SPA6776 Extended Independent Project (30-credits) - 3rd year BSc

Study of Generative Machine Learning Methods for Particle Physics

Student:
Sergio Alfonso Calvo Ordoñez
s.calvoordonez@se19.qmul.ac.uk

Supervisor:
Professor Adrian Bevan
a.j.bevan@qmul.ac.uk

School of Physical and Chemical Sciences
April 23, 2022

Acknowledgements

I especially wanted to thank Prof Adrian Bevan, my dissertation supervisor, for motivating and teaching me the best practices of AI in Physics and Dr Craig Agnor, my academic advisor, for his comprehensive introduction to Scientific Computing that sparked my passion for Data Science. I would also like to mention Martin Fanev for being my project partner and a true friend.

Furthermore, I would also like to thank my close group of friends that have helped me keep on track and have made my university experience unforgettable. Last but not least, I will never be thankful enough to my parents, Gustavo Calvo and Marlyn Ordoñez, as well as my siblings, Marlyn Palomino and Luis Palomino, for their unending support and for making my success possible.

Abstract

Reliable modelling of experiments in particle physics is crucial to build and assemble the modern and high performing detectors being used nowadays in the largest laboratories in the world. As well, simulations are used as a tool to study and interpret the resulting experimental data. The current and most generalised method used for these simulations, as of the date of this document, is Monte Carlo simulations. It is commonly known that this alternative is highly computationally heavy and therefore not efficient enough for the vast demand coming from the scientific community.

In this paper, generative machine learning models will be considered as a more efficient and scalable mechanism to generate these events. More specifically, the most relevancy will be given to generative adversarial networks (GANs) and variational autoencoders (VAE). The main focus will be to dig up into the essence of these models, and, through methods like data transformations, try to solve the problems arising when training with normally distributed data in different dimensions.

Contents

List of Figures

List of Tables

1 Introduction

In the short term future, experiments at the Large Hadron Collider (LHC) are expected to involve an exponentially larger amount of data compared to the past or even current experiments. This means that, at the same time, the current strategies used to produce reliable simulations will suffer from technological limitations and, hopefully, also improvements. This could potentially become a problem that stops the advancement of science and makes it hard, for other than the best-funded projects, to take place. As higher physics precision is required to increase for future experiments, the CPU performance has to also keep up.

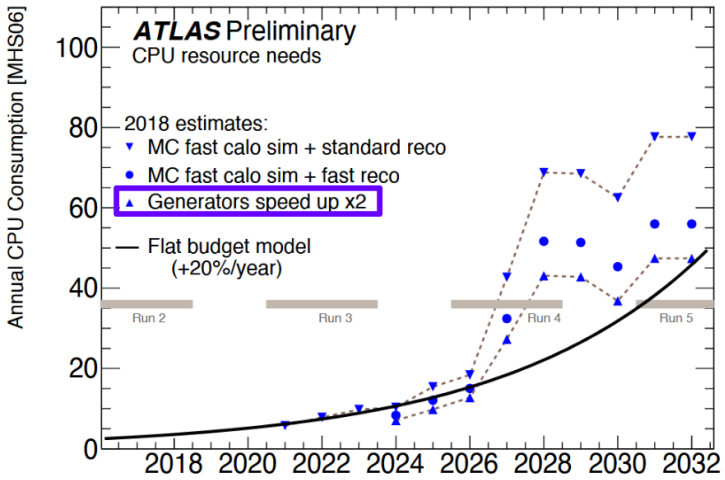


Figure 1: ATLAS estimated CPU usage from previous years extrapolated for the future [44].

Fig. 1 shows that in ATLAS specifically, it has been possible to keep up with the simulations requirements and CPU resources needed for the past years, and it is also predicted that we will be able to keep that up for the upcoming few years. However, from approximately 2026 onward, there will be an exponentially high demand for CPU resources for an almost flat (in comparison) increase in the budget annually. The LHC experiments expectation is that the exploitation of the High Luminosity LHC will require a factor of a hundred improvement in simulation throughput with the same computing resources available today [45].

At the moment, these big laboratories such as ATLAS and CMS use Monte Carlo based event generators such as MADGRAPH5 [6], PYTHIA8 [43], SHERPA [12], POWHEG-BOX [38], MC@NLO [1], and HERWIG7 [10] to create simulations of the hard-scattering process. Best estimates suggest that the simulation of a single event takes already several minutes [16], with $O(10^9)$ events to be generated for each simulation campaign, leading to a huge computational footprint both in terms of CPU usage and disk space [16].

The scientific community and high-energy physics laboratories around the world find themselves in a situation where, in order to study and discover new physics, they are using simulation methods that go back decades in time. Taking this and the exponential evolution

of technology over the past several years into account already suggests the inevitable problems to come. It is clear the need for alternatives, however, one must consider the difficulty of the problem due to the detail-oriented data to simulate. Although there is the possibility of using Monte Carlo fast calorimeter simulations, for example, these simulations should still be reliable and contain a considerable level of detail that contributes to their study. Not only this, but these fast simulations still make use of a big portion of the estimated resources necessary for event generation over the next few years. This is not enough.

One of the solutions that have been looked into, is machine learning (ML), more specifically, Deep Neural Networks (DNN). Some Generative Adversarial Networks [22], an unsupervised learning technique, and architectures have been proposed during the past years. More widely, GANs have been most commonly used for tasks such as photo-realistic portraits generation, music generation, or even art. Their ability to learn latent and meaningful features from the training data and then reproduce it to a high level of quality is a characteristic that differentiates them from many other generative modelling methods. Variational autoencoders [28] are an alternative type of models that also have shown a strong ability to learn target distributions and sample from them meaning that they are capable of generating data to a good level of fidelity.

Although we will be discussing both GANs and VAEs, this paper will be oriented towards focusing in the GAN model. Despite the fact that GANs are relatively recent (2014), there has been a huge interest in the matter that has been accompanied by a surprising progress in the applications and improvements to the original model. In chronological order we could mention several architectures that have been historically relevant, for example, DCGAN [41], CGAN [36], CycleGAN [47], CoGAN [33], ProGAN [27], WGAN [7], SAGAN [46], BigGAN [13], StyleGAN [26], etc. With time, these models not only obtained better performance results, but also increased quite considerably in model complexity. For the purpose of the goals of this paper, we will be using default GANs and DCGANs as well as mentioning WGANs as a potential improvement.

GANs can be several orders of magnitude faster than the event generators that are trying to mimic, and this speed-up can potentially be leveraged to bring down the overall computational cost of the data generation process. In order to train these models, we would still have to make use of data generated by Monte Carlo methods, so it is necessary to take into consideration an amount of training data that would not shorten the overall simulation process, as otherwise, the approach would be unhelpful. We do not want to spend all the CPU power that initially is the problem simply on generating a "big enough" training dataset. In addition, it is a common misconception to believe that it is possible to achieve statistically better than the data it was trained on. Features cannot be resolved from the training data and cannot be extrapolated from the training data [35].

To date, in the literature, we can identify the low performance of generative models such as the ones mentioned above (within the particle physics domain). We suspect that this underperformance is related to the behaviour of the true distributions in nature when it comes to particle physics events. The histograms representing these collisions distributions

tend to be normally distributed, meaning that there is a considerable concentration of the points around the mean leaving much fewer samples on the tails. Our hypothesis consists of associating the lack of accuracy of the generation to the models undergoing overfitting around the mean and a degree of underfitting as the number of standard deviations from the mean increases.

Throughout this document, the hypothesis above will be explained more in-depth and it will be proven why it does not work. Instead, we will take a hands-on approach to investigate the problem from a closer perspective and, unlike previous papers seen, develop a framework that allows a comprehensive numerical comparison between generated results and training samples to determine, in a justifiable manner, whether our results are valid or not. Important propositions for future work, potential alternatives, and different experiments will also be discussed in the latter chapters clarifying the open-ended nature of the problem. The principal aim of this investigation was to achieve a proficient understanding of the systems and models used to leave the door open for new ideas that, hopefully, contribute to the solution of the problem in the future.

