has a generator consisting of a bidirectional LSTM network and a discriminator consisting of a convolutional network. This architecture will be referred to as the s-BiLSTM-CNN GAN (s for "sine") throughout this thesis. Both the generator and the discriminator have a fully-connected layer as the last layer. In the generator, the output of the last layer is activated by a tanh function, meaning that the output is constrained at $[-1,1]$. This activation is used since the amplitudes of the sinusoidal waves in the training data are constrained at the same interval. In the discriminator, a sigmoid activation function is applied to the output of the last layer since the network should output the probability of an input sample being real. Since the exclusion of a max-pooling layer is said to improve training stability [29], strided convolution is used to downsample the time-series in the discriminator. Furthermore, LeakyReLU is included instead of ordinary ReLU. The difference is that while ReLU is non-zero if and only if the input is non-negative, LeakyReLU is non-zero even for negative input. The idea is that LeakyReLU should allow a small, negative gradient. The proportion of the gradient allowed is called *slope* and is set as a hyperparameter. A minibatch discrimination layer is also included in the discriminator to encourage the generator to generate more diverse samples. Lastly, a dropout layer is included in the discriminator to avoid overfitting. The details of the s-BiLSTM-CNN GAN architecture can be found in Table 3.1.

The second architecture is based on an architecture used in previous work [5] and consists of an LSTM generator and an LSTM discriminator. This architecture will be referred to as the s-LSTM GAN architecture. In this case, the discriminator does not perform downsampling as in the s-BiLSMT-CNN GAN case, but instead the output of the LSTM layer is a tensor describing the

probability of each time step being real. The decision for the total sequence is determined by a majority vote, similar to the case shown in Figure 2.8. Also in this architecture, the last layer of both the generator and the discriminator is fully-connected. In the generator, the output of the fully-connected layer is activated with a tanh activation function. In the discriminator, the output is fed through a sigmoid activation function. A detailed description of the s-LSTM GAN architecture can be found in Table 3.2.

Table 3.1: The architecture of the s-BiLSTM-CNN GAN used to generate sinusoidal waves. The generator consists of a bidirectional LSTM network with two layers and 50 hidden units. The output of the LSTM layer is fed into a fully-connected layer with a tanh activation function since the amplitude of the sine waves should be constrained between $[-1, 1]$. The discriminator consists of one convolutional layer with LeakyReLU and dropout, and of a fully-connected layer with a sigmoid activation.

Layer	Properties	Output size
Generator		
Input	-	50×40
BiLSTM	Bidirectional, 50 hidden units, 2 layers	$50 \times 40 \times 100$
Fully connected	Tanh activation	50×40
Discriminator		
Input	-	50×40
Convolutional	10 kernels, kernel size 4, stride 2	$50 \times 10 \times 19$
LeakyReLU	Slope parameter 0.2	$50 \times 10 \times 19$
Flatten	-	50×190
Minibatch discrimination	16 hidden nodes, 3 outputs	50×193
Fully connected	No activation	50×1
Dropout	Probability 0.2	50×1
Sigmoid	-	50×1

Table 3.2: The architecture of the s-LSTM GAN used to generate sinusoidal waves. The generator consists of an LSTM network with one layer and 100 hidden units. The last layer is a fully-connected layer with a tanh activation function since the output should be constrained between $[-1, 1]$. The discriminator is identical, but with a sigmoid activation function in the final fully connected layer and an averaging operation is performed over the full sequence to determine the total probability of the sequence being real.

Layer	Properties	Output size
Generator		
Input	-	50×40
LSTM	100 hidden units, 1 layer	$50 \times 40 \times 100$
Fully connected	Tanh activation	50×40
Discriminator		
Input	-	50×40
LSTM	100 hidden units, 1 layer	$50 \times 40 \times 100$
Fully connected	Sigmoid activation	50×40
Averaging	-	50×1

3.3.2 Generation of Radar Tracker Data

Due to the more complex structure of the multivariate radar time-series dataset, more complex architectures were needed compared to the case of the sinusoidal waves. However, it was desirable to keep the architectures as similar as possible to the architectures already presented, since vastly different architectures might show different convergence patterns, and hence the insights gained by studying the generation of sinusoidal waves might not be applicable.

Table 3.3: The architecture of the conditional t-BiLSTM-CNN GAN used to generate multivariate tracker time-series. The generator consists of a bidirectional LSTM network with two layers and 150 hidden units. The output of the LSTM layer is fed into a fully-connected layer without activation. The discriminator consists of four convolutional layers with LeakyReLU and dropout, and of a fully-connected layer with a sigmoid activation.

Layer	Properties	Output size
Generator		
Input	-	50×20
BiLSTM	Bidirectional, 150 hidden units, 2 layers	$50 \times 20 \times 300$
Fully connected	No activation	50×20
Discriminator		
Input	-	50×20
Convolutional	3 kernels, kernel size 5, stride 1	$50 \times 3 \times 16$
LeakyReLU	Slope parameter 0.2	$50 \times 3 \times 16$
Convolutional	5 kernels, kernel size 3, stride 1	$50 \times 5 \times 14$
LeakyReLU	Slope parameter 0.2	$50 \times 5 \times 14$
Convolutional	8 kernels, kernel size 2, stride 2	$50 \times 8 \times 7$
LeakyReLU	Slope parameter 0.2	$50 \times 8 \times 7$
Convolutional	12 kernels, kernel size 3, stride 2	$50 \times 12 \times 3$
LeakyReLU	Slope parameter 0.2	$50 \times 12 \times 3$
Flatten	-	50×36
Minibatch discrimination	16 hidden nodes, 5 outputs	50×39
Fully connected	No activation	50×1
Dropout	Probability 0.2	50×1
Sigmoid	-	50×1

The first architecture used to generate multivariate radar time-series data is based on the s-BiLSTM-CNN architecture presented in Table 3.1, but with some modifications. It will be referred to as the t-BiLSTM-CNN GAN (t for “tracker”) to avoid confusion with the corresponding sinusoidal architecture. Firstly, a deeper convolutional network is needed in the discriminator to handle a more complex input. Four convolutional layers were found suitable, and the two last convolutional layers perform strided convolution to downsample the time-series. LeakyReLU is used in-between the convolutional layers in the discriminator and a minibatch discrimination

layer is included before the fully-connected layer. The generator consists of a bidirectional LSTM network with two layers and 150 hidden units. The output of the discriminator is still activated using a sigmoid function, but in the generator, tanh activation cannot be used since the synthetic output should *not* be constrained between $[-1, 1]$. Therefore, no activation is applied to the output. Furthermore, both the generator and the discriminator are conditioned on the class labels. Details on this model, the conditional t-BiLSTM-CNN GAN, can be found in Table 3.3.

The LSTM-based architecture can be found in Table 3.4 and is similar to the architecture used to generate synthetic sinusoidal waves. It will be referred to as the t-LSTM GAN architecture. In both the generator and discriminator, two LSTM layers are used instead of a single one as in the case of the sinusoidal waves. Furthermore, 350 hidden units are used in the generator and 150 in the discriminator, compared to 100 in both the generator and discriminator in the sinusoidal case. Also in this network, both the generator and the discriminator are conditioned on the class labels.

Table 3.4: The architecture of the conditional t-LSTM GAN used to generate multivariate tracker time-series. The generator consists of an LSTM network with two layers and 350 hidden units. The last layer is a fully-connected layer with no activation. The discriminator is identical, but with fewer hidden nodes (150) and a sigmoid activation function in the final fully connected layer. Furthermore, an averaging operation is performed over the full sequence to determine the total probability of the sequence being real.

Layer	Properties	Output size
Generator		
Input	-	50×20
LSTM	350 hidden units, 2 layers	$50 \times 20 \times 350$
Fully connected	No activation	50×20
Discriminator		
Input	-	50×20
LSTM	150 hidden units, 2 layers	$50 \times 20 \times 150$
Fully connected	Sigmoid activation	50×20
Averaging	-	50×1

3.3.3 Classification of Radar Tracks

To evaluate the classification performance of the synthetic data, a classifier is needed. Since this thesis partly builds upon work from a previous thesis [3] where the same dataset was used, the same type of classification network will be used in this thesis. The network is simple and consists of a recurrent unit: a gated recurrent unit (GRU). The complete architecture can be found in Table 3.5. Two changes have been made in the network structure. Firstly, a larger batch size has been used. In the previous thesis, a batch size of 1 was used and the best classification results were obtained using the full-length tracks. In this thesis, the batch size in the classification tasks is set to 5. Using a batch size of 1 would be infeasible in this project due to the massive computational time such runs would take. Furthermore, no training was done using the full-

length tracks but only using tracks of lengths 20, as motivated in Section 3.2.2. Lastly, the reason for not including a softmax activation function in the fully-connected layer is that in the computational framework used (PyTorch, see Section 3.4.1), the standard implementation of the cross-entropy loss function expects raw (non-normalized) input.

Table 3.5: The architecture of the recurrent neural network used for classification of multivariate radar time-series, developed in a previous thesis [3].

Layer	Properties	Output size
Input	-	5×20
GRU	25 hidden units, 2 layers	$5 \times 20 \times 25$
Dropout	Probability 0.2	$5 \times 20 \times 25$
Fully connected	No activation	5×2

3.4 Training the Neural Networks

In the following sections, the training processes of the generative adversarial networks and the classification networks are described. Firstly, in Section 3.4.1, the computing platform used to carry out the experiments is briefly described. In Section 3.4.2, the hyperparameters for the networks are given and some training settings.

3.4.1 Computing Platform

All models were implemented in Python 3.7 using PyTorch 1.2 and CUDA 10.2. The experiments were performed on a system running on Unix CentOS 7 with an Intel Xeon Silver 4210 (2.20 GHz) CPU and an Nvidia Quadro RTX 6000 GPU. Several Python open-source libraries were used as a basis for some of the implemented functions. The implementation of maximum mean discrepancy and energy distance was based on the implementation in the torch library `torch-two-samples` [39] but modified to be compatible with tensor computations following the library `opt-mmd` accompanied with a 2016 paper on generative models and maximum mean discrepancy [40]. Furthermore, the implementation of minibatch discrimination was based on code accompanying a 2019 article about using GANs to generate realistic sinusoidal waves and ECG data [7].

3.4.2 Hyperparameters and Training Settings

The selection of hyperparameters in any machine learning task is of great importance for the performance of the trained models [14]. The hyperparameters can both affect the time- and memory consumption during training, as well as the generalizability of the model when tested on held-out samples. The standard ways of selecting hyperparameters are to select them either manually or automatically. Selecting them automatically often includes some type of grid search or random search, where a large number of combinations of hyperparameters are tested to find the optimal set. In this project, the number of hyperparameters is large and the networks

take long to train, so therefore, such an automatic tuning would be infeasible. Therefore, a step-wise manual selection of hyperparameters was utilized. This comes with the advantage that some intuition for how the hyperparameters affect the network performance can be gained. The procedure used was simple: some basic set of parameters were obtained by studying relevant papers. From that starting point, all hyperparameters but one was kept constant and the last hyperparameter was varied. Each model was evaluated by training it several times and monitoring maximum mean discrepancy (MMD) and energy distance (ED). After each epoch, the MMD and ED between a generated synthetic sample and an equally large withheld test set were computed. The test set was 10% of the available data. By considering the smallest MMD and ED scores obtained, different parameter settings could be compared to each other. From this, a set of hyperparameters could be selected.

The chosen architectural hyperparameters for all architectures can be found in Section 3.3. The training hyperparameters used when training both the GANs on sinusoidal data and the multivariate radar dataset can be found in Table 3.6. As can be seen, the choice of training hyperparameters are the same for both the sinusoidal and the multivariate radar datasets.

The training parameters for the GRU classification network was not tuned as carefully, as this was done in a previous thesis [3]. The parameters used can be found in Table 3.7.

Table 3.6: The training parameters used to train the s-BiLSTM-CNN GAN, the s-LSTM GAN, the t-BiLSTM-CNN GAN, and the t-LSTM GAN to generate sinusoidal waves and multivariate radar time-series.

Parameter	Value (BiLSTM-CNN)	Value (LSTM)
Batch size	50	50
Rounds (gen.)	1	1
Rounds (disc.)	3	1
Learning rate (gen.)	0.0002	0.0002
Learning rate (disc.)	0.0002	0.0002
Optimizer	Adam ($\beta_1 = 0.5$, $\beta_2 = 0.999$)	Adam ($\beta_1 = 0.5$, $\beta_2 = 0.999$)
Loss	Binary cross-entropy	Binary cross-entropy

Table 3.7: The training parameters used to train the GRU classifier for classification of multivariate radar time-series. All parameter values except the batch size are consistent with the parameters used in a previous master thesis where classification of the radar tracks was done [3]. Note that the training folds refers to the iterations of MCCV cross-validation, and not the folds in ordinary k-fold cross-validation. In some of the experiments, 10 folds were used but when possible, 20 folds were preferred since the variability between runs was large at times.

Parameter	Value (GRU classifier)
Batch size	5
Epochs	25
Learning rate	0.0005
Optimizer	Adam ($\beta_1 = 0.9$, $\beta_2 = 0.999$)
Loss	Cross-entropy
Training folds	10 or 20

3.5 Experiments Performed on Synthetic Radar Time-Series

In the case of the sinusoidal time-series dataset, the aim was simply to be able to generate realistic-looking synthetic data and to gain some understanding of how well the metrics performed relatively the visual quality of the samples generated. No further experiments were conducted on the generated data. For the multivariate radar time-series data, however, several experiments to evaluate the quality of the generated data were conducted.

The ultimate usage of the data generated in this thesis is to augment an existing real-world dataset that is used to train a classifier for multivariate time-series classification. Evaluating the quality of the data by using it as training and/or test data for such a classifier is therefore highly relevant. This can, however, be done in many ways to investigate different aspects of data quality. The experiments performed on the synthetic data aims at investigating those aspects and will be described in the following sections.

3.5.1 The Effect of Sampling Proportions of Conditional Labels

The first experiment considers the effect of utilizing different sampling techniques when sampling the synthetic labels provided to the generator and the discriminator as conditional information during training. In many applications, the class labels provided to the networks during training are uniformly sampled, meaning that the generator will always generate fake samples that are approximately class balanced, independent of the distribution between classes in the training data. If the classes in the training data are heavily unbalanced, one could argue that the fact that the synthetic samples are always class-balanced can aid the discriminator in the decision that the sample is fake. On the other hand, if the synthetic samples are always class-balanced, the generator will get more feedback regarding the quality of the minority class samples compared to if the label proportions are the same as the class proportions in the training data. It is therefore interesting to investigate if there is a difference between synthetic data generated from a generator trained with uniform sampling of the class labels and synthetic data generated from a generator trained using proportional sampling of the class labels. Furthermore, the extreme case is to train a GAN on each of the two data classes separately and then generate a synthetic dataset by combining samples from those two GAN instances. This yields three different ways to train conditional GANs and to generate synthetic data.

The mentioned methods were tested by training each architecture (t-BiLSTM-CNN GAN and t-LSTM GAN) with the four different sampling techniques (equal, proportional, only birds, and only UAVs). The lowest MMD and ED scores obtained for each run were monitored and averaged over five runs. To examine if there was any difference in classification performance between classifiers trained on synthetic data from the different generators, the GAN models trained with the different sampling techniques were treated as separate models also in the experiments described in the upcoming sections. It should be noted that while the generators trained using equal and proportional sampling can be used directly to generate a synthetic dataset, this is

not the case for the generators trained with only birds and only UAVs. Those generators were used to generate samples from each class separately, which were then combined into a synthetic dataset. In the upcoming sections, this method of generating synthetic data will be referred to as the *separate* sampling method.

Another relevant aspect is how the generator model should be selected. During training, the generator weights are saved after each epoch so the generator model from any epoch can easily be restored. In this project, two different discrepancy metrics (MMD and ED) have been used to monitor the training progress. Therefore, it is natural to base the selection of generator models on those metrics. It turned out that in many cases, MMD and ED are minimized at different epochs of training. It is not unlikely that the inclusion of additional metrics to MMD and ED would result in an even larger ambiguity. So how should the generator model be selected if the MMD and ED result in different models? Since both metrics were to be evaluated, the natural answer was to select both models and compare their performance. As will be seen, all results from the experiments described in the upcoming sections will be presented for datasets generated by models selected using the minimal MMD and for datasets generated by models selected using the minimal ED separately.

3.5.2 The Quality of the Synthetic Samples from a Classification Perspective

There are two main quality aspects of the synthetic data that needs to be evaluated: sample quality and sample diversity. In this section, three different experiments aiming at evaluating both of those aspects are described.

Train on Real, Test on Synthetic

How can the sample quality of multivariate time-series data be evaluated, without visually inspecting the samples? A natural approach is to examine how well a classifier trained on real data generalizes to a test set consisting of only synthetic samples. Following [5], the approach will be called *train on real, test on synthetic* (TRTS). This approach serves as a measure of how well the GAN manages to generate realistic samples. The test is unaffected by whether or not the synthetic data suffers from mode collapse since it is trained on real data where the full distribution is represented.

In this project, TRTS was performed by training a GRU-classifier (see Section 3.3.3) for 25 epochs on a real, balanced dataset and then testing the performance of the classifier on an equally large but fully synthetic dataset. The test datasets were generated using the generators described in the previous sections. The generators are from two different architectures (t-BiLSTM-CNN GAN and t-LSTM GAN), trained using different sampling techniques (equal, proportional, and separate), and selected using two different criteria (MMD and ED). In some cases, MMD and ED selected the same model, so in total, 10 different types of synthetic test datasets were generated. For each type of synthetic test set, the classifier was trained 20 times on real data and each time,

a new synthetic test set was generated by the corresponding generator. The accuracy, precision, recall, and F1-scores were then computed by averaging over those 20 runs. For reference, the classifier was also trained on a real, balanced dataset using 20 iterations of Monte-Carlo cross-validation (MCCV). MCCV was performed by randomly splitting the dataset into a training set (90% of the total samples) and a test set (10% of the total samples). Also in this case, the classifier was trained for 25 epochs. The corresponding evaluation metrics were computed for this reference case too.

Train on Synthetic, Test on Real

Another aspect of synthetic data quality is if the samples are diverse enough. If the synthetic data suffers from a diversity loss compared to the training data, there is a covariate shift in the synthetic data. One method to examine if this is the case is to train a classifier on a purely synthetic dataset and test on real data. This has been done in several previous studies [4, 5] and is called *train on synthetic, test on real* (TSTR). The intuition behind TSTR is that if the GAN has failed to capture the full diversity of the underlying distribution, then a classifier trained on synthetic data will perform worse when tested on real data (in which the full distribution is represented) compared to if the classifier would be trained on real data.

The practical procedure for performing this experiment was similar to the previous experiment. The generators described in the previous section were used to generate 10 different types of *training* datasets. A GRU-classifier (see Section 3.3.3) was trained for 25 epochs on the synthetic datasets and then tested on a balanced real dataset. The synthetic and real datasets used were equally large (3,177 samples per class). Also in this case, the training procedure was repeated 20 times per type of synthetic dataset and then the accuracy, precision, recall, and F1-score were computed by averaging over the 20 runs. For reference, the classifier was trained on purely real data using 20 iterations of MCCV. The average evaluation metrics were computed by averaging over the MCCV iterations.

To generate synthetic data for augmentation purposes, the TSTR score is more relevant to consider since it reflects the ability of the synthetic data to be used to train models. However, for quality evaluation purposes, the TRTS score is important to consider.

Train on a Mixture, Test on a Mixture

When both extremes have been tested, a remaining natural question is: what would happen if the classifier is trained on a mixture of real and synthetic data, and then tested on a mixture of real and synthetic data?

For reference, the classifier was first trained using Monte Carlo cross-validation with 20 iterations on the real, balanced dataset. MCCV was performed by randomly splitting the dataset into a training set (90% of the total samples) and a test set (10% of the total samples). At each iteration,

the classifier was trained for 25 epochs. Instead of computing the evaluation metrics such as accuracy and recall, a confusion matrix was generated after each MCCV iteration and then averaged over. The reason for presenting the results in a confusion matrix instead of giving the numerical evaluation metrics is that the classification result for the second part of this experiment should be presented for real and synthetic samples separately and using a visual presentation was found to be more viable.

To test the classification performance of the classifier on both real and synthetic data, a similar approach was used. Also here, MCCV with 20 iterations was used to obtain average metrics. A classifier was trained on each of the datasets described in previous sections and also here the generator models selected using MMD and ED were considered separately. The proportion of added synthetic data was set to 100%, meaning that the datasets consisted of as many real samples as synthetic samples. The classifier was trained for 25 epochs per MCCV iteration for all datasets. For this part of the experiment, the results were summarized in slightly unconventional confusion matrices, where the real and synthetic samples were treated separately with four true data classes (bird real, UAV real, bird synthetic, and UAV synthetic) but only two predicted classes (bird and UAV). Using this representation, it was found to be simple to compare the performance of the classifier on real and synthetic data.

3.5.3 The Proportion of Synthetic Data Versus Classification Accuracy

Another question that needs consideration is: how large proportion of synthetic data should be used for augmenting the training dataset? A classifier, as described in Section 3.3.3, was trained on a mixture of real and synthetic data. All datasets were class-balanced. The proportion of synthetic data added to the real data was varied between 0% (no synthetic data) and 1000% (ten times more synthetic data observations than real data observations). The trained classifier was then evaluated using a withheld test set, consisting of 10% of the observations in the full real dataset. Note that no synthetic data was included in the test set since the model performance on real data was of interest in this experiment. In this experiment, MCCV with 10 iterations was used for each proportion. The classifier was trained for 25 epochs in each training phase.

4 Results

In the following chapter, the results of the experiments described in Chapter 3 are presented. In Section 4.1, the synthetic sinusoidal data generated by the two recurrent GAN architectures is presented. In Section 4.2, the results from the various experiments conducted on the multivariate tracker time-series dataset are presented.

4.1 Generation of Sinusoidal Time-Series

In the following sections, some results from the training of the sinusoidal-generating GANs and some evaluation of the synthetic data are presented.

4.1.1 Comparison of Architectures

As described in Section 3.3, two different architectures were used to generate synthetic sinusoidal data. One architecture is called s-BiLSTM-CNN GAN and the other one is called s-LSTM GAN. Overall, the training processes of both networks were stable and all runs resulted in sinusoidal waves of visually high quality.

There were some notable differences between training the networks. The first one was the time it took to train the networks. The s-BiLSTM-CNN GAN took about four hours to train 1,000 epochs, and the s-LSTM GAN took about three hours to train 1,000 epochs. It should be noted that while the networks were trained for 1,000 epochs to allow the study of the convergence of the networks, fewer epochs were required to reach samples that visually could be verified to be sinusoidal waves. The exact number varied slightly between different runs, but the s-BiLSTM-CNN GAN required about 150 epochs and the s-LSTM GAN required about 200 epochs.

In Figure 4.1, examples of synthetic samples from both architectures during different epochs of training are shown. The samples are generated using the same latent (noise) vector, which allows for a direct comparison of the development of the synthetic samples during training between the architectures. As can be seen, neither network outputs anything similar to sinusoidal waves at epoch 1, but already at epoch 10, the generators have learned to generate data with some periodic behavior. Here, it is clear that the s-BiLSTM-CNN GAN learns faster since the samples from this generator at epoch 10 are better compared to the corresponding samples

from the s-LSTM generator. As can be seen in the figure, the discrepancy metrics maximum mean discrepancy (MMD) and energy distance (ED) strictly improves as the network training progresses. The figure also indicates that as the training progresses and samples of higher visual quality are generated, the MMD and ED values tend to decrease. However, it is also clear from the figure that one should be careful to compare values of the metrics directly: after one epoch, the samples from the s-BiLSTM-CNN GAN better resembles sinusoidal waves compared to the samples from the s-LSTM generator, but they have larger values of both MMD and ED. The discrepancy metrics can say something about how the training progresses, but single such numbers can be misleading and do not necessarily reflect the visual quality.

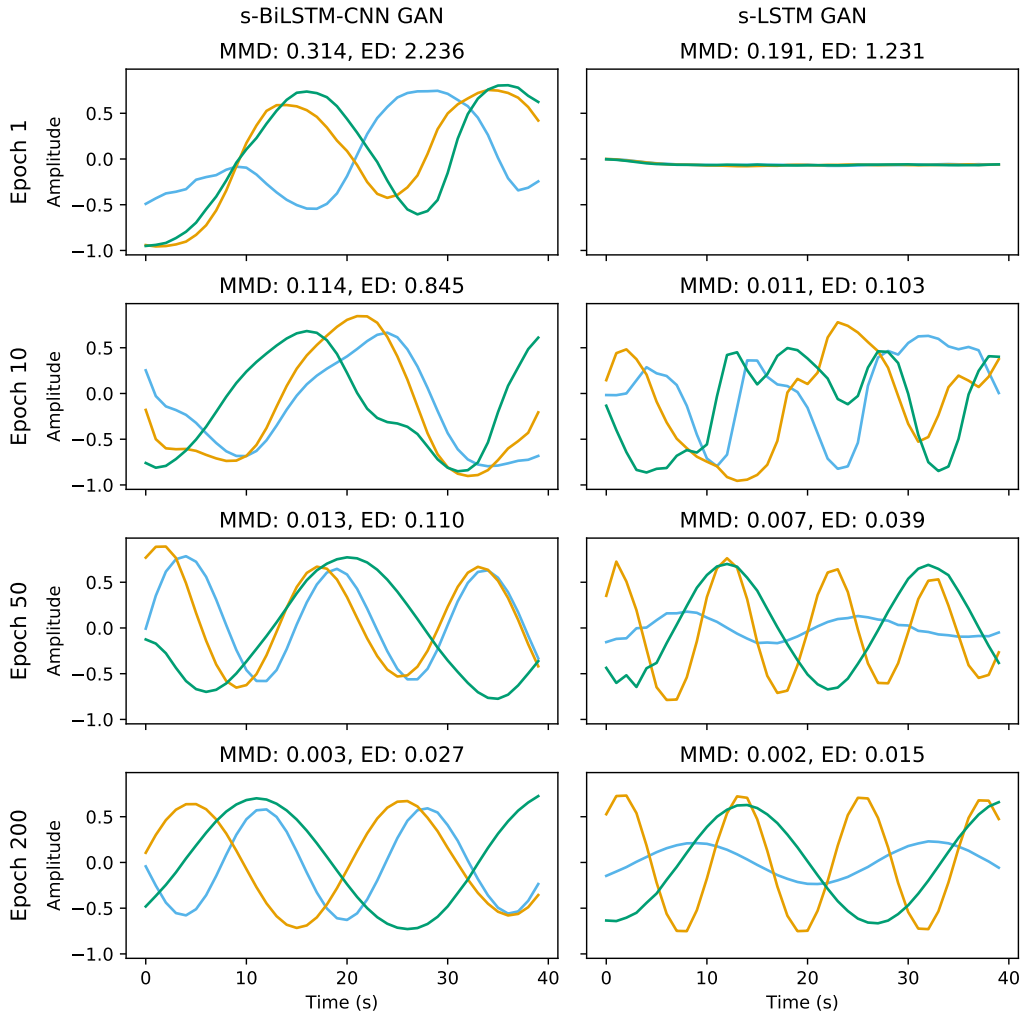


Figure 4.1: Synthetic samples generated by the generators in the s-BiLSTM-CNN GAN and the s-LSTM GAN at four different epochs (after 1, 10, 50, and 200 epochs) of training. As can be seen, both networks gradually learn the distribution of the training dataset and are after 200 epochs able to generate realistic sinusoidal waves. Also, note that both MMD and ED are strictly decreasing as the samples improve. However, a lower MMD or ED does not necessarily correspond to a higher quality of the generated samples, which is visible in the samples generated after the first epoch of training.

The discrepancy metrics as a function of epochs for both architectures can be found in Figure 4.2. Note that both the true values and the moving average over a window of 20 epochs are shown.

As can be seen, both MMD and ED decreases as the training progresses. It is also clear that the s-LSTM GAN architecture stabilizes at lower MMD and ED levels compared to the s-BiLSTM-CNN GAN. Since the scale on the y axis is logarithmic, the fluctuations of the discrepancy metrics at later epochs are small. As can also be seen in the figure, the training process of the s-LSTM GAN seems to be more stable with smaller variances of both MMD and ED. It should be noted that a stable training process does not necessarily correspond to generated samples of high quality, but in general, a more stable training process is of interest to encourage convergence of the network.

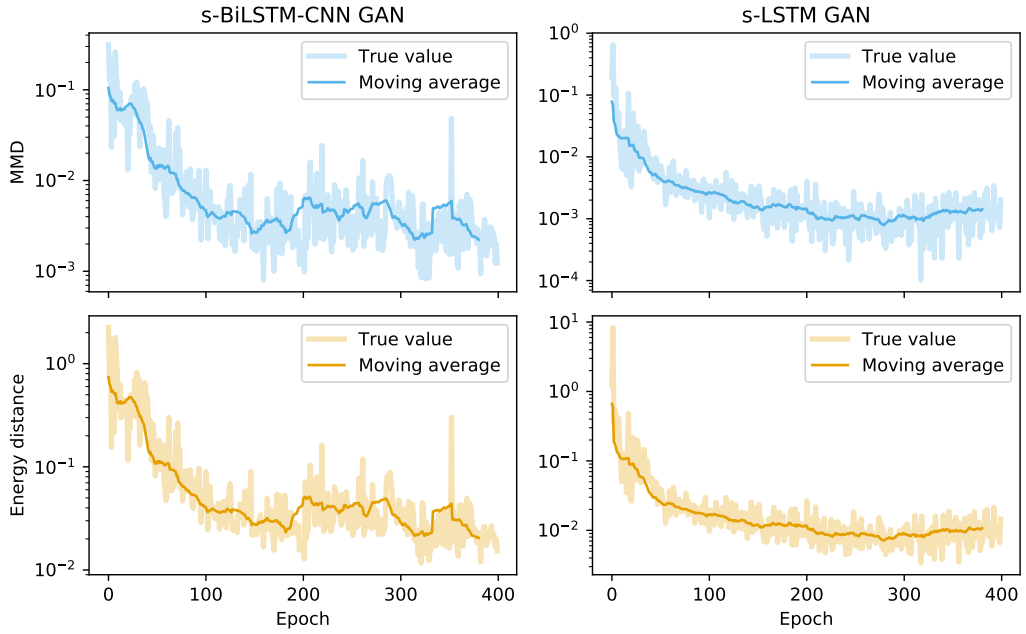


Figure 4.2: Maximum mean discrepancy (MMD) and energy distance (ED) between synthetic samples and test samples from real data across epochs during the training of s-BiLSTM-CNN GAN and s-LSTM GAN on sinusoidal waves. The figure shows both the actual obtained values and the moving weighted average computed over a window over 20 epochs. Note that the scales on the y-axes are logarithmic, so the actual changes in the discrepancy metrics at later epochs are small.

4.1.2 Training Convergence

For the sinusoidal waves, monitoring the training progress is simple since the quality of the generated data can be evaluated by visually inspecting the generated samples after each epoch. But other data, such as multivariate time-series radar data, is not as easily visually inspected. Therefore, a comparison of other methods to monitor the training process is needed. A common method is to follow how the losses of the generator and the discriminator are developing. In contrast to when training other types of deep neural networks, the loss plots obtained during the training of a GAN do not necessarily reflect the quality of the generated samples. Furthermore, losses for different architectures can behave quite differently. To allow for a comparison, both the losses and the discrepancy metrics were monitored during training. The results can be found in Figure 4.3 for the s-BiLSTM-CNN GAN architecture and in Figure 4.4 for the s-LSTM

architecture. In both figures, the generator and discriminator losses (for fake and real images separately) are plotted as well as MMD and ED for each epoch. Note that also in these plots, the moving average is shown together with the true values.