Lastly, we would like to mention that all of the code (expanding on the Listings included at the end of this paper) can be found publicly in github on the following link: Gihub repository: "generative_ml_for_particle_physics".

2 Generative methods in particle physics

We based this work in the excellent notes on Monte Carlo techniques and Monte Carlo events generators made available by the Particle Data Group. Equally, we also got inspiration from the book in [21], more specifically regarding chapter 17.

2.1 What is generative modelling?

A generative model describes how a dataset is generated in terms of a probabilistic model. By sampling from this model, new data is generated [18]. Before diving into the field of particle physics in the next section, a simpler example will be used to describe what generative modelling is, and in what cases it can be beneficial.

Suppose there is a problem for which an unusual amount of images of houses/apartments is needed. There are already datasets containing images of this class, however, for this case, it is useful to generate even more images that have never existed but seem real thanks to a model that has learnt the general rules on how do houses/apartments look like. One of the best solutions for this type of problem is using generative modelling. In order to efficiently use this technique, a dataset containing many examples of images, in this case, of houses and apartments is required. These images will be used as part of the *training data* - a subset of the original dataset that will consist of many *data points* (each data point represents one image). Generally, each data point consists of a large number of *features*, this depends on the specific data but for the example provided, each feature would correspond to each pixels' value in the image.

Furthermore, generative models are probabilistic, one should not identify them as deterministic as the model does not simply execute a fixed computation. There is a stochastic component to the models, they do not output the same result for the same input repeatedly, the randomness influences the individual samples generated by the model. In other words, there is a probabilistic distribution, from which each individual generated data point is sampled, that explains the criteria used by the model to mimic the training dataset.

2.2 Monte Carlo Techniques

Monte Carlo methods are commonly used as a solution to be able to evaluate complicated integrals or to sample random variables from a complex probability density distribution. These algorithms return answers with a random amount of error. The amount of error can be reduced by expending more computational resources, however, as expressed in section 1, this is not a viable alternative. In this subsection we will describe a few important methods for sampling some typical probability density functions (PDFs).

2.2.1 Basics of Monte Carlo sampling

The most basic type of Monte Carlo sampling is usually employed to compute sums or integrals that cannot be computed exactly. The idea is to view the sum or integral as if

it were an expectation under some distribution and to approximate the expectation by a corresponding average [21]. Let a specific sum or integral be rewritten as an expectation:

$$s = \sum_{\mathbf{x}} p(\mathbf{x}) f(\mathbf{x}) = E_p[f(\mathbf{x})], \tag{2.1}$$

or:

$$s = \int p(\mathbf{x}) f(\mathbf{x}) d\mathbf{x} = E_p[f(\mathbf{x})], \tag{2.2}$$

these equations have the constraint that p is a probability distribution in the case of the sum and a probability density over the variable $\mathbf{x}$ for the integral. Here is the moment when sampling comes into play. We can use it to approximate s simply by drawing n samples from p and then computing an average.

$$\hat{s}_n = \frac{1}{n} \sum_{i=1}^{n} f(x^{(i)}). \tag{2.3}$$

It is achievable to explain this step by recurring to a few characteristics of the system. Firstly, note that the estimator $\hat{s}$ is unbiased:

$$\mathbb{E}[\hat{s}_n] = \frac{1}{n} \sum_{i=1}^{n} \mathbb{E}[f(\mathbf{x}^{(i)})] = \frac{1}{n} \sum_{i=1}^{n} s = s. \tag{2.4}$$

Secondly, the **law of large numbers** also applies in this case. This law implies that, with a sufficient amount of samples, the average will eventually converge to the expected value.

$$\lim_{n \to \infty} \hat{s}_n = s. \tag{2.5}$$

With the condition that the variance of the individual terms is bounded.

$$Var[\hat{s}_n] = \frac{1}{n^2} \sum_{i=1}^{n} Var[f(\mathbf{x})] \tag{2.6}$$

$$= \frac{Var[f(\mathbf{x})]}{n}. \tag{2.7}$$

As n increases, the variance of the estimator will converge to 0 as long as $Var[f(\mathbf{x}^{(i)})] < \infty$.

The **central limit theorem** states that the distribution $\hat{s}_n$, converges to a Gaussian distribution with mean s and the variance shown in equation 2.7. This opens up the opportunity of estimating confidence intervals around the estimate $\hat{s}_n$, using the cumulative distribution of the normal density.

2.2.2 Importance Sampling

Note that, in equation 2.2, the integrand was divided into two different functions, $p(\mathbf{x})$ and $f(\mathbf{x})$, choosing this split is an important task that has to be decided carefully. More specifically, $p(\mathbf{x})$ represents the probability distribution of $\mathbf{x}$, while $f(\mathbf{x})$ is the function for which the expected value under $p(\mathbf{x})$ needs to be approximated. In the immense amount of combinations that could form this division relies the difficulty of the decision. As a general rule, $p(\mathbf{x})f(\mathbf{x})$ can be expressed in the following way:

$$p(\mathbf{x})f(\mathbf{x}) = q(\mathbf{x})\frac{p(\mathbf{x})f(\mathbf{x})}{q(\mathbf{x})}, \tag{2.8}$$

where $q(\mathbf{x})$ is the distribution from where we sample, and the rest of the equation averages.

We call optimal importance sampling to the optimal choice of q, q^*, and its derivation is relatively simple. Taking into account equation 2.8, a Monte Carlo estimator becomes:

$$\hat{s}_n = \frac{1}{n}\sum_{i=1,\mathbf{x}^{(i)}\sim q}^{n} f(x^{(i)}), \tag{2.9}$$

and can be turned into an estimator for importance sampling:

$$\hat{s}_q = \frac{1}{n}\sum_{i=1,\mathbf{x}^{(i)}\sim q}^{n} \frac{p(\mathbf{x}^{(i)})f(\mathbf{x}^{(i)})}{q(\mathbf{x}^{(i)})}. \tag{2.10}$$

Because the expectation value of this estimator does not depend on q:

$$\mathbb{E}_q[\hat{s}_q] = \mathbb{E}_q[\hat{s}_p] = s. \tag{2.11}$$

Lastly, the optimal q will be at the point for which the variance is minimum. A variance given by:

$$Var[\hat{s}_q] = \frac{Var[\frac{p(\mathbf{x})f(\mathbf{x})}{q(\mathbf{x})}]}{n}. \tag{2.12}$$

The minimum of the above expression, and therefore q^*, happens when:

$$q^*(\mathbf{x}) = \frac{p(\mathbf{x})|f(\mathbf{x})|}{Z}, \tag{2.13}$$

where Z is the normalisation constant. Better importance sampling distributions put more weight where the integrand is larger [21]. As a side note, it might be relevant to mention that importance sampling is also widely used to estimate the log-likelihood in deep directed models just like the variational autoencoder (see section 3.1).

2.2.3 Markov Chain Monte Carlo methods

Markov Chain Monte Carlo methods (MCMC) is the family of algorithms that use the mathematical tool called **Markov chain** to approximate samples from $p_{model}(\mathbf{x})$. This is usually necessary when there are no analytical methods for drawing exact samples from the distribution $p_{model}(\mathbf{x})$, normally seen when the model is represented by an undirected model.

The most popular MCMC are only applicable when the model does not assign a zero probability to any state along with the distribution. To achieve such a requirement, it is convenient to present these techniques as sampling from an energy-based model (EBM):

$$p(\mathbf{x}) \propto e^{-E(\mathbf{x})}. \tag{2.14}$$

The concept of a Markov chain is to have a initial arbitrary state $\mathbf{x}$ and keep randomly updating it over time. At some point, $\mathbf{x}$ will become a really approximated sample to the distribution $p(\mathbf{x})$. Markov chains are defined by the random state $\mathbf{x}$ and a transition distribution $T(\mathbf{x}^{'}|\mathbf{x})$ where $\mathbf{x}^{'}$ is the next updated state following $\mathbf{x}$. The algorithms that apply Markov chains keep updating $\mathbf{x}$ with the new state $\mathbf{x}^{'}$ sampled from $T(\mathbf{x}^{'}|\mathbf{x})$.

Metropolis-Hastings algorithm - An example: This algorithm is frequently used to sample multidimensional points $\mathbf{x}$ from a proportional PDF to a given target distribution, $p(\mathbf{x})$. A specific benefit of this method is that is not necessary for $p(\mathbf{x})$ to be normalised, fact that is useful in Bayesian statistics because in many cases the posterior probability densities have undetermined normalisation constants. The algorithm goes as follow:

1. Define the target distribution $p(\mathbf{x})$, and the proportional distribution, $g(\mathbf{x})$.

$$p(\mathbf{x}) \propto g(\mathbf{x}) \tag{2.15}$$

2. Set an initial value for $\mathbf{x}$, $\mathbf{x}_0$.

3. For $i = 1, ..., m, repeat$:

   (a) Sample a candidate $\mathbf{x}^* \sim q(\mathbf{x}^*|\mathbf{x}_{i-1})$, where q is the proposed distribution.

   (b) Compute the Hastings test ratio, α:

$$\alpha = \frac{g(\mathbf{x}^*)/q(\mathbf{x}^*|\mathbf{x}_{i-1})}{g(\mathbf{x}_{i-1})/q(\mathbf{x}_{i-1}|\mathbf{x}^*)} = \frac{g(\mathbf{x}^*)q(\mathbf{x}_{i-1}|\mathbf{x}^*)}{g(\mathbf{x}_{i-1})q(\mathbf{x}^*|\mathbf{x}_{i-1})} \tag{2.16}$$

   (c) If $\alpha \geq 1$ accept $\mathbf{x}^*$ and set $\mathbf{x}_i \longleftarrow \mathbf{x}^*$.

   If $0 < \alpha < 1$, accept $\mathbf{x}^*$ and set $\mathbf{x}_i \longleftarrow \mathbf{x}^*$ with probability α, or reject $\mathbf{x}^*$ and set $\mathbf{x}_i \longleftarrow \mathbf{x}_{i-1}$ with probability $1 - \alpha$

Steps (b) and (c) act as a correction since the proposal distribution q is not the target distribution g. At each step in the chain, a candidate is evaluated to decide if it's used as an update or to remain in its current state. This decision will be made in terms of evaluating

whether the potential update is advantageous or not. If it's advantageous the algorithm will get updated, if it's not, there is a chance the algorithm gets updated, only with probability α.

One careful choice that is required to be taken is related to the proposed distribution q. It is also possible to choose a distribution that does not depend on the previous $\mathbf{x}$. Another popular algorithm that does this is **random walk Metropolis-Hastings**. Here the proposed distribution is centered around the previous iteration.

2.3 Monte Carlo event generators

To introduce the concept of Monte Carlo within particle physics it is important to, firstly, understand the complexity of the computations involved in the simulations and see where and why MCM can be useful.

In particle physics, the probability that two particles collide and react in a specific way is referred to as **cross section**. It is challenging to calculate these probabilities/cross-sections as there are many involuted integrals over momentum space, in fact, for each of the particles involved in the interaction. One advantage for this that MCM have is that we probe randomly, there is no correlation between the events and therefore it is possible to use these as representatives of the computations. These event generators used the techniques displayed in previous sections along with section 2 in order to approximate the integrals. The specific details of the generators are quite complex and therefore, as it is not the main focus of the investigation, it has been decided to leave the technical details and inner workings outside of this paper. Again, we can mention the most famous generators together with their respective references for further information: MADGRAPH5 [6], PYTHIA8 [43], SHERPA [12], POWHEG-BOX [38], MC@NLO [1], and HERWIG7 [10].

3 Theoretical background in generative machine learning

In the following section, we base the structure and mathematical derivations on two deep learning books that manage to provide an adequate picture of the theoretical pillars which are relevant to our research. These texts are the following: *Advanced Deep Learning with Tensorflow with Tensorflow 2 and Keras* [9] and *Deep Learning* [21].

3.1 Variational autoenconders (VAEs)

The variational autoencoder [28] is a directed model that uses learned approximate inference and can be optimised by uniquely using gradient-based methods. They are useful to design complex generative models of data that can be after fitted into large datasets.

The model consists of two networks working together, an encoder and a decoder that act in accordance to a loss function. The structure is sequential in the sense that the data goes first through the encoder, $q(\mathbf{z}|\mathbf{x})$; gets compressed into a hidden representation, $\mathbf{z}$; goes into the decoder, $p_{model}(\mathbf{x}|\mathbf{z})$; and finally, we get our output reconstruction. More specifically, the decoder outputs parameters of the probability distribution of the data, and it includes weights and biases. Note that the encoder outputs parameters to $q(\mathbf{z}|\mathbf{x})$ which is a Gaussian probability density. We will sample from this distribution to get values with noise of $\mathbf{z}$.

The fact that the variational autoencoder is composed by two different networks and that these are simultaneously trained, provides an interesting property. This is that the model is enforced to learn a predictable coordinate system that the encoder can capture. This makes it an excellent manifold learning algorithm.

3.1.1 Principles of VAEs

As it can be expected, in a generative model, approximating the true distribution of the training dataset is often of interest.

$$\mathbf{x} \sim P_\theta(\mathbf{x}), \tag{3.1}$$

where θ represents the parameters achieved after training.

In general, when looking at machine learning applications, in order to perform inference it will be interesting to find $P_\theta(\mathbf{x}, \mathbf{z})$, the joint distribution between the inputs, $\mathbf{x}$, and latent variables, $\mathbf{z}$. These latent variables have the aim of encoding the properties of the dataset, they are usually unknown and allow the model to extrapolate features to the newly generated data. If we go back to the example used in section 2.1, the latent variables could be the rectangular shapes of the apartment buildings, the fact that there are doors, windows, etc.

One could state the following:

$$P_\theta(\mathbf{x}) = \int P_\theta(\mathbf{x}, \mathbf{z}) d\mathbf{z}. \tag{3.2}$$

This is because $P_\theta(\mathbf{x}, \mathbf{z})$ is pretty much a distribution of the input data points and their attributes. Therefore, $P_\theta(\mathbf{x})$ could be computed from the marginal distribution. This makes sense as, if all of the attributes are considered, we should end up with the distribution that describes the inputs. However, there is a problem with eq. 3.2, this is that it does not seem to have an analytic form, therefore it is intractable, it is not possible to differentiate with respect to its parameters. This does not allow room for optimization.

With Bayes' theorem, eq. 3.2 can be rewritten as:

$$P_\theta(\mathbf{x}) = \int P_\theta(\mathbf{z}|\mathbf{x})P(\mathbf{x})d\mathbf{z}. \tag{3.3}$$

where $P(\mathbf{z})$ is a prior distribution over just $\mathbf{z}$ and therefore not dependent on any observable. If $\mathbf{z}$ is discrete and $P_\theta(\mathbf{z}|\mathbf{x})P(\mathbf{x})$ is continuous, then $P_\theta(\mathbf{x})$ is a combination of Gaussians, on the other hand, if $\mathbf{z}$ is discrete instead, then $P_\theta(\mathbf{x})$ is an infinite combination of Gaussians.