As can be seen in Figure 4.3, both the generator loss and the discriminator loss are quite stable as the training progresses for the s-BiLSTM-CNN architecture. From the loss plot, it is not possible to tell how the training progresses since the losses are relatively constant throughout the epochs. From the MMD and ED plot, however, it is clear that the largest decrease in both MMD and ED is achieved up to about 200 epochs and due to the previously presented results, it is reasonable to assume that this also means that the quality of the synthetic data improves mostly during the 200 first epochs, and then stabilizes. Such information could not be read out from the loss plot.

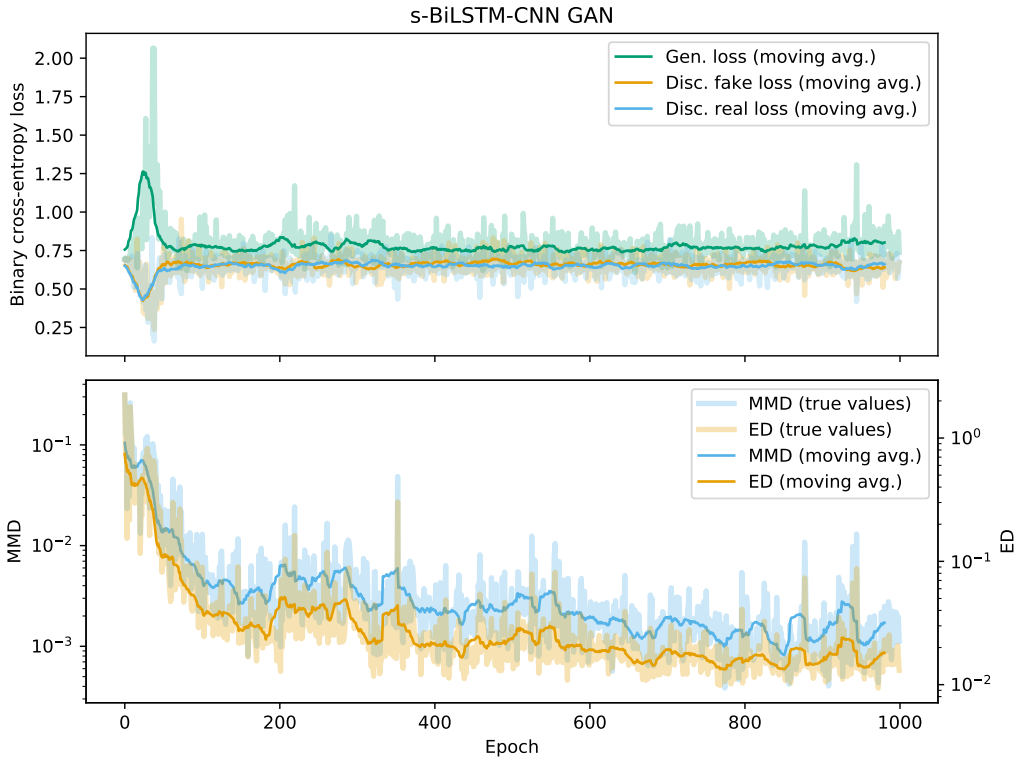


Figure 4.3: Binary cross-entropy loss of the generator and discriminator (split into the loss of real and fake batches) and the discrepancy metrics maximum mean discrepancy (MMD) and energy distance (ED) as a function of epoch for the s-BiLSTM-CNN architecture.

In Figure 4.4, the losses and discrepancy metrics are presented for the s-LSTM GAN architecture. In contrast to the losses for the s-BiLSTM-CNN architecture presented in Figure 4.3, the losses are not roughly constant during the 1,000 epochs of training. As can be seen, the generator loss seems to increase with epochs and the discriminator losses decrease. The moving averages of the discriminator losses seem to be similar, which indicates that the discriminator recognizes real sinusoidal waves as well as synthetic samples. Another notable detail in the discrepancy metrics plot is that both MMD and ED seem to stabilize at lower values for the s-LSTM architecture. This indicates that the synthetic samples from the s-LSTM architecture are more similar to the

samples in the withheld test dataset compared to samples from the s-BiLSTM-CNN architecture, but such differences cannot be verified visually. Furthermore, the scale of the double y-axis is logarithmic so the actual differences are small.

A comment should be made about the dips in the MMD curve in the s-LSTM case. From around 800 epochs, at four occurrences, a drop in the MMD level can be observed. Corresponding drops cannot be seen in the ED curve. At a closer inspection of the values, what happens is that the MMD values at those points are small negative numbers. Naturally, MMD cannot be negative since it is a squared metric, but the *estimate* of MMD can be negative. The reason for this is that the terms $i \neq j$ are excluded in the definition (see Equation 2.31 in Section 2.4.1) to make the estimator unbiased [31]. This means that non-negativity cannot be guaranteed. In this case, the negative values are small (of order 1×10^{-5}) and indicate that low MMD values close to zero have been obtained.

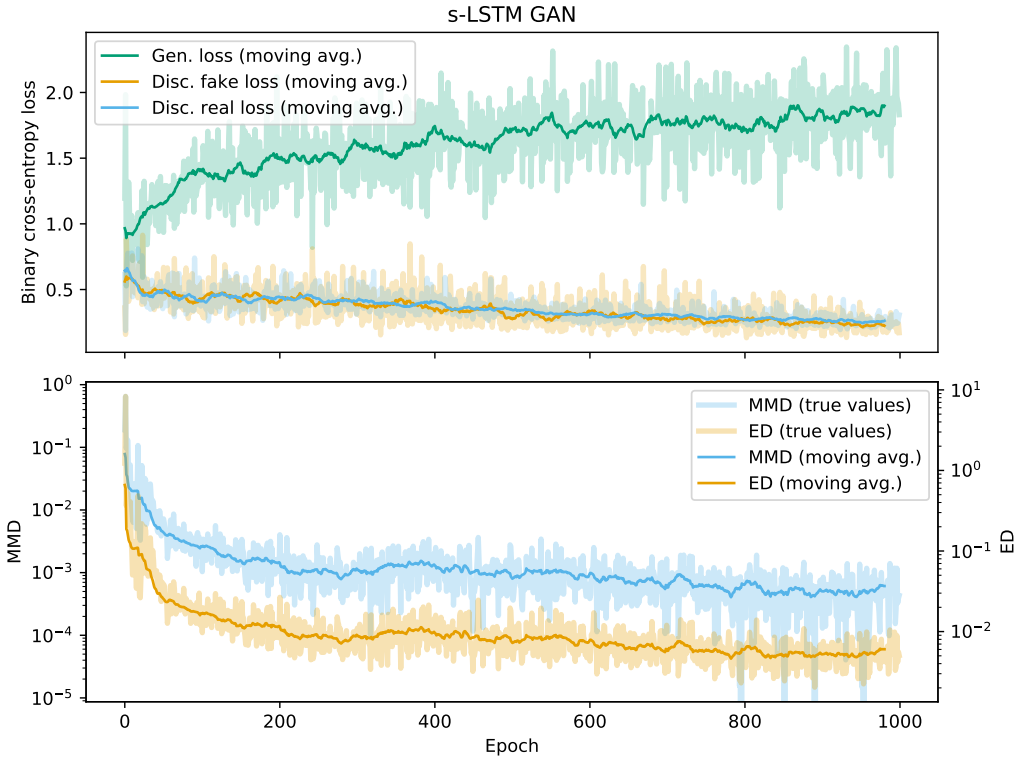


Figure 4.4: Binary cross-entropy loss of the generator and discriminator (split into the loss of real and fake batches) and the discrepancy metrics maximum mean discrepancy (MMD) and energy distance (ED) as a function of epoch for the s-LSTM GAN architecture.

4.2 Generation of Multivariate Radar Time-Series

In the following sections, some results from the training phase of the multivariate time-series-generating GANs are presented as well as the results of the experiments performed on the synthetic data.

4.2.1 The Training Process and Convergence of the Models

Also for the multivariate time-series radar dataset, two different architectures, the t-BiLSTM-CNN GAN and the t-LSTM GAN, were used to generate synthetic data. Both architectures are described in Section 3.3. As was described in Section 3.5.1, eight different networks were trained in total (four of each architecture, with equal, proportional, only birds, and only UAVs sampling), and therefore, it is not possible to include results from the training processes of all networks here. However, a couple of typical examples of the developments of the losses and discrepancy metrics across epochs can be of interest and are therefore included here.

In Figure 4.5, the training losses and the discrepancy metrics from one training of the t-BiLSTM-CNN GAN architecture are presented. In this specific run, proportional sampling was used. As can be seen, the losses do not change much throughout the epochs. The behavior of the losses was consistent for all training processes for this specific architecture. Some occasional "spikes" can be observed, mostly for the generator loss, but overall, the losses show a stable behavior. The discrepancy metrics show a rather unstable behavior throughout the training process. Initially, there is almost no decrease in the metrics, but after about 500 epochs, there is a quick drop in both MMD and ED. After this drop, both metrics increases again, to later again drop at a slower rate. This behavior could be seen in many of the training processes of this specific architecture, and it was noted that such changes in the discrepancy between real and synthetic samples were difficult, if not impossible, to infer from the loss plots. Furthermore, it was noted that both the loss curves and the discrepancy curves for this architecture behaved similarly to the corresponding curves in the case of the s-BiLSTM-CNN GAN.

In Figure 4.6, the loss curves and discrepancy metrics of the t-LSTM GAN with proportional sampling are presented. Although the exact values of the binary cross-entropy loss vary between runs, it was noted that the same tendencies of initial "spikes" in generator and discriminator losses were observed each time the t-LSTM GAN architecture was trained. After this initial spike, the generator loss seems to diverge and the discriminator losses approach zero. The discrepancy metrics of the t-LSTM GAN shows similar behavior to the discrepancy metrics of the s-LSTM GAN. The metrics decrease over epochs and seem to be quite stable.

Overall, the t-LSTM GAN reached lower MMD and ED scores compared to the t-BiLSTM-CNN GAN, which can also be observed in Figures 4.5 and 4.6.

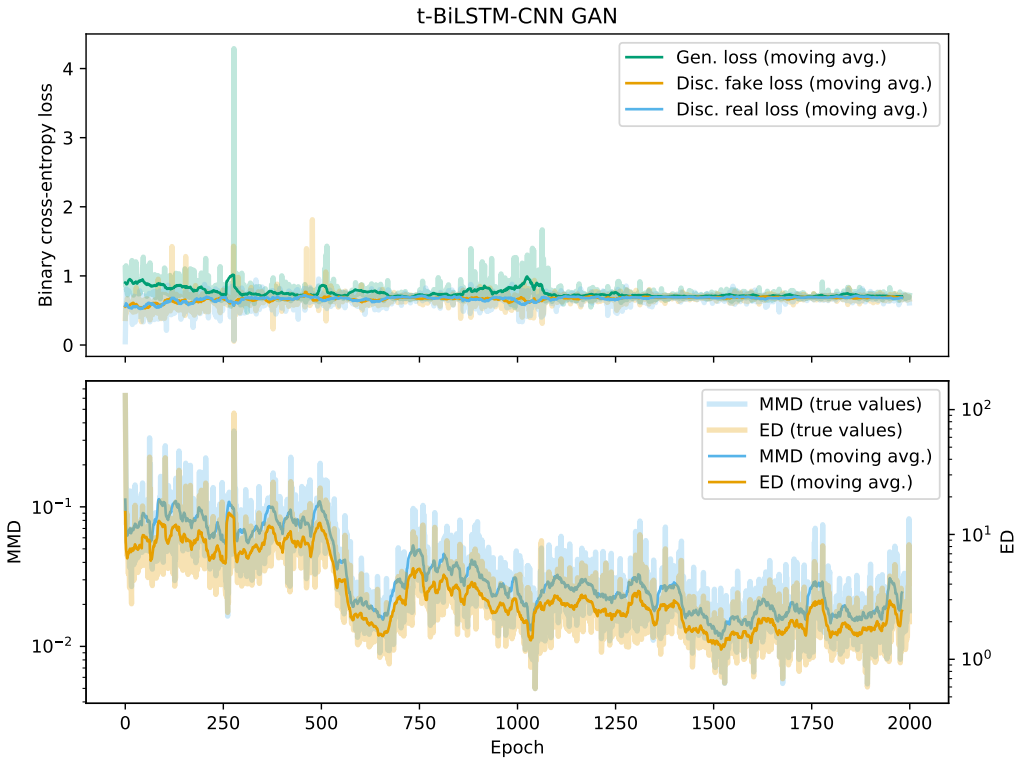


Figure 4.5: Training losses and discrepancy metrics of the t-BiLSTM-CNN GAN trained on the multivariate time-series radar dataset.

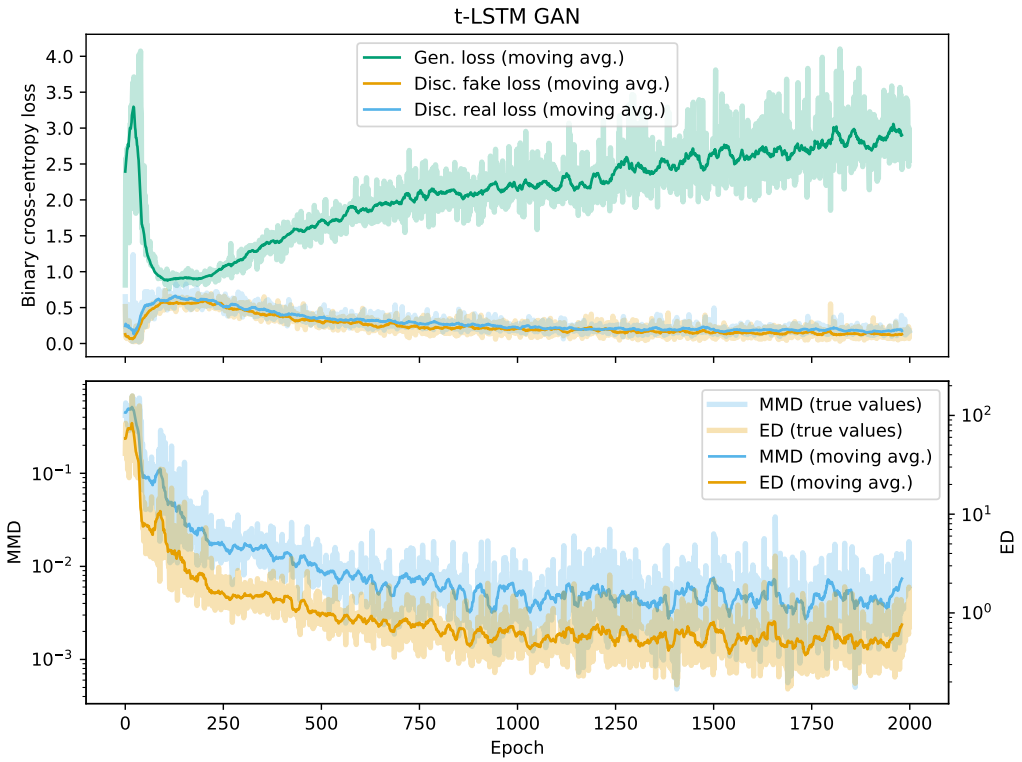


Figure 4.6: Training losses and discrepancy metrics of the t-LSTM GAN trained on the multivariate time-series radar dataset.

4.2.2 Effect of Sampling Proportion of Conditional Labels

As was described in Section 3.5.1, one aspect to consider when training a conditional GAN is how the synthetic labels, which are fed to the generator together with a latent vector, should be sampled. Three different methods were examined: to generate equal proportions of the classes, to generate the same proportion between the classes as in the dataset, and to train the same GAN architecture separately on the classes.