As good as this might seem, in practice, applying eq. 3.3 to estimate without an appropriate loss function, after training, the model will reach to the conclusion that $P_\theta(\mathbf{z}|\mathbf{x}) = P_\theta(\mathbf{x})$ which is the trivial solution. This happens because the neural network simply ignores $\mathbf{z}$. Therefore, it will be required to use an alternative way of expressing the equation:

$$P_\theta(\mathbf{x}) = \int P_\theta(\mathbf{x}|\mathbf{z})P(\mathbf{z})d\mathbf{z}. \tag{3.4}$$

However, $P_\theta(\mathbf{z}|\mathbf{x})$ is also intractable. The goal of a VAE is to find a tractable distribution that closely estimates $P_\theta(\mathbf{z}|\mathbf{x})$, an estimate of the conditional distribution of the latent attributes, $\mathbf{z}$, given the input, $\mathbf{x}$. [9]

3.1.2 Variational Inference

Following the previous subsection, the problem is focused on how a VAE can infere $P_\theta(\mathbf{z}|\mathbf{x})$ if it is intractable. This is done thanks to an encoder, which is a model of variational inference.

$$Q_\phi(\mathbf{z}|\mathbf{x}) \approx P_\theta(\mathbf{z}|\mathbf{x}). \tag{3.5}$$

Here, $Q_\phi(\mathbf{z}|\mathbf{x})$ will provide a good approximation of $P_\theta(\mathbf{z}|\mathbf{x})$ and we will end up obtaining an expression that, unlike $P_\theta(\mathbf{z}|\mathbf{x})$, it is parametric and differentiable with respect to ϕ, this will allow optimisation. Normally, $Q_\phi(\mathbf{z}|\mathbf{x})$ is chosen to be a multivariate Gaussian:

$$Q_\phi(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\mathbf{z}, \mu(\mathbf{x}), diag(\sigma(\mathbf{x})^2)). \tag{3.6}$$

All of the parameters of this multivariate Gaussian are computed by the encoder part of the variational autoencoder, with the exception of the latent variables $\mathbf{z}$. This is implied from the fact that the covariance matrix is diagonal, e.g. $\mathbf{z}$ and $\mathbf{x}$ are completely uncorrelated (they are independent from each other).

From the nomenclature of the equations, we can deduce the following:

1. $Q_\phi(\mathbf{z}|\mathbf{x})$ generates a vector, $\mathbf{z}$, composed of latent variables, it acts as the encoder in an autoencoder model.

2. $P_\theta(\mathbf{x}|\mathbf{z})$ generates or reconstructs the input from the latent vector, $\mathbf{z}$. This time, this part of the model acts as the decoder.

There clearly is a relationship between both of these components. This is what must be identified in order to properly estimate $P_\theta(\mathbf{x})$, doing this will lead to the finding of a core equation of VAEs. The derivations starts with the computation of the Kullback-Leibler (KL) divergence [30] to calculate the distance between $Q_\phi(\mathbf{z}|\mathbf{x})$ and $P_\theta(\mathbf{z}|\mathbf{x})$:

$$D_{KL}(Q_\phi(\mathbf{z}|\mathbf{x})||P_\theta(\mathbf{z}|\mathbf{x})) = \mathbb{E}_{z\sim Q}[log(Q_\phi(\mathbf{z}|\mathbf{x})) - log(P_\theta(\mathbf{z}|\mathbf{x}))]. \tag{3.7}$$

Now, using Bayes' theorem:

$$P_\theta(\mathbf{z}|\mathbf{x}) = \frac{P_\theta(\mathbf{x}|\mathbf{z})P_\theta(\mathbf{z})}{P_\theta(\mathbf{x})}, \tag{3.8}$$

into eq. 7:

$$D_{KL}(Q_\phi(\mathbf{z}|\mathbf{x})||P_\theta(\mathbf{z}|\mathbf{x})) = \mathbb{E}_{z\sim Q}[log(Q_\phi(\mathbf{z}|\mathbf{x})) - log(P_\theta(\mathbf{x}|\mathbf{z})) - log(P_\theta(\mathbf{z}))] + log(P_\theta(\mathbf{x})). \tag{3.9}$$

Since $log(P_\theta(\mathbf{x}))$ is independent of $\mathbf{z} \sim Q$, then it is valid to take it out of the expectation. Rearranging:

$$log(P_\theta(\mathbf{x})) - D_{KL}(Q_\phi(\mathbf{z}|\mathbf{x})||P_\theta(\mathbf{z}|\mathbf{x})) = \mathbb{E}_{z\sim Q}[log(P_\theta(\mathbf{x}|\mathbf{z}))] - D_{KL}(Q_\phi(\mathbf{z}|\mathbf{x})||P_\theta(\mathbf{z})). \tag{3.10}$$

This results comes after also substituting:

$$\mathbb{E}_{z\sim Q}[log(Q_\phi(\mathbf{z}|\mathbf{x})) - log(P_\theta(\mathbf{z})] = D_{KL}(Q_\phi(\mathbf{z}|\mathbf{x})||P_\theta(\mathbf{z})). \tag{3.11}$$

Eq. 3.10 is an important equations for VAEs and, in it, we can identify several components. Firstly, on the left hand side there is the function being maximised, $log(P_\theta(\mathbf{x}))$ minus the distance between the estimated $Q_\phi(\mathbf{z}|\mathbf{x})$ and the true $P_\theta(\mathbf{z}|\mathbf{x})$, this distance can be seen as the error between the two quantities. Secondly, on the right hand side, the first term, $P_\theta(\mathbf{x}|\mathbf{z})$, resembles the decoder part of an autoencoder which takes samples from the inference model and reconstructs the input. The second term is simply another distance, more concretely, between $Q_\phi(\mathbf{z}|\mathbf{x})$ and the prior $P_\theta(\mathbf{z})$. Since the KL divergence is always positive, this last term is the lower bound of $log(P_\theta(\mathbf{x}))$, also known as the **variational lower bound**, we will be using it in the next section when discussing learning and optimisation.

3.1.3 Loss function

A key feature of variational autoencoders relies in its training. These networks can be trained maximising the variational lower bound $\mathcal{L}(Q)$ associated with $\mathbf{x}$ (a data point):

$$\mathcal{L}(Q) = \mathbb{E}_{\mathbf{z}\sim Q}\, log(P_\theta(\mathbf{z},\mathbf{x})) + \mathcal{H}(Q_\phi(\mathbf{z}|\mathbf{x})) \tag{3.12}$$

$$= \mathbb{E}_{\mathbf{z}\sim Q}\, log(P_\theta(\mathbf{x}|\mathbf{z})) - D_{KL}(Q_\phi(\mathbf{z}|\mathbf{x})||P_\theta(\mathbf{z})) \tag{3.13}$$

$$\leq log(P_\theta(\mathbf{x})). \tag{3.14}$$

We can see several known terms here. In equation 12 we find the joint log-likelihood of the hidden and visible variables, and the entropy of the encoder. This entropy facilitates the increase of the standard deviation of Q whenever it is chosen to be a Gaussian distribution with added noise. This means that it motivates the variational posterior to raise the probability mass on many z values that could have generated $\mathbf{x}$, rather than converging to just one point estimate. The second term of equation 13 uses the Kullback-Leibler divergence [reference] in order to make $Q_\phi(\mathbf{z}|\mathbf{x})$ and the model prior $P_\theta(\mathbf{z})$ approach each other by optimising the distance between each other.

Maximising variational lower bound by optimising the parameters ϕ and θ of the neural network means that:

- $D_{KL}(Q_\phi(\mathbf{z}|\mathbf{x})||P_\theta(\mathbf{z}|\mathbf{x})) \rightarrow 0$, e.g. the inference model improves its performance of encoding $\mathbf{x}$ in $\mathbf{z}$.
- $log(P_\theta(\mathbf{x}|\mathbf{z}))$ is being maximised, therefore, the decoder part gets better at reconstructing $\mathbf{x}$ from an input of a latent vector, $\mathbf{z}$.

If the decoder network does not properly reconstruct the data, we say that it parameterises a likelihood distribution that does not have high enough probability mass on the true data. Evidently, a deficient reconstruction implies that it will have a large cost in the loss function.

3.2 Generative adversarial networks (GANs)

One can understand GANs [22] as a generative modelling approach based on a "game" theoretic scenario in which two different networks compete. A **generative network** tries to deceive an adversary **discriminator network** (Fig. 2).

In practice, the basic idea is that the generator network produces samples $\mathbf{x} = g(\mathbf{z};\theta^{(g)})$ (generally Gaussian) while the discriminator network aims to accurately classify data points drawn from the generator and the training data set. As in any classification task, the discriminator network outputs a probability value given by $\mathbf{x} = d(\mathbf{x};\theta^{(d)})$, this number indicates whether the evaluated data point belongs to the generated distribution or if it has been drawn from the training data.

In other words, the input of the generator is noise, $\mathbf{z}$, while the output is synthesized data. On the other hand, the input for the discriminator can be either real data coming from the

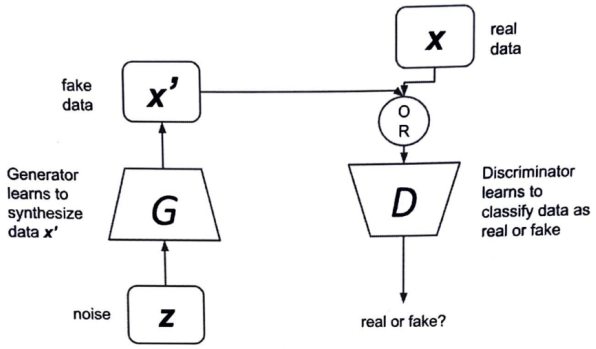


Figure 2: GAN diagram displaying the structure composed by two different networks, a generator G, and a discriminator D. The generator outputs fake data that then serves as an input to the discriminator which needs to learn to differentiate between these fake samples and real data coming from the training dataset. [9]

training dataset, or the synthesized data generated by the adversarial network. The true data is aimed to be classified with a label of 1.0 (100% probability of it being real), while the fake data should be classified with a label of 0.0 (0% chance of it being real). To learn to correctly distinguish between real data and fake data is the goal of the discriminator network. Because of the fact that this process is automated, GANs are considered an unsupervised learning approach.

3.2.1 Learning

At regular intervals, the output of the generator will be used as an input for the discriminator with the idea of pretending that this fake data is actually genuine and hoping that the classifier outputs 1.0. However, when the fake data is presented to the discriminator, naturally it will be classified as fake with a label close to 0.0. During training, the network's optimiser calculates the parameters for the generator, in every step, based on the current labels and its own prediction. The GAN will use backpropagation to take the gradients from the last layer of the discriminator all the way down to the first layer of the generator. This generator then uses these gradients to update its parameters and improve its performance during the next epochs. It is necessary to note that during this backpropagation process, the weights and biases from the discriminator are frozen for the interest of convergence.

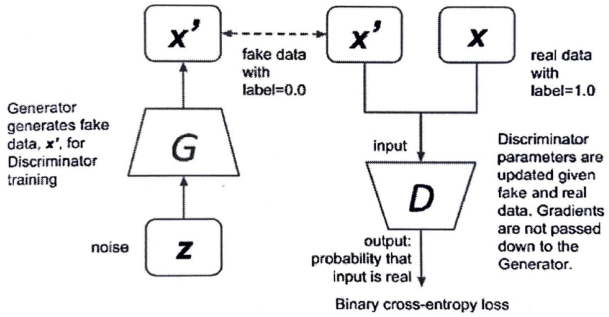


Figure 3: Diagram representing the training process of the discriminator is similar to training a binary classifier network using binary cross-entropy loss. The fake data is supplied by the generator, while the real data is from true samples. [9]

A way to define this process in generative adversarial networks is as a zero-sum game. A function $V(\theta^{(g)}, \theta^{(d)})$ regulates the optimisation of the discriminator, the generator uses the same function with opposite sign as its own payoff. Throughout the training stage (i.e. learning) both agents try to maximise their correspondent **V**. The discriminator is trained to maximise the probability of correctly labelling training examples and samples from **g** while, at the same time, the generator network trains to minimise $log(1 - d(g(\mathbf{z}))$. The idea is that, at convergence,

$$g^* = argmin_g max_d V(g, d). \tag{3.15}$$

The choice for V is,

$$V(\theta^{(g)}, \theta^{(d)}) = \mathbb{E}_{x \sim p_{data}} log(d(\mathbf{x})) + \mathbb{E}_{\mathbf{z}} log(1 - d(g(\mathbf{z}))). \tag{3.16}$$

This incentivises the discriminator to learn to correctly classify samples as real of fake (Fig. 3). At the same time, the generator tries to fool the classifier into believe its samples belong to the training set. Eventually, once training is over, the ideal situation would be to have the discriminator to output 0.5, meaning that it's indecisive to classify the data points that it is getting as inputs. One of the principle key features of GANs is that the learning process does not require approximate inference nor approximation of a partition function gradient, unlike many generative models. If $max_d v(g, d)$ is convex in $\theta^{(g)}$, then the procedure **will** converge.

It is important to realise that we can define the loss function, $\mathcal{L}$, of the discriminator as the negative of the value function of the generator, $V^{(g)}$.

$$V^{(g)}(\theta^{(g)}, \theta^{(d)}) = -\mathcal{L}^{(d)}(\theta^{(g)}, \theta^{(d)}). \tag{3.17}$$

In order to minimise the loss function, the discriminator parameters, $\theta^{(d)}$, will be updated through backpropagation by correctly identifying the genuine data, $d(\mathbf{x})$, and the generated data, $1 - d(g(\mathbf{z}))$.

In terms of the generator network (Fig. 4), the GAN uses the both of the networks losses components as a zero-sum game. As expected from a zero-sum game, the generator loss is simply the negative of the discriminator loss function.

$$\mathcal{L}^{(g)}(\theta^{(g)}, \theta^{(d)}) = -\mathcal{L}^{(d)}(\theta^{(g)}, \theta^{(d)}). \tag{3.18}$$

As presented in Eq. 3.15, for the generator, the value function should be minimised, in contrast, from the discriminator's perspective, the value function should be maximised. What it is achieved when maximising with respect to $\theta^{(d)}$ is, as mentioned above, that the optimiser sends gradient updates towards the discriminator parameters to consider the generated data as real. Simultaneously, minimising with respect to $\theta^{(g)}$, the generator will be trained by the optimiser on how to fool the classifier. In practise, this theoretical framework is known to come with some limitations. Mainly, the discriminator network is usually confident about its classification predictions and therefore it will not update the GAN parameters. Furthermore,

the gradient updates are small and tend to be smaller as they propagate to the generator layers, this raises a convergence issue for the generator network. The solution is to redefine the loss function of the generator.

$$\mathcal{L}^{(g)}(\theta^{(g)}, \theta^{(d)}) = -\mathbb{E}_{\mathbf{z}}\, log(d(g(\mathbf{z}))). \tag{3.19}$$

This time, the loss function just maximises the likelihood of the discriminator misclasifying the samples by training the generator. It stops being a zero-sum game and it becomes heuristics-driven.

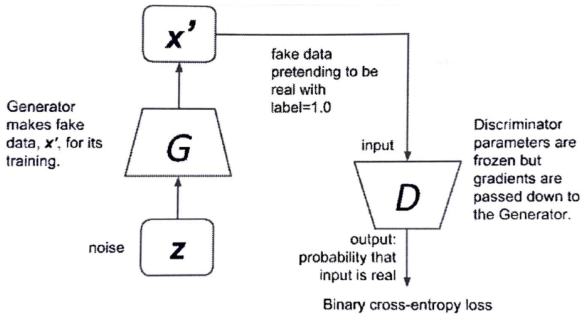


Figure 4: Training the generator is like training a network using a binary cross-entropy loss function. The fake data from the generator is presented as genuine. [9]

3.2.2 GANs challenges

The objective of the adversarial game between the generator and discriminator that constitutes training GANs, is to reach a concept called Nash equilibrium [39] which is widely used across fields such as game theory and economics. This equilibrium occurs when agents would not be better off changing their own strategy, assuming the other agents do not change theirs. The authors of the original GAN paper demonstrated that a GAN can only reach such equilibrium when the generator is producing perfect outputs (the estimated distribution completely matches the true distribution). This is theoretically appealling as it seems to suggest that by training the GAN for long enough, it will eventually reach perfection. However, in practice is not that simple; Nash equilibrium is not guaranteed.

It is well-know that GANs are one of the most important advancements in generative modelling. However, it is proved that there is a significant difficulty in terms of stability during training. In this section, we will explore some of the major problems encountered.