The result of the experiment is summarized in Table 4.1. Each sampling method (equal, proportional, only birds, and only UAVs) was tested five times for each architecture. The numbers presented in Table 4.1 are averaged over those five runs and correspond to the mean lowest MMD and ED scores obtained in each run. If the different methods affect the performance of the generator differently, one would expect at least some changes in the minimal obtained MMD and ED scores. As can be seen by considering the standard deviations, the variability between the runs was sometimes large. However, a few comments should be made. For the t-BiLSTM-CNN GAN, both the mean minimal MMD and ED are smaller for the proportional sampling compared to the equal sampling. The largest mean minimal MMD and ED obtained are from the separate sampling of birds. The only UAV case gives a mean minimal MMD comparable to the proportional sampling case, but a slightly larger value of ED.

For the t-LSTM GAN, the differences between equal and proportional sampling are larger. Furthermore, the observed MMD and ED values are much smaller compared to the corresponding values observed in the t-BiLSTM-CNN GAN case. In the case of the network trained on only UAVs, the mean minimal MMD value is negative. However, the corresponding mean minimal ED value is not smaller compared to the proportional sampling case.

The average minimal MMD and ED values seem to follow each other in almost all cases. This means that when the MMD score is large, then the ED score is in general large too. Considering that the MMD and ED curves are following each other reasonably well in for instance Figures 4.5 and 4.6, this result is consistent with the results previously presented.

Table 4.1: The average lowest maximum mean discrepancy (MMD) and energy distance (ED) scores obtained when training the t-BiLSTM-CNN GAN and the t-LSTM GAN using different sampling methods. The means and standard deviations are computed from five runs for each model.

Architecture	Sampling method	MMD	ED
BiLSTM-CNN GAN	Equal	0.00957 ± 0.0025	1.06 ± 0.22
	Proportional	0.00759 ± 0.0040	0.891 ± 0.44
	Only birds	0.0123 ± 0.011	1.61 ± 1.4
	Only UAVs	0.00718 ± 0.0031	1.00 ± 0.37
LSTM GAN	Equal	0.00150 ± 0.00050	0.282 ± 0.061
	Proportional	0.000439 ± 0.00039	0.172 ± 0.052
	Only birds	0.000827 ± 0.0012	0.316 ± 0.18
	Only UAVs	-0.000258 ± 0.00056	0.241 ± 0.036

4.2.3 The Quality of Synthetic Data from a Classification Perspective

In the following sections, the results of the three experiments performed to investigate the classification performance of the synthetic multivariate time-series data as described in Section 3.5.2 are presented. The results for the train on real, test on synthetic (TRTS) and train on synthetic, test on real (TSTR) experiments will be presented using four common classifier evaluation metrics: accuracy, precision, recall, and F1-score. For reference, the definitions of those evaluation metrics are given below together with some comments about how the metrics should be interpreted.

Firstly, define the *true negatives* (TN) as the number of birds classified as birds, the *true positives* (TP) as the number of UAVs classified as UAVs, the *false negatives* (FN) as the number of UAVs classified as birds, and the *false positives* (FP) as the number of birds classified as UAVs. The accuracy is defined as the proportion of the total number of samples classified correctly:

$$\text{accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}. \tag{4.1}$$

Precision is defined as the proportion of the total samples classified as UAVs that were actual UAVs:

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}. \tag{4.2}$$

From the definition, it is simple to see that a high precision corresponds to a low false positive rate (few birds classified as UAVs). Next, recall is defined as the proportion of the actual UAVs classified as UAVs:

$$\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}. \tag{4.3}$$

From the definition, it is clear that high recall corresponds to a low false negative rate (UAVs classified as birds). Lastly, the F1-score is a weighted average of precision and recall and defined as:

$$\text{F1} = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}. \tag{4.4}$$

F1-score thus takes into account both the false positive and the false negative rates. Note that the metrics are defined as proportions, but a common alternative presentation is to express the metrics as percentages. In this thesis, all those evaluation metrics will be expressed as percentages.

In general applications, high false positive and false negative rates are considered equally bad. However, in for instance military applications, a larger false-positive rate might not necessarily pose any problems but a larger false negative rate might. To put it simply: the possible consequences of misclassifying a bird as a UAV are less severe than the possible consequences of misclassifying a UAV as a bird.

Train on Real, test on Synthetic

To examine the quality of the synthetic samples, an experiment called train on real, test on synthetic (TRTS) was performed. The experiment is described in detail in Section 3.5.2. The idea is that if the GAN generators manage to create sufficiently realistic samples, a classifier trained on real data should be able to classify the synthetic samples approximately as good as real samples.

The average accuracy, precision, recall, and F1-score obtained for a classifier trained on real data and tested on synthetic data generated using models selected with minimal MMD can be found in Table 4.2. Note that the values are averaged over 20 runs. The last row of the table is a reference case for a classifier trained on real data and tested on real data. Note that the classifier in this reference case is trained on a smaller proportion of training data (90% of the full real dataset) while the classifiers in the other cases presented in the table are trained on the full real dataset. The reason for this difference is that a proportion of the real data needs to be saved for testing in the reference case.

Table 4.2: The average accuracy, precision, recall, and F1-score obtained when training a classifier on real data and then testing the classifier on synthetic data generated by generators of two different architectures (t-BiLSTM-CNN GAN and t-LSTM GAN) and three different sampling methods (equal, proportional, and separate). The generators were selected using the lowest obtained MMD score. As a reference, the corresponding metrics for a classifier trained on only real data are included in the bottom of the table.

TRTS, generator selected using MMD					
Architecture	Sampl. method	Accuracy (%)	Precision (%)	Recall (%)	F1 (%)
t-BiLSTM-CNN	Equal	52.3 ± 4.2	51.8 ± 3.5	61.9 ± 10	56.2 ± 5.7
	Proportional	53.1 ± 2.3	63.5 ± 10	16.1 ± 5.2	25.2 ± 6.7
	Separate	89.5 ± 2.9	91.5 ± 5.1	87.5 ± 2.1	89.4 ± 2.6
t-LSTM	Equal	$\mathbf{90.1 \pm 1.4}$	$\mathbf{94.5 \pm 1.6}$	85.1 ± 4.1	89.5 ± 1.9
	Proportional	90.0 ± 1.9	86.2 ± 3.8	$\mathbf{95.6 \pm 1.7}$	$\mathbf{90.6 \pm 1.6}$
	Separate	87.6 ± 1.1	90.6 ± 2.3	84.0 ± 2.9	87.1 ± 1.2
Reference (real)	-	90.2 ± 1.7	89.3 ± 4.1	91.7 ± 3.1	90.4 ± 1.5

As can be read out from the table, the quality of samples generated by different GAN architectures using different sampling techniques seem to vary. The best-performing models for each evaluation metric are marked with bold font. The t-LSTM GAN architecture with equal sampling reaches the highest accuracy and precision. The accuracy is on a par with the reference case, and the precision is several percentages above the reference value. This indicates that of all synthetic datasets, the classifier, in general, manages to classify the largest number of samples (independent of class) correctly from the t-LSTM GAN with equal sampling. The high precision indicates that the false positive rate is low, meaning that few birds are misclassified as UAVs. If instead recall and F1-score are considered, the best-performing model is the t-LSTM GAN with proportional sampling, reaching a recall well above the reference case and an F1-score slightly

above the reference case. This indicates that a classifier trained on real data manages to correctly classify a larger proportion of the synthetic UAVs compared to when tested on real data.

In general, the generators selected using MMD seem to generate synthetic data of high quality. Only two models seem to produce samples of quality much lower compared to real data: the t-BiLSTM-CNN GAN with equal sampling and the t-BiLSTM-CNN GAN with proportional sampling.

The average accuracy, precision, recall, and F1-score for a classifier trained on real data and tested on synthetic data from generators selected using ED can be found in Table 4.3. In this case, a single model reaches the highest values for all metrics: the t-LSTM GAN with equal sampling. Overall, nearly all values are either equally large as in the MMD case or smaller. It should be noted that for the t-BiLSTM-CNN with proportional and equal sampling, the values of all evaluation metrics are the same in the MMD and ED cases since the same generator model was selected by MMD and ED. There is one exception where the obtained metric is higher in the ED case: the accuracy of the t-LSTM GAN with equal sampling, which is slightly larger in the ED case. A notable difference is also the metrics for the t-BiLSTM-CNN GAN with separate sampling, which is much lower when the generator is selected using ED compared to the MMD case. This indicates that the models selected using ED, in general, perform worse in this classification experiment: they seem to generate synthetic data of lower quality compared to the corresponding models selected with MMD.

Table 4.3: The average accuracy, precision, recall, and F1-score obtained when training a classifier on real data and then testing the classifier on synthetic data generated by generators of two different architectures (t-BiLSTM-CNN GAN and t-LSTM GAN) and three different sampling methods (equal, proportional, and separate). The generators were selected using the lowest obtained ED score. As a reference, the corresponding metrics for a classifier trained on only real data are included in the bottom of the table.

TRTS, generator selected using ED					
Architecture	Sampl. method	Accuracy (%)	Precision (%)	Recall (%)	F1 (%)
t-BiLSTM-CNN	Equal	52.3 ± 4.2	51.8 ± 3.5	61.9 ± 10	56.2 ± 5.7
	Proportional	53.1 ± 2.3	63.5 ± 10	16.1 ± 5.2	25.2 ± 6.7
	Separate	67.5 ± 5.0	63.0 ± 4.8	87.1 ± 2.2	73.0 ± 2.9
t-LSTM	Equal	$\mathbf{90.4 \pm 1.0}$	$\mathbf{90.9 \pm 3.0}$	$\mathbf{90.1 \pm 2.6}$	$\mathbf{90.4 \pm 0.9}$
	Proportional	86.4 ± 1.5	84.2 ± 3.9	90.0 ± 3.4	86.9 ± 1.1
	Separate	83.5 ± 1.9	84.0 ± 3.2	83.1 ± 3.2	83.5 ± 1.8
Reference (real)	-	90.2 ± 1.7	89.3 ± 4.1	91.7 ± 3.1	90.4 ± 1.5

Train on Synthetic, Test on Real

To examine the ability of the synthetic data to be used as training data for machine learning methods, a classifier was trained on synthetic data and tested on real data (TSTR) as described in Section 3.5.2. The results for the classifier trained on synthetic data from generator models

selected using the MMD criterion can be found in Table 4.4. Also in this case, a reference result for the classification performance when trained and tested on real data is included.

Table 4.4: The average accuracy, precision, recall, and F1-score obtained when training a classifier on synthetic data generated by generators of two different architectures (t-BiLSTM-CNN GAN and t-LSTM GAN), and three different sampling methods (equal, proportional, and separate) and then testing the classifier on *only* real data. The generators were selected using the lowest obtained MMD score. As a reference, the corresponding metrics for a classifier trained on only real data are included in the bottom of the table.

TSTR, generator selected using MMD					
Architecture	Sampl. method	Accuracy (%)	Precision (%)	Recall (%)	F1 (%)
t-BiLSTM-CNN	Equal	57.2 ± 2.1	68.7 ± 4.0	26.6 ± 6.4	37.9 ± 6.8
	Proportional	74.5 ± 2.7	77.3 ± 2.1	69.6 ± 8.3	72.9 ± 4.5
	Separate	77.0 ± 1.2	70.0 ± 1.4	$\mathbf{94.7 \pm 1.1}$	80.5 ± 0.7
t-LSTM	Equal	76.1 ± 3.3	72.1 ± 5.2	86.7 ± 4.4	78.5 ± 1.9
	Proportional	$\mathbf{85.5 \pm 0.9}$	86.7 ± 2.6	84.1 ± 3.8	$\mathbf{85.3 \pm 1.2}$
	Separate	83.7 ± 2.6	$\mathbf{89.9 \pm 2.3}$	76.1 ± 7.3	82.2 ± 3.9
Reference (real)	-	90.2 ± 1.7	89.3 ± 4.1	91.7 ± 3.1	90.4 ± 1.5

As can be seen in the table, the accuracy of a classifier trained on synthetic data from the different GAN generators is smaller in all cases compared to the reference case. When a classifier is trained on data from the best succeeding model, the t-LSTM GAN with proportional sampling, an accuracy of 85.5% is reached on average, which is lower compared to the reference case of 90.2%. The worst performing model considering accuracy is the t-BiLSTM-CNN GAN model with equal sampling, where a classifier trained on this data reaches an accuracy of 57.2%. However, the accuracy metric only reflects the total classification performance and not how well the classifier manages to classify samples from the different classes. If the recall column is considered, there is one outstanding result: the recall of the t-BiLSTM-CNN GAN model with separate sampling, which is 94.7%. This score is several percentages higher than the reference case recall of 91.7%. As was described in Section 4.2.3, the recall is the proportion of UAVs correctly classified. By training the classifier on purely synthetic data, the classifier classifies a larger proportion of the UAVs correctly compared to when trained on real data. However, the precision of this model is lower, which means that the number of false positives (birds classified as UAVs) is larger compared to the reference case. The worst-performing model, if the recall is considered, is the t-BiLSTM-CNN GAN model with equal sampling, where a classifier trained on synthetic data from this model reaches a recall of 26.6%. The model performing best if both precision and recall are taken into account is the t-LSTM GAN with proportional sampling, reaching an F1-score of 85.3%. This is a few percentages lower than the reference case. The model reaching the highest precision score is the t-LSTM GAN with separate sampling, meaning that a classifier trained on this model is least prone to misclassify birds as UAVs.

The results for the classifier trained on synthetic data from generator models selected using the ED criterion can be found in Table 4.5. The best-performing models for each of the evaluation metrics are slightly different from the result presented for the MMD case in Table 4.4 and the

top values of the metrics are overall lower. The model reaching the highest accuracy (84.0%) is the t-LSTM GAN with equal sampling. This model is also the best-performing model if F1-score is considered. The t-BiLSMT-CNN GAN model with separate sampling once again manages to score the highest recall value of 89.2%, which is lower than both the reference score and the score obtained for the same model in the MMD case. The t-LSTM GAN with proportional sampling reaches the highest precision score.

Table 4.5: The average accuracy, precision, recall, and F1-score obtained when training a classifier on synthetic data generated by generators of two different architectures (t-BiLSTM-CNN GAN and t-LSTM GAN) and three different sampling methods (equal, proportional, and separate) and then testing the classifier on *only* real data. The generators were selected using the lowest obtained ED score. As a reference, the corresponding metrics for a classifier trained on only real data are included in the bottom of the table.

TSTR, generator selected using ED					
Architecture	Sampl. method	Accuracy (%)	Precision (%)	Recall (%)	F1 (%)
t-BiLSTM-CNN	Equal	57.2 ± 2.1	68.7 ± 4.0	26.6 ± 6.4	37.9 ± 6.8
	Proportional	74.5 ± 2.7	77.3 ± 2.1	69.6 ± 8.3	72.9 ± 4.5
	Separate	64.9 ± 4.6	60.3 ± 4.1	$\mathbf{89.2 \pm 3.2}$	71.8 ± 2.5
t-LSTM	Equal	$\mathbf{84.0 \pm 1.5}$	81.4 ± 3.7	88.6 ± 3.0	$\mathbf{84.7 \pm 0.9}$
	Proportional	80.3 ± 2.0	$\mathbf{85.6 \pm 2.6}$	73.3 ± 6.8	78.7 ± 3.3
	Separate	80.7 ± 1.7	85.2 ± 2.2	74.5 ± 4.9	79.4 ± 2.4
Reference (real)	-	90.2 ± 1.7	89.3 ± 4.1	91.7 ± 3.1	90.4 ± 1.5

Overall, synthetic data from the models selected using MMD seem to perform better as training data for a classifier compared to synthetic data generated by the models selected using ED.

Train on a Mixture, Test on a Mixture

As described in Section 3.5.2, it is interesting to investigate the classification performance of a model trained on a mixture of real and synthetic data. By also testing the classifier on a mixture of real and synthetic data, any difference in classification performance between real and synthetic test samples can be examined.

In Figure 4.7, the confusion matrix obtained when training the classifier on real data and tested on real data can be found. This confusion matrix can be used as a reference when studying the confusion matrices of the train on a mixture, test on a mixture (TMTM) experiment presented later in this section. Note that the reference confusion matrix is from the same run as the reference evaluation metrics presented in Tables 4.2, 4.3, 4.4, and 4.5, meaning that the confusion matrix in Figure 4.7 is merely an alternative way of presenting the same data.

The confusion matrices of the generator models selected using MMD can be found in Figure 4.8. As can be seen, each confusion matrix represents one specific architecture and label sampling method. The classification counts of real and synthetic samples have been separated in the matrices to clarify any difference in classification performance between real and synthetic

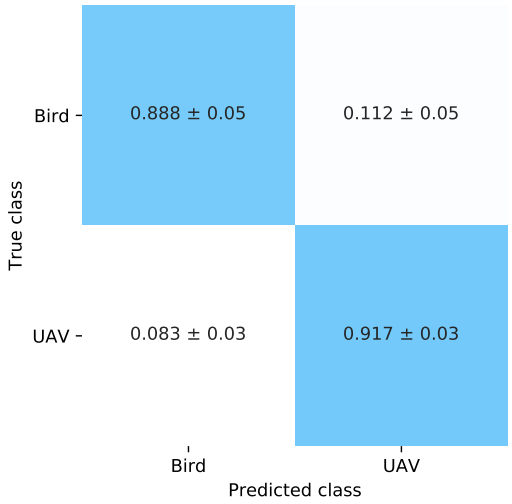


Figure 4.7: The confusion matrix obtained when testing a classifier trained on only real data on a withheld test set, summarized over 20 iterations of MCCV. Each test set consists of 10% of the total samples in the real dataset.

samples. Note however, that the real and synthetic samples were not treated as separate classes while training the classifier. Also in the matrices showed in Figure 4.8, the values are computed from 20 MCCV iterations.

In all confusion matrices, it is clear that the classifier has managed to classify the synthetic samples better compared to the real samples. For the t-BiLSTM-CNN GAN with equal and separate sampling, the proportion of correctly classified synthetic samples of both classes are close to 1.0, meaning that no synthetic samples are misclassified. For the t-LSTM GAN, the proportion of correctly classified synthetic samples are lower compared to the t-BiLSTM-CNN GAN, but still higher compared to the reference case in Figure 4.7. Ideally, if the GANs have managed to capture the full distribution of the training data, the proportions of correctly classified samples should be roughly equal for real and synthetic data.

Another aspect to consider is if the classifier seems to have been aided by the synthetic data added to the real training data. One way to evaluate this is to consider the classification performance on the real samples in the test set. Although this evaluation method has some similarities with the TSTR experiment, it is important to remember one large difference: in this experiment, the classifier has been trained on a mixture of real and synthetic data, and therefore the results of this experiment indicate the ability of the synthetic data to add meaningful information to the real data. This interaction of synthetic and real data during training is not captured by the TSTR experiment.

For several of the models, including t-BiLSTM-CNN GAN with equal and separate sampling, and t-LSTM GAN with equal, proportional, and separate sampling, the proportion of correctly classified birds is larger compared to the reference case. This indicates that the synthetic data aids the classifier to classify birds correctly. However, for all but the t-LSTM GAN with equal sampling, the proportion of correctly classified UAVs are smaller compared to the reference case.

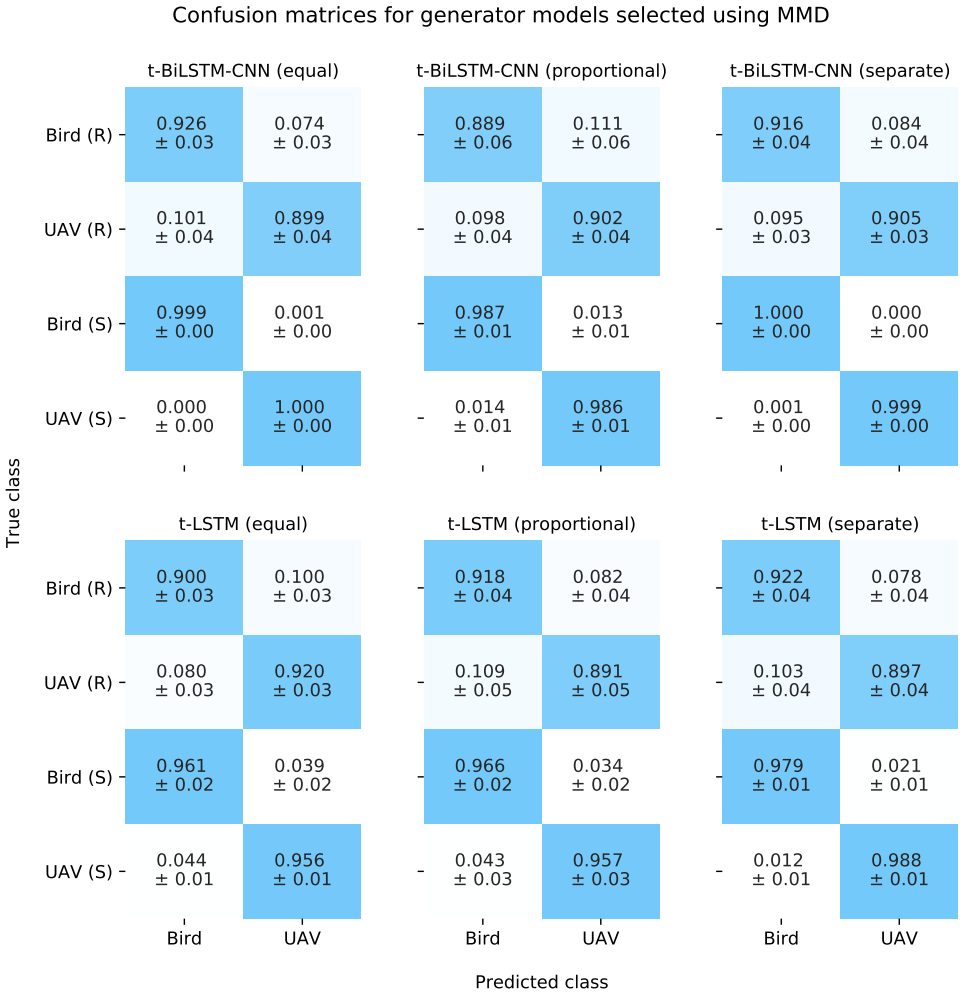


Figure 4.8: Confusion matrices for the six synthetic datasets (three each for models t-BiLSTM-CNN GAN and t-LSTM GAN with proportional, equal, and separate label sampling) where the optimal generator models have been selected using the lowest obtained MMD value. Each confusion matrix has four rows and two columns, where each row corresponds to a real data class and each column to the predicted class by the classifier. Note that the synthetic and real samples are separated in the matrix, but were not treated as different classes during the training of the classifier.

Hence, the t-LSTM GAN with equal sampling seems to be the only model that improves the overall accuracy of the classifier, under the assumptions made in this experiment.

In Figure 4.9, the classification results from a classifier trained on a mixture of real and synthetic data from generators selected using ED are presented. Also in this case, the classifier is better at classifying synthetic samples from all generator models. Furthermore, the usage of synthetic data from all models but one (t-BiLSTM-CNN GAN with proportional sampling) improves the proportion of correctly classified birds compared to the reference case. However, none of the models generate data that improves the proportion of correctly classified UAVs. The best generator model in this aspect is the t-BiLSTM-CNN GAN with separate sampling, which manages to classify 91.3% of the UAVs correctly.

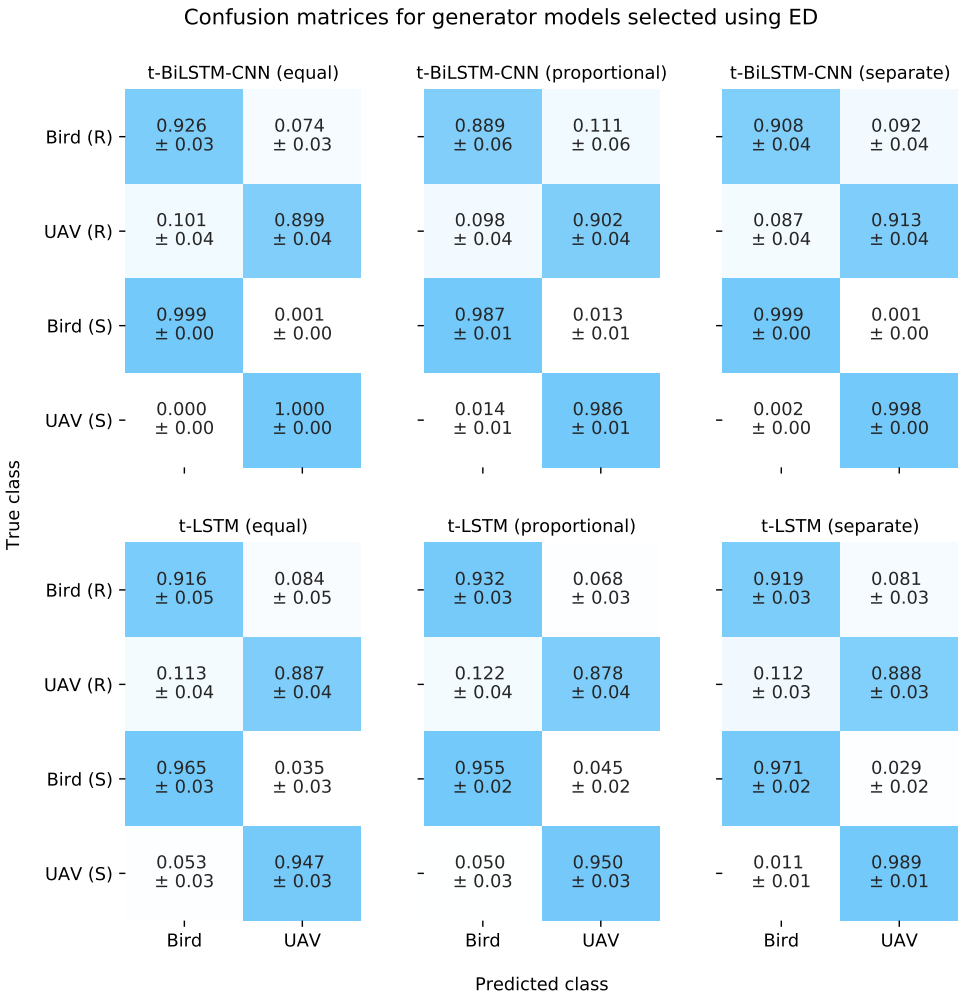


Figure 4.9: Confusion matrices for the six synthetic datasets (three each for models t-BiLSTM-CNN GAN and t-LSTM GAN with proportional, equal, and separate label sampling) where the optimal generator models have been selected using the lowest obtained ED value. Each confusion matrix has four rows and two columns, where each row corresponds to a real data class and each column to the predicted class by the classifier. Note that the synthetic and real samples are separated in the matrix, but were not treated as different classes during the training of the classifier.

4.2.4 The Proportion of Synthetic Data Versus Classification Accuracy

As was described in Section 3.5.3, the classification accuracy as a function of the proportion of added synthetic data to the training dataset was measured. The results were averaged over 10 runs. The results for the cases where synthetic data from generators selected using MMD was used can be found in Figure 4.10. Here, results for the t-BiLSTM-CNN GAN and the t-LSTM GAN are presented separately. The shaded regions in the figure represent the standard deviation of the average values over those 10 runs. The solid lines are the average accuracies computed over 10 runs. The dashed line is a linear ordinary least squares (OLS) estimate to capture the change of accuracy as the number of synthetic samples is increased. Note that the figure has a logarithmic x-axis, and therefore, the lines are not linear in the figure.

From Figure 4.10, it can be observed that the classifier trained on a mixture of real and synthetic data from the t-BiLSTM-CNN generator selected using MMD seem to be rather unaffected by the addition of larger proportions of synthetic data. This result holds for all sampling methods. A small decline in accuracy for the separate and proportional sampling methods can be observed. For the t-LSTM GAN, the accuracy of a classifier trained on real and synthetic data from a generator trained with either proportional and equal sampling is unchanged even for larger proportions of synthetic data. For the separate sampling case, however, there is a clear decline in accuracy as larger proportions of synthetic data are added.

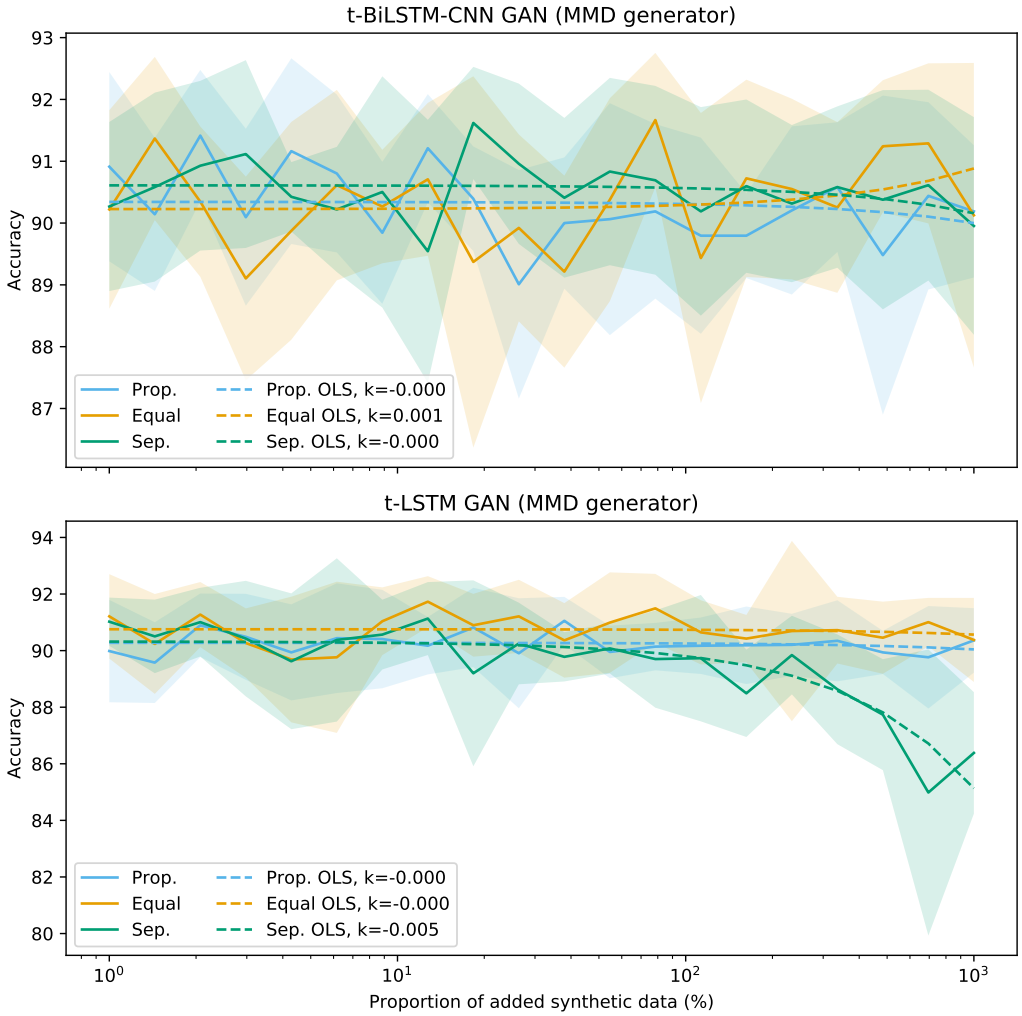


Figure 4.10: The accuracy of the GRU classifier, tested on real data only, as a function of the proportion of added synthetic data from generators selected using MMD. The accuracy is computed by averaging over 10 runs, where the test and training sets are randomly selected in each run. The shaded regions in the figure correspond to the standard deviations of the accuracies over the 10 runs and the dashed line is an ordinary least squares (OLS) fit of the mean accuracies to illustrate the development of the accuracy over the different proportions.