The first set of issues to be studied, will be related to the **loss oscillating**. It is common to observe the loss of the discriminator and generator start oscillating wildly rather than achieving convergence. Moderated oscillations are often common to see from batch to batch, however, in the long term one should be looking for a loss that stabilises, or at least increases or decreases gradually. Fig 5 shows an example of this issue.

The second challenge derived from training GANs is called **mode collapse**. This occurs when the generator finds a set of samples that successfully confuse the discriminator and

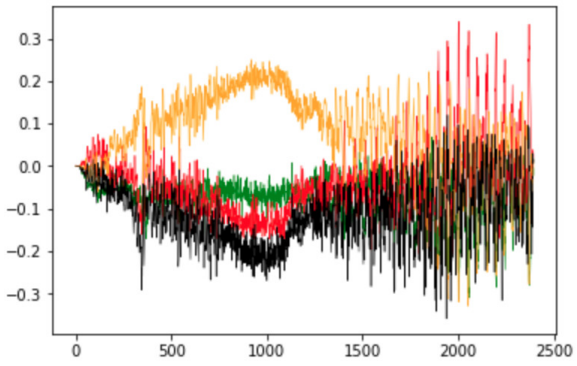


Figure 5: Oscillating loss in an unstable GAN. [19]

therefore it is not able to produce any examples other than these. This can happen in a case where the generator is being train over many different batches without updating the discriminator in the process. Here, the generator would be biased towards finding one observation that always fools the discrimination and would start mapping the whole latent space to this single observation. The loss function would quickly decrease to around 0, and the generator will stop having an incentive to keep improving his performance and, at the same time, come up with different outputs. The consequences of mode collapse can be seen in Fig. 6.

Figure 6: Mode collapse example using CIFAR-10 dataset.

Next, there is the phenomenon of **uninformative loss**. Ultimately, GANs can be regarded as a sort of min-max game, this means that the generator is solely evaluated against the discriminator network, this is how the loss is computed. However, since the discriminator keeps improving every iteration, it is not possible to compare the loss between two different iteration steps. This lack of correlation between the loss metric and the quality of the outputs can lead to confusion as, for example, there are cases in which, although the loss increases, the relative quality of the generated samples has also improved.

Lastly, another significant complication with the training process are the many different potential combinations of **hyperparameters**. This is due to the large number of possible hyperparameters to tune in both the discriminator and generator, parameters that affect processes like batch normalisation, learning rate, dropout, kernel size, batch size, activation layers, convolutional filters, etc. These networks are extremely sensitive to changes, and

finding the perfect setup is a task that cannot really be predicted and most of the times is achieved after an exercises of trial and error. This is the reason why it is key to understand the importance of each of the hyperparameter, how they work, and how the architecture of the GAN is relevant to them.

3.2.3 Tackling the GANs challenges

Thanks to the acknowledgement of these issues, during the past years, there have been several key advancements that have considerably improved the overall stability of GAN models and reduced the chances of stumbling upon those limitations. Throughout this section, a couple of methodologies will be explored that, through a set of modifications to the traditional architecture of GANs, manage to increase the odds of convergence. Expect that, in spite that it could be possible to dive deep into these concepts, for conciseness sake, they will be displayed with a summarised introduction.

1. **Wasserstein GAN (WGAN)** [7]: For this model, there were principally two main modifications (quoted from the paper):

   - A meaningful loss metric that correlates with the generator's convergence and sample quality.
   - Improved stability of the optimization process.

   The original paper argues that the instability of usual GANs is due to its loss function, which is based on the **Jensen-Shannon (JS)** distance:

   $$D_{JS}(p_{data}||p_g) = \frac{1}{2}\,\mathbb{E}_{x\sim p_{data}}\, log\frac{p_{data}(x)}{\frac{p_{data}(x)+p_g(x)}{2}} + \frac{1}{2}\,\mathbb{E}_{x\sim p_g}\, log\frac{p_g(x)}{\frac{p_{data}(x)+p_g(x)}{2}} = D_{JS}(p_g||p_{data}). \tag{3.20}$$

   As referred in previous sections, the main objective of GANs is to try to minimise the distance between a target distribution and its own estimate that comes from transforming another source (e.g. noise) into a meaningful distribution. In practise, this can sometimes be a problem as not always there is a smooth path to minimise this JS distance. This will not allow the training to converge. Using the Wasserstein loss function will lead to have new expressions for both $\mathcal{L}^{(g)}$, and $\mathcal{L}^{(d)}$.

   $$\mathcal{L}^{(d)} = -\,\mathbb{E}_{x\sim p_{data}}\, d_w(\mathbf{x}) - \mathbb{E}_{\mathbf{z}}\, d_w(g(\mathbf{z})). \tag{3.21}$$

   $$\mathcal{L}^{(g)} = -\,\mathbb{E}_{\mathbf{z}}\, d_w(g(\mathbf{z})). \tag{3.22}$$

2. **Least-squares GAN (LSGAN)** [34]: This model proposes to use, again, a different loss - least squares loss. The authors prove that the use of sigmoid cross-entropy loss in GANs leads to poorly generated data quality.

One factor that GANs overlook is the true hyperspace distance between the samples and the true distribution. This is because, once a fake sample is already in the correct side of the decision boundary generated through learning, the gradients vanish. As always, a lack of gradients implies that the generator network will stop having incentives to keep improving its performance. Using the least squares loss function will avoid gradients to vanish if the fake sample is far from the real ones. This will cause that the generator makes a bigger and prolonged effort for properly estimating the real density distribution of the data. For this reason, here again, $\mathcal{L}^{(g)}$ and $\mathcal{L}^{(d)}$ will be redefined:

$$\mathcal{L}^{(d)} = \mathbb{E}_{x \sim p_{data}}(d(\mathbf{x}) - 1)^2 + \mathbb{E}_{\mathbf{z}}\, d(g(\mathbf{z}))^2. \tag{3.23}$$

$$\mathcal{L}^{(g)} = \mathbb{E}_{\mathbf{z}}(d(g(\mathbf{z})) - 1)^2. \tag{3.24}$$

Note that these two methods are just two out of several potential methods developed over the past years to improve ordinary GANs performance. Some other examples of potential alternatives are models and architectures such as the auxiliary classifier GAN (ACGAN), the conditional GAN (CGAN), several types of disentangled representation GANs, or even an extension of WGANs including gradient penalty loss (WGAN-GP). However, the two previous examples are the most relevant cases of improvements that could be beneficial for our purpose in this paper.

4 Models

Throughout this section we will focus more carefully in the GAN models used for the experiments. Mainly, two distinct models leveraged to different number of dimensions were used for this study. They can be separated into two clear groups, the GANs based on Multilayer Perceptrons [23] or the ones that made use of convolution layers [31]. The former networks will be the ones whose results will be compared to variational autoencoders with, as well, MLP structures. The objective of this chapter is to provide with a solid understanding of the theoretical bases of the architectures, and later on, also talk about the specific structure of the models used including specific hyperparameter setups.

4.1 Multilayer Perceptron (MLP) based GAN

4.1.1 The Perceptron

The Perceptron is the baseline for deep learning and the simplest ANN architectures. It was invented in 1957 by Frank Rosenblatt [42]. It is based on a type of artificial neuron called *threshold logic uni* (TLU), or *linear threshold unit* (LTU). The main purpose of the TLU is to compute a weighted sum of all of the inputs which are connected with a specific weight:

$$z = w_1 x_1 + w_2 x_2 + ... + w_n x_n = \mathbf{x}^T \mathbf{w}. \tag{4.1}$$

Then, it applies a step function to the previous computation:

$$h_w(x) = step(z). \tag{4.2}$$

Usually, the functions used as step are either the Heavyside step function:

$$heaviside(z) = \begin{cases} 0 \text{ if } z < 0 \\ 1 \text{ if } z \geqslant 0 \end{cases} \tag{4.3}$$

Or, the sign function:

$$sgn(z) = \begin{cases} -1 \text{ if } z < 0 \\ 0 \text{ if } z = 0 \\ +1 \text{ if } z > 0 \end{cases} \tag{4.4}$$

The structure of the Perceptron is composed of one unique layer of TLUs with each of these being connected to every input. This type of layer is called *dense layer* or *fully connected layer* because all of the neurons in the layer are connected to all of the instances of the input layer.

Using linear algebra to handle the input data as a matrix, $\mathbf{X}$, and the weights matrix, $\mathbf{W}$, it is feasible to calculate the results of an output layer (e.g. the outputs resultant from many input data points at the same time):

$$h_{\mathbf{W},\mathbf{b}}(\mathbf{X}) = \phi(\mathbf{W}\mathbf{X} + \mathbf{b}), \tag{4.5}$$

where **b** is the bias vector and ϕ is the **activation function**.

Perceptron training follows the **Perceptron learning rule** (Fig. 7), which tries to reinforce connections that help reduce the error, just like any other optimisation algorithm. For every output neuron that produces a wrong prediction, the model adjusts its weights looking for different values that would contribute towards a correct decision. The rule can be described with the following equation:

$$w'_{i,j} = w_{i,j} + \eta(y_j - \hat{y}_j)x_i, \tag{4.6}$$

where $w_{i,j}$ is the weight connecting the ith input and jth outputs, x_i is the value of the ith input neuron, $\hat{y}_j$ is the value of the jth output neuron, y_j is the target output of the jth output neuron, and η is the constant known as learning rate.

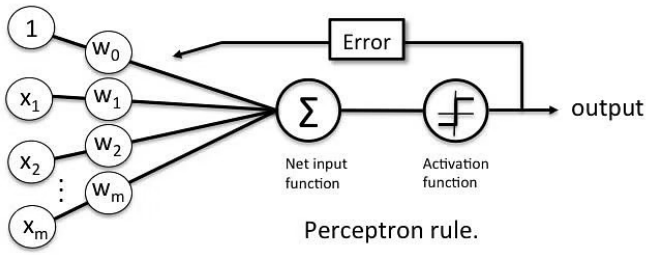


Figure 7: Perceptron diagram showing the simple structure of the model. It consists of a set of inputs, weights, an input function, an activation function, and it uses the error together with a type of backpropagation for training.

4.1.2 Multilayer Perceptron and Back-propagation

A multilayer Perceptron (MLP) is a class of feedforward artificial neural network (FNN) and an extension of the above descrived Perceptron. It is composed of one input layer, one or more layers of TLUs called hidden layers, and one final layer of TLUs known as the output layer (Fig. 8).

When a MLP contains several hidden layers it starts to be considered as a deep neural network. An important step in the literature that allowed the creation and viability of MLPs as well as most of the neural networks known nowadays, is **back-propagation** [32]. In order to understand back-propagation (backprop), let us review one of the fundamental rules used in calculus, the **chain rule** for differentiation.

Let $x \in \Re$, and let f and g be functions mapping a real number to another real number. If $y = g(x)$ and $z = f(g(x)) = f(y)$, then the chain rule is:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}. \tag{4.7}$$

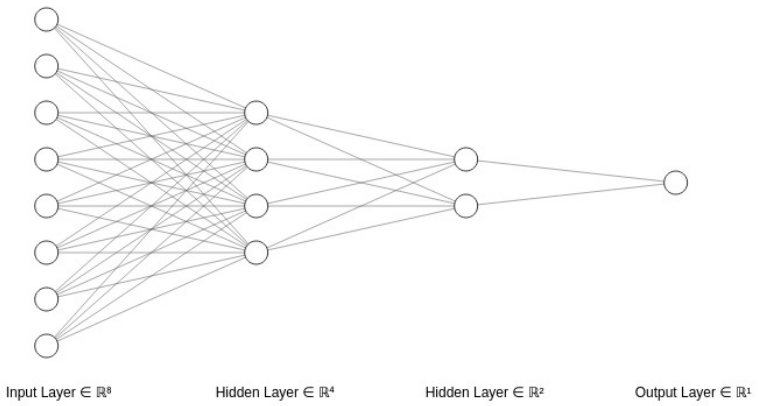


Figure 8: Multilayer perceptron (MLP) example architecture. This specific example consists of an eight neuron input layer, two hidden layers with four neurons and two neurons respectively, and a final output layer with one node.

It is possible to generalise this equation for it to work with vectors, or even tensors. Consider, $\mathbf{x} \in \Re^m$, $\mathbf{y} \in \Re^n$, $g : \Re^m \mapsto \Re^n$, and $f : \Re^n \mapsto \Re$. If $\mathbf{y} = g(\mathbf{x})$ and $z = f(\mathbf{y})$, then:

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}. \tag{4.8}$$

This, in vector notation:

$$\nabla \mathbf{x}^z = (\frac{\partial \mathbf{y}}{\partial \mathbf{x}})^T \nabla \mathbf{y}^z, \tag{4.9}$$

where $(\frac{\partial \mathbf{y}}{\partial \mathbf{x}})$ is the Jacobian matrix of g. The back-propagation algorithm consists of performing such a Jacobian-gradient product for each connection in the model. Nonetheless, backprop is usually applied to tensors in practice and, although the derivation is outside the scope of this text, it is interesting to show the resultant equation. The chain rule as it applies to tensors is:

$$\nabla \mathbf{X}^z = \sum_j (\nabla \mathbf{X} \mathbf{Y}_j) \frac{\partial z}{\partial \mathbf{Y}_j}, \tag{4.10}$$

where the gradient of a value z with respect to a tensor $\mathbf{X}$ is being expressed as $\nabla \mathbf{X}^z$, just as if $\mathbf{X}$ was a vector. The equation above was expressed, also, assuming that $\mathbf{Y} = g(\mathbf{X})$ and $z = f(\mathbf{Y})$.

We can get to understand how back-propagation works through evaluating the example of its computations in a fully connected MLP. For this purpose, it is important to evaluate and internalise the algorithms 1 and 2 of the appendix.

- Algorithm 1 represents the process of forwards propagation that aims to calculate the loss $L(\hat{\mathbf{y}}, \mathbf{y})$ for a specific set of parameters, and an input and target from the training dataset.

- Algorithm 2 demonstrates how the corresponding process of back-propagation works for the same computational graph corresponding to the MLP.

Although this example helps understanding the inner workings of the algorithm, in practise, modern implementations are build upon a more generalised version of back-propagation that ends up constituting a more optimal solution. However, for the scope of this paper, it will be enough to grasp a solid understanding of the basis of this technique without having to go far into more technical detail.

4.1.3 Glash - Our generative adversarial network

Glash is the name given to the generative adversarial network we designed to tackle the experiments carried out throughout the investigation. The first reliable model we used was based on a multilayer perceptron architecture for both the generator and the discriminator networks. The specific code used to build this model is shown in Listing 5.

The **generator network** (Fig. 9) is constructed using one input layer, two hidden layers, and one output layer. The input layer has the shape of $(5,)$ where five was a number arbitrarily chose as the number of latent dimensions of the latent space used for these experiments, this is one more hyperparameter in the model. Both hidden layers are Dense layers, and have the same format, this being the use of ten hidden neurons, ReLU as the activation function [5] (although the code allows this to change through a change of the arguments of the function), and the "he uniform" kernel initialiser. Following these two layers comes the output layer. The output layer outputs what will be the input to the discriminator network and for the case of the generator network it uses, again, ReLU as its activation function.

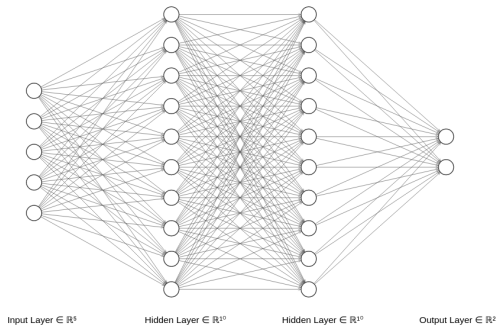


Figure 9: Generator network present within the architecture of the Glash based on a multilayer perceptron.