In Figure 4.11, the corresponding plot for a classifier trained on synthetic data from generators selected using ED is shown. Overall, the results are consistent with the MMD case. For the t-BiLSTM-CNN architecture, a decline in accuracy can be seen for the separate sampling case.

For the t-LSTM architecture, a decline for all three sampling methods can be observed. In this case, the separate sampling method seems to yield the largest decline in accuracy for the t-LSTM architecture in the MMD case.

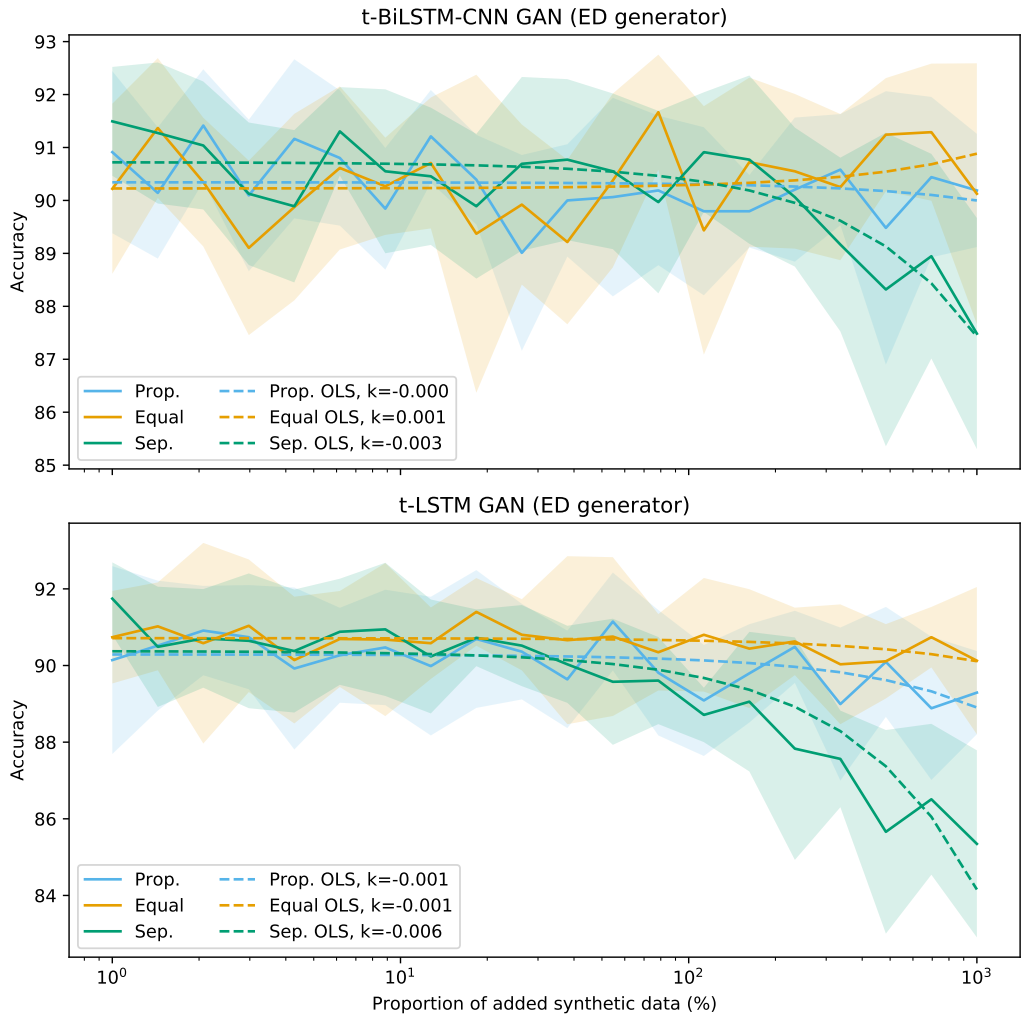


Figure 4.11: The accuracy of the GRU classifier, tested on real data only, as a function of the proportion of added synthetic data from generators selected using ED. The accuracy is computed by averaging over 10 runs, where the test and training sets are randomly selected in each run. The shaded regions in the figure correspond to the standard deviations of the accuracies over the 10 runs and the dashed line is an ordinary least squares (OLS) fit of the mean accuracies to illustrate the development of the accuracy over the different proportions.

5 Discussion

In the following chapter, the results presented in Chapter 4 will be discussed. Furthermore, the data preprocessing, the dataset itself, and the applicability of the methods will be discussed.

5.1 The Multivariate Time-Series Radar Dataset

Before discussing the results of the experiments, some comments regarding the quality and preprocessing of the multivariate time-series dataset used as training data for the conditional recurrent GAN models are needed.

The dataset used was prepared and labeled as preparatory work in a previous thesis [3]. The labeling was done manually, and the dataset may therefore contain mislabeled samples. During data collection, the UAV was either tracked with GPS or flown in certain patterns to make sure the UAV tracks would be distinguishable from bird tracks. However, an assumption made in the process of labeling the data is that all samples that are known to *not* be UAV tracks are labeled as birds. The possibility that UAVs other than the one flown by Saab were recorded during data collection cannot be completely ruled out. Therefore, some bird samples may be UAVs. However, the authors of the previous thesis estimated the risk of incorrectly labeled data as relatively small.

Another quality aspect of the recorded data is diversity. As was described in Section 3.2.2, the UAVs were mostly flown in similar patterns to tune the signal processing system of the radar. Therefore, the recorded UAV samples are not as diverse as they would have been if the UAV had been flown in more varied patterns. The fact that the UAV samples were split into relatively short sequences (20 time steps) should to some extent prevent the networks to learn the flying patterns of the UAV. However, the flying patterns of the UAV present in the data recordings might not be representative of flying patterns of UAVs in general.

Another quality aspect worth mentioning is that some physical features, which are likely to differ between birds and UAVs, are missing from the data. One such example is doppler spectral widening, which was not used as a feature and potentially could give important information about how different parts of the target are moving. It is reasonable to believe that if this feature

would be present in the dataset, it could be important both for learning a GAN to produce realistic samples and for improving the performance of a classifier.

The preprocessing of the dataset was performed using the same processing steps as in a previous master thesis [3]. One step that might require a comment is the fact that the UAV samples are split into subsequences of 20 time steps and treated as independent samples. In reality, subsequences from the same time-series are not independent. This technique of augmenting time-series data has been used in other studies [41, 42], and has been shown to work well in practice. To the author, no study is known where this method has been evaluated in depth. However, since the raw dataset was heavily unbalanced and the method has been used successfully in previous studies, it was decided to use it although the assumption of independence does not hold in practice.

5.2 The Distribution Discrepancy Metrics

In the following sections, different aspects of the usage of the two discrepancy metrics maximum mean discrepancy (MMD) and energy distance (ED) are discussed.

5.2.1 Applicability of Metrics

In this thesis, both MMD and ED have been used to evaluate the training progress and for model selection. Since they are distance-based, both metrics are simple and fast to compute and there are no practical issues with implementing the metrics for neither univariate time-series or multivariate time-series data. In several previous studies, MMD has been used for either monitoring the training progress or selecting the generator models to generate synthetic data from [7, 9, 10]. No previous work is known to the author where ED has been used as either a tool for training monitoring or generator selection.

In the case of sinusoidal waves, MMD and ED perform very similarly. Both scores decrease with increasing epochs and seem to reflect the improvement of the generated samples well. However, as was mentioned in Section 4.1.1, values of both MMD and ED are sometimes smaller for sequences not resembling sinusoidal waves, although the metrics are computed by comparing the generated samples with actual sinusoidal waves. This suggests that single MMD and ED values sometimes can be misleading in relation to sample quality. However, since the metrics seem to correspond well with the visual improvement of the sinusoidal waves, the metrics were used to evaluate the training process also for the multivariate time-series GANs.

In the multivariate time-series case, the applicability of the metrics is harder to evaluate since no visual inspection of the generated samples can be performed. However, both MMD and ED showed similar behavior when monitored during epochs compared to when used on the sinusoidal data. This can be seen by comparing Figures 4.5 and 4.6 with Figures 4.3 and 4.4. As can be read out from Table 4.1, lower MMD and ED scores were obtained for the t-LSTM GANs if compared to the corresponding t-BiLSTM-CNN GANs. Does this mean that lower

MMD and ED scores correspond to more realistic samples? Naturally, this is desirable since the usage of MMD and ED aims at evaluating the sample quality during training. To answer this question, Tables 4.2 and 4.3 should be considered. Here, the best-performing model is the t-LSTM architecture with equal sampling if all metrics are considered in the ED case, and if accuracy and precision are considered in the MMD case. The t-LSTM GAN with proportional sampling is the best performing model if recall and F1-score are considered in the MMD case. Since those GAN models also yield the lowest MMD and ED scores during training (as can be seen in Table 4.1), the results indicate that lower obtained MMD and ED values correspond to samples of higher quality for the multivariate time-series data too. It should be noted that it is difficult to draw any conclusions in the separate sampling case. The minimal MMD and ED values obtained during training (see Table 4.1) for the only bird and only UAV cases are in most cases much lower compared to the equal and proportional sampling. However, this is expected since the discrepancy within-class is should be smaller compared to the discrepancy between-class. Therefore, such low values are not necessarily lower due to better sample quality, but could also be lower since only one class is present in the samples.

5.2.2 The Choice of Evaluation Metrics for GAN Samples

Both maximum mean discrepancy and energy distance are distance-based kernel metrics. The intention was to include several different metrics from both the statistical and information theory domain. However, many evaluation metrics were ruled out for different reasons. A problem with some of the statistical metrics considered is that they are not suitable for multivariate data and/or not easily interpreted for time-series data. Examples of metrics ruled-out for those reasons are the Anderson-Darling and Kolmogorov-Smirnoff tests. These tests are popular statistical tests for testing if a sample is drawn from a given distribution (Anderson-Darling) or if two samples are drawn from the same distribution (Kolmogorov-Smirnoff), but does not generalize to multivariate time-series data. From the information theory domain, a metric called Jensen-Shannon divergence (JSd) was considered. Since JSd measures the discrepancy between distributions, it would be natural to use this metric to measure the discrepancy between the real and synthetic data distribution during the training of a GAN. Furthermore, this metric appears in previous work as an evaluation metric for generative models [25]. However, a key point here is that JSd measures the discrepancy between *distributions*. If the data distribution is known, this metric is simple to compute. In this project, the distributions of the real or synthetic data are not known - only samples from them are available. Computing JSd, or rather computing an estimate of JSd, under so-called *weak agnostic assumptions* with only finite i.i.d. samples available has been the subject of previous work [43]. A common method to estimate JSd is to first estimate the densities p and q, and then use the estimates to compute the divergences [44]. Several studies suggest that an initial estimate of the distributions is not necessary to estimate JSd [44, 45], but it is unclear if such an estimate would be a suitable evaluation metric in this application. Although investigating the usage of such methods would be interesting, it was considered to be beyond the scope of this thesis.

The metrics left at this point were MMD and ED. Both metrics were found to be stable, fast to compute, and intuitively simple to understand. Therefore, they were selected as discrepancy metrics in this thesis. Since no previous works have been found where samples from GANs have been evaluated using ED, it was interesting to investigate if this metric had any advantages to the more well-used MMD metric.

5.2.3 Evaluation Metrics Versus Loss for Training Monitoring

One issue with training GANs is that no universal stopping criterion exists. The most common way to determine if the adversarial training should be stopped is to visually inspect the generated samples and determine if they look sufficiently realistic. This method is feasible for image data, but not for numerical data of the type that has been handled in this thesis. Furthermore, it is far from certain that changes in the distribution of synthetic data are directly visible, even for image data. Using visual inspection for determining when to stop training a GAN is therefore not a very precise method.

When training neural networks in general, it is helpful to inspect the training and validation losses to determine when the training should be stopped. At a certain point, the validation loss ceases to decrease, and this is a sign that all training after this point overfits the network on the training data. However, as could be seen in for instance Figures 4.5 and 4.6, the behavior of the losses of the generator and discriminator does not necessarily reflect the development of the sample quality. This is, in particular, true for the case in Figure 4.5, where the losses show a stable behavior, but yet the sample quality seems to fluctuate. Therefore, an important contribution of this thesis is to show that two such simple metrics as MMD and ED could be used to monitor how the sample quality changes over time. This conclusion is based on the discussion in Section 5.2.1: the GAN architecture that in the TRTS experiment seemed to have generated the most realistic samples was also the model whose samples reached the minimal MMD and ED values during the training process. Compared to visual inspection or monitoring of the loss function, both MMD and ED seem to be better indicators of the training progress.

5.2.4 Maximum Mean Discrepancy or Energy Distance?

In this work, the usage of MMD and ED has two purposes: firstly, to monitor the development of the sample quality during training and secondly, to select a suitable generator model to generate synthetic data from.

For monitoring the training progress, the two metrics have performed similarly in almost all runs. For the monitoring purpose, it is enough to have metrics that roughly capture the improvements of the synthetic samples, and both metrics seem to have managed this. Compared to the information that can be obtained from monitoring the loss and visual inspection, this is a clear improvement. In some runs, slightly negative MMD values have been observed. As was explained in Section 4.1.2, although MMD is a squared metric and hence always non-negative,

the unbiased estimate of MMD is not guaranteed to be non-negative. Such negative values show up as sudden dips in the MMD curve. This can, for instance, be seen in Figure 4.4. Such behavior cannot be seen for ED since the energy distance estimate used is guaranteed to be non-negative. However, the small negative values in some of the MMD curves do not affect the applicability of the metric for monitoring purposes.

For selecting a suitable generator model, the experiments show differences in performance between models selected using MMD and ED. It should be noted that for two GANs, namely the t-BiLSTM-CNN GAN with equal and proportional sampling, the metrics select the same generator model. For the other GANs, the selected models differ. In the train on real, test on synthetic (TRTS) experiment (see Tables 4.2 and 4.3 for results), classifiers tested on synthetic data from generators selected using MMD achieves higher precision, recall, and F1-score compared to classifiers tested on synthetic samples from generators selected using ED. This can be seen by considering the values marked in bold in Tables 4.2 and 4.3, which are the highest obtained metric scores. The highest obtained accuracy was higher in the ED case, but the value is only slightly larger than in the MMD case. For the train on synthetic, test on real (TSTR) experiment (see Tables 4.4 and 4.5), similar tendencies can be seen. A classifier trained on synthetic data from a generator selected using MMD reaches higher values of all metrics compared to the ED case. Furthermore, in Table 4.4, it can be seen that the recall of a classifier trained on synthetic data from one of the generators (t-BiLSTM-CNN GAN with separate sampling) selected using MMD was several percentages higher compared to the reference case (94.7% compared to 91.7%). For the ED case, the recall of the same GAN model is lower than the reference case (89.2% compared to 91.7%). This particular result is important since the recall is of large interest in this specific military application: misclassification of UAVs can lead to more severe consequences compared to misclassification of birds. In the train on a mixture, test on a mixture (TMTM) experiment, it was noted that in the ED case, none of the architectures seem to be able to generate data that improve the proportion of correctly classified birds *and* UAVs. In the MMD case, one architecture was observed to be able to do this (the t-LSTM GAN architecture with equal sampling).

Overall, the results indicate that MMD is a better choice for selecting a suitable generator model to use for generating synthetic samples. To monitor the training progress of the GANs, no major difference in performance between the two metrics was observed.

5.3 The Classification Performance of Synthetic Data

In the following sections, the different experiments performed on the synthetic multivariate time-series data and the results of those experiments will be discussed. Instead of discussing the experiments separately, the section has been divided into answering the following four questions:

1. Could different sampling methods of the synthetic labels during the training of a GAN affect the classification performance of the synthetic data?

2. Do the GANs manage to generate realistic data, i.e., samples of high quality?
3. Do the GANs manage to capture the full distribution of the training dataset, i.e., is there any detectable covariate shift in the synthetic data?
4. Does using synthetic data from GANs to augment existing real-world datasets seems like a useful method?

Those questions will be answered one-by-one in the following sections.

5.3.1 The Methods for Sampling Conditional Labels During Training

In general, no clear pattern could be seen between the performance of the datasets generated from GANs trained using the different sampling techniques. However, for the t-BiLSTM-CNN GAN, the separate sampling of the synthetic labels seems to generate data of higher quality compared to the t-BiLSTM-CNN GAN trained using the other sampling techniques. This can be read out from the TRTS Tables 4.2 and 4.3, and the TSTR Tables 4.4 and 4.5. In the TRTS case, in essence, all metric scores are higher for the separate sampling case compared to equal and proportional sampling. There is one exception: the precision in the case of TRTS with generator models selected using ED. However, the standard deviations of the precision for the t-BiLSTM-CNN GAN are large, so it is difficult to draw any conclusions from this slightly smaller mean value. In the TSTR case, separate sampling with the t-BiLSTM-CNN architecture yields higher accuracy, recall, and F1-score in the MMD case and higher recall in the ED case compared to the other sampling methods.

Altogether, the results indicate that separate sampling seems to generate more realistic samples in the t-BiLSTM-CNN GAN case. In a way, this is not surprising. With separate sampling, the same GAN architecture is trained on the data classes separately. Hence, a dataset generated from two such GANs is generated using twice as many trainable parameters compared to the other sampling methods. This should lead to models with smaller bias, but also most likely models with larger variance due to larger model complexity. Since an increase in sample quality can be seen by training the t-BiLSTM-CNN GAN separately on the two classes, the architecture itself may contain too few trainable parameters. This would also explain why the t-BiLSTM-CNN GAN architecture with other sampling techniques seems to produce less realistic samples. However, if an architecture already contains a sufficient number of trainable parameters, training on each of the classes separately might lead to overfitting. The results from the experiment where the accuracy of a classifier trained on different proportions of synthetic data (Figures 4.10 and 4.11) indicate that this might be the case for the t-LSTM GAN architecture. The accuracy of the t-LSTM GAN with separate sampling declines much faster as larger proportions of synthetic data is added, which could indicate synthetic data of lower quality, perhaps due to the GAN overfitting the training data.

5.3.2 The Sample Quality of the Synthetic Data

As was briefly explained in Section 3.5.2, the train on real, test on synthetic (TRTS) experiment can measure if the synthetic samples are realistic. The idea is that if a synthetic sample is sufficiently realistic, a classifier trained on real data should be able to recognize and classify synthetic samples roughly as well as real samples. Since the classifier is trained on the full real dataset (where all data modes are present), the result of the experiment is not affected by a possible covariate shift in the synthetic dataset.

As can be seen in Tables 4.2 and 4.3, several of the trained GAN generators seem to be able to create highly realistic samples. Samples from the t-BiLSTM-CNN GAN with separate sampling (in the MMD case) and the t-LSTM GAN with equal, proportional, and separate sampling (in both MMD and ED cases), seem to be classified approximately as good as real samples by a classifier trained on real data only. Lower metric scores indicate that the samples are not similar to the real samples: the features learned from the real data cannot be found in the synthetic samples by the classifier. Examples of cases where this is happening are for the t-BiLSTM-CNN GAN with equal and proportional sampling for both the MMD and ED cases and the t-BiLSTM-CNN GAN with separate sampling in the ED case. Higher metric scores compared to the real case would indicate that the GAN has succeeded in identifying important features of the real data, but have failed to capture the full complexity of the real data. A classifier trained on real data could then easily classify synthetic samples correctly. However, such a result has not been observed in this experiment.

Another result worth mentioning is the differences in classification accuracy between real and synthetic samples in the train on a mixture, test on a mixture (TMTM) experiment. When a classifier is trained on an equal proportion of real and synthetic data, it classifies a larger proportion of the synthetic samples correctly compared to the real samples. This seems to be particularly true for the cases where a classifier is trained on a mixture of real data and less realistic synthetic data. Why is the classifier better at predicting the correct class of synthetic samples of lower quality?

A possible explanation is that a synthetic dataset with samples of lower quality has a less complex decision boundary. A decision boundary is a hypersurface separating the two data classes in some high-dimensional data space. The classifier classifies all points on one side of the boundary to one class and all points on the other side of the boundary to the other class. The training of a classifier aims at localizing this decision boundary. Since the classifiers in the TMTM experiment seem to be able to more easily separate the synthetic samples compared to the real samples, this indicates that the decision boundaries of the synthetic datasets are less complex. A previous study also mentions diversity loss in the boundary regions, i.e. boundary distortion, as a possible explanation [4]. Boundary distortion will be further discussed in the next section.

5.3.3 The Diversity of the Synthetic Samples

The second quality aspect of the synthetic data regards the diversity of the synthetic dataset. It is not enough for the GAN to generate realistic samples: the real data distribution must be well-represented. As was presented in Section 1.3, one usually refers to the difference in distribution between the synthetic and real samples as a covariate shift. Two main types of covariate shifts are commonly discussed: mode collapse and boundary distortion.

Mode collapse refers to the case when the generator is more likely to produce some modes of the training data. In the extreme case, the generator is only able to generate a single mode. In this study, conditional GANs have been used for the multivariate time-series data. This means, that since both the generator and discriminator are conditioned on the class labels, the conditioned information determines which modes to be generated. Therefore, mode collapse in its commonly described form does not occur while training conditional GANs. The second type of covariate shift, boundary distortion, describes the loss of diversity within each class and more specifically, a failure of capturing the training data distribution at boundary regions of the support. This type of covariate shift is relevant for this project. Although it is possible to steer the proportion of *samples* in each class, it is not possible to steer the *diversity* within each class.

The main finding indicating boundary distortion in the synthetic data is the fact that although at least some of the GAN generators seem to be able to produce highly realistic synthetic samples (for instance t-LSTM with equal, proportional, and separate sampling in Tables 4.2 and 4.3), this data seems to not be able to improve the classification accuracy of a classifier trained on only or partially synthetic data. This can be observed in the TSTR experiments (Tables 4.4 and 4.5), where a classifier trained on the various synthetic datasets at all times performs worse compared to when trained on real data. Since it has been shown that at least the t-LSTM architecture generates highly realistic samples independent of the sampling method, the lower numbers cannot be explained with low sample quality. Therefore, those results indicate that the data suffer from a loss of diversity.

In the experiment investigating the classification accuracy as a function of added synthetic proportion, the accuracy seems to be either unaffected or decline as larger proportions of synthetic data from the generators are added. The results can be found in Figures 4.10 and 4.11. Considering that the classifier seemed to perform worse when trained on synthetic data, it is not completely surprising that adding more synthetic data leads to a decline in classification accuracy. However, one result is surprising: although the t-BiLSTM-CNN GAN has shown to produce less realistic samples, especially in the equal and proportional sampling cases, the accuracy of the classifier barely decreases at all when larger proportions of synthetic data from those generators are added. This holds for all t-BiLSTM-CNN generators selected using MMD and the equal and proportional sampling t-BiLSTM-CNN generators selected using ED. This

could indicate that the diversity of the samples from the t-BiLSTM-CNN GAN is larger and that this compensates for less realistic samples.

A possible explanation for a larger diversity in the t-BiLSTM-CNN data is that this architecture includes a minibatch discrimination layer, forcing the generator to produce more diverse samples. Such a layer was included in the BiLSTM-CNN architectures since it is commonly used together with convolutional layers in discriminators.

5.3.4 The Applicability of GANs as an Augmentation Method

As was discussed in the previous sections, at least some of the GANs seem to be able to generate highly realistic samples. However, the performance gain of providing a classifier additional synthetic samples seem to have been modest in most cases. It is reasonable to use synthetic samples from GANs for augmentation purposes?

First of all, how well have others succeeded in using synthetic samples from GANs to augment real-world datasets? Several of the experiments performed in this thesis have to some extent been performed in previous work. The train on synthetic, test on real (TSTR) experiment was first introduced in a study aiming at generating multivariate medical time-series using GANs [5]. The authors of this study concluded that they managed to generate data at times comparable to real data, but the classifier used did not perform as well on the synthetic data for any of the features considered. Regardless of this, the authors believe that with small refinements of their methods, such synthetic data could be of use in real applications.

An experiment similar to the accuracy versus synthetic proportion experiment was performed in a previous work [46]. There, the authors measured the accuracy as a function of the proportion of synthetic data added to the training dataset from different GAN models. However, they did not present the average accuracy over a fixed number of runs, but over the top five runs with the highest accuracy. Only two of the six models managed to improve the classification accuracy when up to 50% synthetic data were added, and for larger proportions, the top five accuracies declined also for those models. The training data, in this case, were images from the database ImageNet, and the GAN architectures used were, with that said, different from the ones used in this study. A study that compared traditional augmentation methods with GANs was published in 2017 [47]. Traditional methods include cropping, rotating, and flipping of the training images. The presented results showed that the traditional augmentation performed better compared to using synthetic GAN data when tested using a CNN classifier.

There also exist studies where GANs have been used successfully as an augmentation method. One such study was published in 2019 [48]. In this work, the authors trained GANs on different multivariate datasets. None of the datasets contained any time-series data. For one of the datasets, the Breast Cancer Wisconsin (Diagnostic) Dataset, the accuracy of a classifier improved when trained on purely synthetic data and then tested on a real samples, compared to when trained and tested on real data. The same tendencies could not be seen for the other dataset

tested in the study: for them, the obtained test accuracy was about as high as for the real case.

Making comparisons like this is difficult since different studies handle different datasets and use different architectures. In all presented studies, relatively simple GANs have been used, and none of the architectures are specially tailored for handling classification problems. Overall, it seems like using such simple GANs as an augmentation method has been tested to some extent, mostly for image data, and that the results vary. For some applications, the usage of this method could be successful but a common problem seems to be that the synthetic data lack sufficient realism or suffer from a lack of diversity.

Does this mean that the usage of GANs as an augmentation method is doomed? Not necessarily. For some types of data, such as time-series data, traditional augmentation methods are not suitable. Furthermore, this study has shown that the GANs seem to be able to generate highly-realistic samples, and therefore, it might be a question of finding a different architecture and/or better hyperparameters for creating samples with larger diversity. The fact that other studies have succeeded in improving classification accuracy is also promising and also indicates that there might be a question of finding a better GAN setup.

It should also be mentioned that there exist GANs tailored to dealing with classification problems. Such GANs has different loss functions and a discriminator that does not only discriminate between real and fake images, but also between different data classes. Those networks will not be discussed here. However, one such combined GAN and augmentation method called deep adversarial data augmentation (DADA) is briefly described in Section 6.2.1.

6 | Conclusion

In the following sections, the thesis will be summarized and some additional relevant methods, which could be interesting to further investigate, are described.

6.1 Summing Up

In this thesis, recurrent GANs and conditional recurrent GANs have been trained to generate two different types of time-series data: univariate time-series consisting of sinusoidal waves and multivariate radar time-series. An important part of the thesis work has been to examine different methods for evaluating synthetic GAN data. Two distance-based kernel metrics, maximum mean discrepancy and energy distance, were examined for monitoring the training progress and selecting a suitable generator model for the generation of synthetic samples. For evaluating the quality of the synthetic dataset, the data was tested in a classification setting with several experiments aiming at examining the sample quality and the in-class diversity.

Overall, both GAN architectures tested seem to be able to create highly realistic time-series, both simpler and more complex ones. However, not all hyperparameter settings and sampling methods tested yield realistic samples. Furthermore, it was shown that the synthetic data could suffer from loss of in-class diversity as well as a distortion of the decision boundary region, making the synthetic samples easier for a classifier to classify when trained on both real and synthetic data.

The contribution of this thesis is to show that GANs can be a feasible method for generating realistic synthetic time-series data in military applications. Although adding synthetic samples to the real data did not improve the classification performance trained on such a mixture, highly realistic data could be of interest for other usages. Furthermore, the thesis has contributed with the insight that two such simple metrics as MMD and ED could be a more useful stopping criterion compared to loss monitoring and visual inspection for GAN training. Lastly, the thesis has shown that under some circumstances, it is possible to improve the classification performance using synthetic data. Even if the gain in performance so far is small, adjustments of the network architectures and hyperparameters could generate even more diverse synthetic data.

6.2 Future Work

In this thesis, generative adversarial networks have been used to generate synthetic data for augmentation of multivariate time-series. During the project, several other methods came to mind that might be useful to study as a continuation and extension of this project. Some such methods will be briefly described in the following sections.

6.2.1 GANs Tailored for Data Augmentation in Classification Tasks

In this project, conditional recurrent GANs have been used to generate synthetic data, which is then evaluated using a classifier. This makes the method used in this thesis a two-step process. An alternative approach would be to combine the training and the evaluation process. Such an approach is described in a 2018 article, where the authors propose a new type of GAN called deep adversarial data augmentation (DADA), shown to be efficient even in the extremely low data regime [49].

The proposed structure of the DADA GAN is similar to an ordinary GAN. The generator, in the paper called *augmenter*, produces synthetic samples from a latent vector and is conditioned on the class labels. The discriminator differs from the ordinary discriminator of GANs in mainly two aspects: firstly, instead of outputting a single value indicating the probability of the sample being real, the discriminator outputs $2k$ values, where k is the number of classes in the dataset. The first k outputs represent the probabilities of the input sample belonging to one of the k real classes and the second k outputs represent the probabilities of the input belonging to any of the k fake classes. In the ideal case, the discriminator should be unable to discriminate between real and fake samples. The second aspect where the discriminator differs is that the loss is computed over those $2k$ classes, using the cross-entropy loss function instead of the binary cross-entropy loss.

The mentioned paper performed several experiments on common benchmarking datasets such as CIFAR-10 and CIFAR-100. By selecting a small subset of the datasets for training and hence simulating a low data regime, the authors could present significantly better results using DADA compared to using synthetic data from for instance ordinary GANs. Using such a network to generate synthetic data could potentially create more diverse data and improve the results presented in this thesis further. Furthermore, the DADA method is interesting to consider for other reasons too. In military applications, as briefly discussed in Section 1.1, obtaining training data can be difficult due to the highly sensitive nature of such data. That means, that developing methods that work well in the extremely low data regime is appealing since a minimal amount of real data would be needed to train the models.

6.2.2 Normalizing Flows

A family of alternative generative models that have been developed to take care of some of the shortcomings of GANs, such as an unstable training process and certain types of covariate shifts, is *normalizing flows*. This family of generative models provides tractable likelihoods and the possibility of an exact evaluation of the probability density of new observations [2]. The idea is that an initial, simple density is transformed into a more complex one through a series of differentiable and invertible transformations [50]. If $\mathbf{X} \in \mathbb{R}^d$ is a random variable with a tractable distribution $p_{\mathbf{X}} : \mathbb{R}^d \to \mathbb{R}$ (for instance the multivariate standard normal distribution), $\mathbf{Y} = \mathbf{f}(\mathbf{X})$ an invertible and differentiable function, and $\mathbf{X} = \mathbf{g}(\mathbf{Y})$ is the inverse function of $\mathbf{f}$, then we may use the change of variable formula to obtain the density of the transformed variable $\mathbf{Y}$ [2]:

$$p_{\mathbf{Y}}(\mathbf{y}) = p_{\mathbf{X}}(\mathbf{g}(\mathbf{y})) \left| \det \frac{\partial \mathbf{g}}{\partial \mathbf{y}} \right| = p_{\mathbf{X}}(\mathbf{g}(\mathbf{y})) \left| \det \frac{\partial \mathbf{f}}{\partial \mathbf{g}(\mathbf{y})} \right|^{-1}. \tag{6.1}$$

The transformation of the initial density of the random variable $\mathbf{X}$ towards the final, more complex density is called the *generative direction*. The opposite direction is called the *normalizing direction*. The name "normalizing flows" is based on this "normalizing" operation of the more complex density. Also, the simple initial density is in practice often a normal density, which is another reason for the name "normalizing flows". Note that in order to obtain the final density, multiple transformations with different functions $\mathbf{f}_i$ is usually performed. Given that the transformation $\mathbf{f}_i$ can be arbitrarily complex, it has been theoretically shown that normalizing flows can be used to generate any density $p_{\mathbf{Y}}$ from any simple distribution $p_{\mathbf{X}}$ [2].

Normalizing flows was proposed in 2015 and has gained popularity during the last few years. However, the usage of normalizing flows is not widely studied for time-series data, let alone multivariate continuous time-series data. However, a recent study investigated the usage of conditional normalizing flows in combination with an autoregressive model for multivariate time-series data forecasting [51]. The paper states that the authors received state-of-the-art results using this method. Furthermore, a slightly older paper investigated the usage of normalizing flows for anomaly detection in an industrial time-series dataset [52]. Also this paper presents promising results compared to a standard method for anomaly detection called Local Outlier Factor (LOF).

6.2.3 Topological Data Analysis

Topological data analysis (TDA) is a statistical field based on the idea that datasets can be analyzed using methods developed from algebraic topology and computational geometry [53]. While analysis of high-dimensional datasets can be difficult due to sensitivity to noise and restrictions to specific metrics, TDA is robust to noise and the usage of the method is independent on the distance metric chosen (however, the choice of metric might affect the quality of information obtained from the analysis). Such an analysis method could be useful to analyze

the structure of for instance the multivariate time-series dataset explored in this thesis and how synthetic such data differs from real data.

Intuitively, topological data analysis is about analyzing the high-dimensional topological and geometrical shapes of the data in some metric space. It is assumed that the input data is a finite set of points, with some distance metric associated with it [53]. It must not necessarily be a Euclidean distance metric but could be any finite metric space. The set of points is assumed to be a discrete representation of a continuous space, and therefore, a continuous representation of the data is needed: the points themselves do not hold any topological information. Obtaining a continuous data representation is usually done by enclosing all points with the smallest shape that encloses them all: in mathematical contexts usually referred to as the *complex hull*. A *k-simplex*, spanned by $k+1$ independent points X, is the complex hull of X. In Figure 6.1, four examples of k-simplices are presented for $k = 0$, $k = 1$, $k = 2$, and $k = 3$.

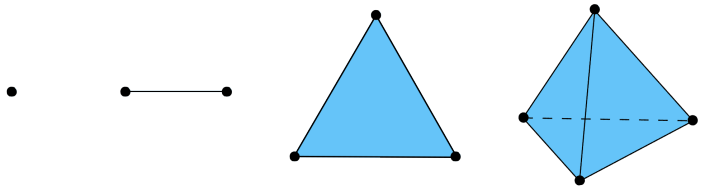


Figure 6.1: Four examples of k-simplices for $k = 0$, $k = 1$, $k = 2$, and $k = 3$. A *k-simplex*, spanned by $k+1$ independent points X, is the complex hull of X, or more intuitively: the smallest shape that encloses all $k+1$ points in X.

A collection of simplices is called a *geometric simplicial complex* K in $\mathbb{R}^d$. Such collections are created by connecting nearby k-simplices. A geometric simplicial complex K spans a subspace of $\mathbb{R}^d$ which is called the *underlying space* of K [53]. This underlying space inherits the topology of $\mathbb{R}^d$, and can hence be used for topological analysis.

There are many ways to form geometric simplicial complexes from data. One natural way is to extend the idea of forming α-neighboring graphs. Such a graph is formed by drawing edges between vertices located closer than some distance α from each other. If two vertices are closer than α to each other, then a line (1-simplex) can be drawn between them. If three vertices are closer than α from each other, then a triangle (2-simplex) can connect them. If k vertices are closer than α from each other, then they can be connected by a k-simplex. The result is a geometric simplicial complex.

Using this continuous representation of the dataset, one can utilize different tools to analyze the dataset. Such tools will not be described in depth here, as there already exist many well-written summaries of TDA methods such as [53], just to mention one example. However, one method deserves to be mentioned since it is by far the most common tool used in general within TDA. It is called *persistent homology*. The idea is to count the number of n-dimensional holes in the dataset, in a structured way: the data is sliced (intuitively similar to an MRI scan of human organs), and the holes are counted to investigate the importance of the holes in the data. To put it simply, holes that are present in many such slices are more important than holes that are

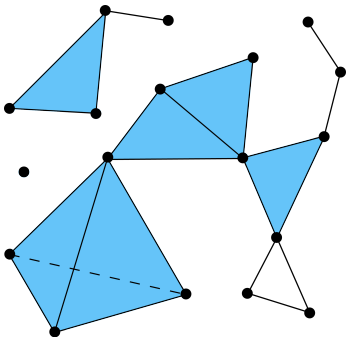


Figure 6.2: An example of a geometric simplicial complex, consisting of several k-simplices.

present in fewer. The persistent homology can be summarized in *barcode diagrams*, and with the help of those diagrams, two spaces can be statistically compared. This could be interesting for several reasons. Firstly, it could be an interesting method to visualize the multivariate tracker data used in this thesis. Secondly, it could be interesting to compare samples from the real data and synthetic data using TDA. TDA is robust even for small sample sizes, which suits this application well. Furthermore, a recent study developed new metrics based on persistent homology and used it to obtain good results in several univariate time-series classification task, compared to conventional methods such as dynamic time warping (DTW) [54].

There exist several robust and well-documented libraries for topological data analysis in Python, R, and C++. In Python, `scikit-tda` and `GUDHI` are two examples.

Bibliography

[1] Connor Shorten and Taghi M. Khoshgoftaar. “A Survey on Image Data Augmentation for Deep Learning”. In: *Journal of Big Data* (2019).

[2] Ivan Kobyzev, Simon Prince, and Marcus A Brubaker. “Normalizing Flows: An Introduction and Review of Current Methods”. In: *arXiv preprint arXiv:1908.09257v2* (2019).

[3] Henrik Andersson and Chi Thong Luong. “Classification Between Birds and UAVs Using Recurrent Neural Networks”. Gothenburg: Chalmers University of Technology, 2019.

[4] Shibani Santurkar, Ludwig Schmidt, and Aleksander Madry. “A Classification-Based Study of Covariate Shift in GAN Distributions”. In: *arXiv preprint arXiv:1711.00970* (2017).

[5] Cristóbal Esteban, Stephanie L. Hyland, and Gunnar Rätsch. “Real-valued (Medical) Time Series Generation with Recurrent Conditional GANs”. In: *arXiv preprint arXiv:1706.02633v2* (2017).

[6] Yann LeCun. *The MNIST database of handwritten images*. 1998. URL: `http://yann.lecun.com/exdb/mnist/` (visited on 02/07/2020).

[7] Anne Marie Delaney, Eoin Brophy, and Tomas E. Ward. “Synthesis of Realistic ECG using Generative Adversarial Networks”. In: *arXiv preprint arXiv:1909.09150v1* (2019).

[8] Fei Zhu et al. “Electrocardiogram Generation With a Bidirectional LSTM-CNN Generative Adversarial Network”. In: *Scientific Reports* 9 (2019), pp. 1–11.

[9] Dan Li et al. “Anomaly Detection with Generative Adversarial Networks for Multivariate Time Series”. In: *arXiv preprint arXiv:1809.04758v3 [cs.LG]* (2018).

[10] Dan Li et al. “MAD-GAN: Multivariate Anomaly Detection for Time Series Data with Generative Adversarial Networks”. In: *arXiv preprint arXiv:1901.04997v1 [cs.LG]* (2019).

[11] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning. Data Mining, Inference, and Prediction*. Springer, 2009.

[12] Bernhard Mehlig. “Artificial Neural Networks”. In: *arXiv preprint arXiv:1901.05639v2 [cs.LG]* (2019).

[13] Warren S. McCulloch and Walter H. Pitts. “A Logical Calculus of the Ideas Immanent in Nervous Activity”. In: *Bullentin of Mathematical Biophysics* (1943).

[14] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[15] Diederik P. Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: *arXiv preprint arXiv:1412.6980 [cs.LG]* (2014).

[16] Gareth James et al. *An Introduction to Statistical Learning. With Applications in R*. Springer, 2013.

[17] Nitish Srivastava et al. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *Journal of Machine Learning Research* 15 (2014), pp. 1929–1958.

[18] Qing-Song Xu, Yi-Zeng Liang, and Yi-Ping Du. “Monte Carlo Cross-Validation for Selecting a Model and Estimating the Prediction Error in Multivariate Calibration”. In: *Journal of Chemometrics* 18 (2004), pp. 112–120.

[19] Eric W. Weisstein. *Convolution*. 2008. URL: `http://mathworld.wolfram.com/Convolution.html` (visited on 03/16/2020).

[20] Andrej Karpathy. *Convolutional Neural Networks for Visual Recognition*. 2019. URL: `http://cs231n.github.io/convolutional-networks/` (visited on 03/16/2020).

[21] Christopher Olah. *Understanding LSTM Networks*. 2015. URL: `https://colah.github.io/posts/2015-08-Understanding-LSTMs/` (visited on 03/20/2020).

[22] Ian Goodfellow et al. “Generative Adversarial Nets”. In: *Advances in neural information processing systems*. 2014, pp. 2672–2680.

[23] Ian Goodfellow. “NIPS 2016 Tutorial: Generative Adversarial Networks”. In: *arXiv preprint arXiv:1701.00160v4 [cs.LG]* (2017).

[24] Mehdi Mirza and Simon Osindero. “Conditional Generative Adversarial Nets”. In: *arXiv preprint arXiv:1411.1784v1 [cs.LG]* (2014).

[25] Henrik Arnelid, Edvin Listo Zec, and Nasser Mohammadiha. “Recurrent Conditional Generative Adversarial Networks for Autonomous Driving Sensor Modelling”. In: *IEEE Intelligent Transportation Systems Conference (ITSC)* (2019).

[26] Jon Gauthier. *Conditional Generative Adversarial Nets for Convolutional Face Generation*. Tech. rep. Symbolic Systems Program, Natural Language Processing Group, Stanford University, 2015.

[27] Levent Karacan et al. “Learning to Generate Images of Outdoor Scenes from Attributes and Semantic Layouts”. In: *arXiv preprint arXiv:1612.00215v* (2016).

[28] Tim Salimans et al. “Improved Techniques for Training GANs”. In: *arXiv preprint arXiv:1606.03498v1 [cs.LG]* (2016).

[29] Alec Radford, Luke Metz, and Soumith Chintala. “Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks”. In: *arXiv preprint arXiv:1511.06434v2 [cs.LG]* (2015).

[30] Christian Szegedy et al. “Rethinking the Inception Architecture for Computer Vision”. In: *arXiv preprint arXiv:1512.00567v3 [cs.CV]* (2015).

[31] Arthur Gretton et al. “A Kernel Two-Sample Test”. In: *Journal of Machine Learning Research* 13 (2012), pp. 723–773.

[32] Dougal J. Sutherland et al. “Generative Models and Model Criticism via Optimized Maximum Mean Discrepancy”. In: *arXiv preprint arXiv:1611.04488v5 [stat.ML]* (2016).

[33] Dino Sejdinovic and Arthur Gretton. *What is an RKHS?* 2014. URL: `http://www.stats.ox.ac.uk/~sejdinov/teaching/atml14/Theory_2014.pdf` (visited on 03/24/2020).

[34] Gábor J. Székely and Maria L. Rizzo. “Energy Statistics: A Class of Statistics Based on Distances”. In: *Journal of Statistical Planning and Inference* 143 (2013), pp. 1249–1272.

[35] Gábor J. Székely and Maria L. Rizzo. “Testing for Equal Distributions in High Dimension”. In: *InterStat* (2004).

[36] Gábor J. Székely and Maria L. Rizzo. “Energy Distance”. In: *WIREs Computational Statistics* (2016), 8:27–38.

[37] Gábor J. Székely. *$\mathscr{E}$-Statistics: The Energy of Statistical Samples*. Tech. rep. 02-16. Department of Mathematics and Statistics, Bowling Green State University, Ohio, 2002.

[38] Dino Sejdinovic et al. “Equivalence of distance-based and RKHS-based statistics in hypothesis testing”. In: *arXiv preprint arXiv:1207.6076v3 [stat.ME]* (2012).

[39] Josip Djolonga. *A PyTorch Library for Differentiable Two-Sample Tests*. 2017. URL: `https://github.com/josipd/torch-two-sample` (visited on 03/30/2020).

[40] Dougal J. Sutherland et al. “Generative Models and Model Criticism via Optimized Maximum Mean Discrepancy”. In: *arXiv preprint arXiv:1611.04488v5 [stat.ML]* (2016).

[41] Zhicheng Cui, Wenlin Chen, and Yixin Chen. “Multi-Scale Convolutional Neural Networks for Time Series Classification”. In: *arXiv preprint arXiv:1603.06995v4 [cs.CV]* (2016).

[42] Arthur Le Guennec, Simon Malinowski, and Romain Tavenard. “Data Augmentation for Time Series Classification using Convolutional Neural Networks”. In: *ECML/PKDD Workshop on Advanced Analytics and Learning on Temporal Data, September 2016, Riva Del Garda, Italy* (2016).

[43] Paul K. Rubenstein et al. “Practical and Consistent Estimation of f-Divergences”. In: *arXiv preprint arXiv:1905.11112v2 [stat.ML]* (2019).

[44] Fernando Pérez-Cruz. “Kullback-Leibler Divergence Estimation of Continuous Distributions”. In: *2008 IEEE International Symposium on Information Theory* (2008).

[45] Hideitsu Hino and Noboru Murata. “Information Estimators for Weighted Observations”. In: *Neural Networks* 46 (2013), pp. 260–275.

[46] Suman Ravuri and Oriol Vinyals. “Classification Accuracy Score for Conditional Generative Models”. In: *arXiv preprint arXiv:1905.10887v2 [cs.LG]* (2019).

[47] Luis Perez and Jason Wang. “The Effectiveness of Data Augmentation in Image Classification using Deep Learning”. In: *arXiv preprint arXiv:1712.04621v1 [cs.CV]* (2017).

[48] Fabio Henrique Kiyoiti dos Santos Tanaka and Claus Aranha. “Data Augmentation Using GANs”. In: *arXiv preprint arXiv:1904.09135v1 [cs.LG]* (2019).

[49] Xiaofeng Zhang et al. “DADA: Deep Adversarial Data Augmentation for Extremely Low Data Regime Classification”. In: *arXiv preprint arXiv:1809.00981v1 [cs.CV]* (2018).

[50] Danilo J. Rezende and Shakir Mohamed. “Variational Inference with Normalizing Flows”. In: *arXiv preprint arXiv:1505.05770v6* (2016).

[51] Kashif Rasul et al. “Multi-variate Probabilistic Time Series Forecasting via Conditioned Normalizing Flows”. In: *arXiv preprint arXiv:2002.06103v2 [cs.LG]* (2020).

[52] Maximilian Schmidt and Marko Simic. “Normalizing Flows for Novelty Detection in Industrial Time Series Data”. In: *arXiv preprint arXiv:1906.06904v1 [cs.LG]* (2019).

[53] Frédéric Chazal and Bertrand Michel. “An Introduction to Topological Data Analysis: Fundamental and Practical Aspects for Data Scientists”. In: *arXiv preprint arXiv:1710.04019v1 [math.ST]* (2017).

[54] Yu-Min Chung, William Cruse, and Austin Lawson. “A Persistent Homology Approach to Time Series Classification”. In: *arXiv preprint arXiv:2003.06462v1 [stat.ME]* (2020).