The **discriminator network** (Fig. 10) is similarly composed of the input layer, two hidden layers, and one output layer. The input layer has the shape $(2,)$ where the two comes from the dimensions, one point per axis of the data, in this case because the model is aimed for two-dimensional data then we have two input values. The discriminator will iterate over the whole array contained in the batches receiving each coordinate one by one. The first hidden layer is a Dense layer with fifteen hidden neurons, using the ReLU activation function [5], and the "he uniform" kernel initialiser. A similar structure is found for the second hidden

layer this time with ten hidden neurons instead. Lastly, the output layer contains one hidden neuron which will output a number n, where $0 \leq n \leq 1$, that represents the confidence of classification that the model has. If this output is closer to zero, then the discriminator believes that it is likely that it is a fake data point, otherwise, if it is close to one, the model classifies it as a real sample. This output layer uses, as per usual in binary classification, the sigmoid activation function [21].

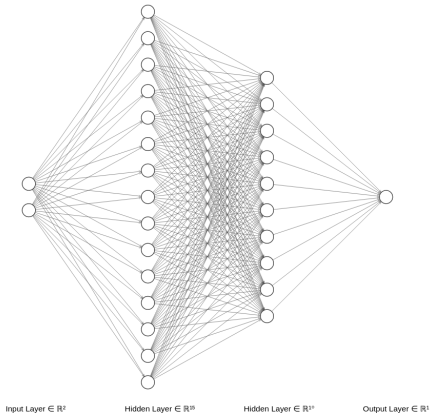


Figure 10: Discriminator network present within the architecture of the Glash based on a multilayer perceptron.

4.2 Deep Convolutional GAN (DCGAN)

4.2.1 Convolution operation

In its most general form, convolution is an operation on two functions of a real-values argument and describes how the shape of one of the functions is modified by the other. It is denoted with an asterisk, and it is defined as:

$$s(t) = \int x(a)w(t-a)da \tag{4.11}$$

$$= (x * w)(t). \tag{4.12}$$

Convolution is defined for any two functions for which the integral above is defined. The first function used as an argument is referred as the **input**, and the second function as the **kernel**. Also, the output of the convolution operation is commonly known as **feature map**.

In computation the data will be quantified and divided into regular intervals, this suggests that it could be possible to use a more realistic definition of the convolution operation for neural networks purposes. Then, index t takes values of a discrete set. If the fact that x and w are defined only on t is assumed, then we can define the new convolution:

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a). \tag{4.13}$$

Because each element of the input and kernel must be explicitly stored separately, it is assumed that these functions are zero everywhere but in the finite set of points for which we store the values [references deep learning]. This means that in practice, we can implement the infinite summation as a summation over a finite number of array elements [21].

One of the most common applications for convolution is computer vision which involves mainly images as input. These images are usually multi-dimensional, often at least, two-dimensional. For a 2D image, I; a 2D kernel, K, will be required; and the feature map, S, will also be 2D.

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n). \tag{4.14}$$

Using the commutative property of convolution:

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n). \tag{4.15}$$

This latter expression is left in this form for the sake of implementation. For machine learning applications, it is easier to apply because of the tighter variation in the range of valid values for m and n.

4.2.2 Motivation

There are several reasons why convolution is helpful for neural networks. Here, we will focus on going through, specifically, three main ideas that convolution leverages.

Default neural networks are based on having each of the output units interacting with every input unit. This is related to how neural networks work, their inner process consists in computing large matrix multiplications where one of the matrices includes parameters that reflect how the input and output units interact. This idea is changed with the implementation of convolution, it introduces **sparse interactions**. It is thanks to the new possibility of having a kernel with size smaller than the input that allows finding meaningful features such as edges by analysing a considerable reduced number of pixels, that we are able to considerably store less parameters therefore increasing efficiency while, at the same time, reducing memory requirements.

It is also simple to show how much more efficient the neural networks can become due to convolution. The complexity of the matrix multiplication algorithms is $O(m * x)$ where m is the number of inputs and n is the number of outputs. This means that, if the interactions between the units in the input and output layers are limited to k interactions, where $k << m$, then the time complexity of the computations carried by the network will considerably decrease.

Another important benefit of convolution is what is called **parameter sharing**. As the name suggest, this feature consists of using the same parameter for more than one function

in the model. In contrast to classic neural networks where each parameter is just used once to compute its respective output layer, here the weights are attached together and applied in several places. The parameter sharing is meant to allow the network to learn the parameters of a set instead of all the locations inside the kernel. Not only it does not affect the runtime, it still is $O(k * n)$, but it also reduces further the amount of parameters required to store implying that convolutional networks become dramatically more efficient.

Parameter sharing in convolution leads to the third property, this is **equivariance** to translation. Equivariance in functions means that the changes applied to the input are also applied to the ouput in a similar manner. The property can be described mathematically with the following expression, if:

$$f(g(x)) = g(f(x)), \tag{4.16}$$

then f is equivariant to g. In the context of convolution, if we let g be any function that translates the input (shifts it), then the convolution function is equivariant to g [21].

As a final note, it is relevant to mention that some kinds of data cannot be handled by the neural networks based on matrix multiplication. Convolution however, makes possible to process this types of data. This is an interesting topic, however it is outside of the scope of this document.

4.2.3 Deep Convolutional Glash

Starting from the generator, we will describe both networks that conform to the model. This time, it will not be such a simple structure as was the case for the networks based on multilayer perceptrons with few hidden layers. Now, there are extra hyperparameters in play such as strides, filter layers, and concepts like batch normalisation [25], and, as the name indicates convolutional layers. As it will be explained more in detail later on within the section of "Experiments", this network's first task was planned to be working with the MNIST handwritten digits. The network, therefore, was optimised for this purpose and this becomes clear in the choice of the configuration of the model.

The process of building the generator starts by a resize of the input images, setting the kernel size of the convolutions to five, and deciding the values for the filter layers. The first dense layer has a set number of hidden neurons that depends on the size of the images and the current filter layer the model is iterating on. The amount of hidden neurons is given by the image size (in this case 7x7 after resizing) times the value of the filter layer. For DCGlash we use four filter layers starting with one hundred and twenty-eight and halving this value every step until after the third layer, which goes from thirty-two to directly one. Just after this first layer, the reshaping layer acts to modify the shape of the input to $(7, 7, 128)$, where the last dimension is the start point of the filter layer. Here is the instance in which the iteration over the filter layers starts. The first layer of the loop applies batch normalisation to the data using the ReLU activation function. Following this, we arrive at the first convolutional layer that consists of a two-dimensional transpose layer that uses the current number of filters,

the kernel size set initially, two strides (except during the last two iterations for which the filter layer is one), and using the "same" technique for padding. As mentioned already, this will go on in a loop four times as there are four filter layers. Once the loop is over, the data goes through an activation layer that uses the sigmoid activation function as the last step.

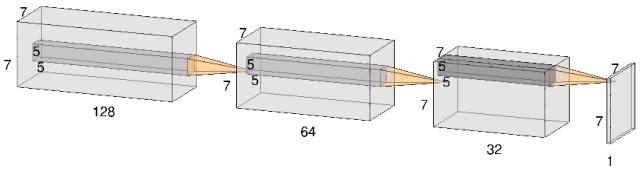


Figure 11: AlexNet style diagram that represents the generator network used in DCGlash. We can see the resized images (7, 7) the number of filters as the length of the main boxes starting from 128 and ending with 1, and the size of the convolutional filters (a constant kernel size of 5x5).

The specifications of the discriminator model are slightly different with a few key modifications that contribute towards the harmony of the model allowing convergence and training stability. We use the same kernel size of five in this case, the main difference found here is the fact that the filter layers instead of halving, increase doubling its value each iteration and starting back from the value of thirty-two and increasing all the way until two hundred and fifty-six. The loop starts with the inputs (output of the generator model or real images) that go through a LeakyReLU activation function and lead to a two-dimensional convolutional layer that uses all of the values of the filter layers (one per iteration), one stride (except during the last iteration where it uses two), and again "same" as padding technique. As the loop ends, the data gets flattened with a flatten layer, finally arriving at the output layer which is a dense layer with one hidden neuron that will output the certainty of the model on the sample being real or fake from zero to one.

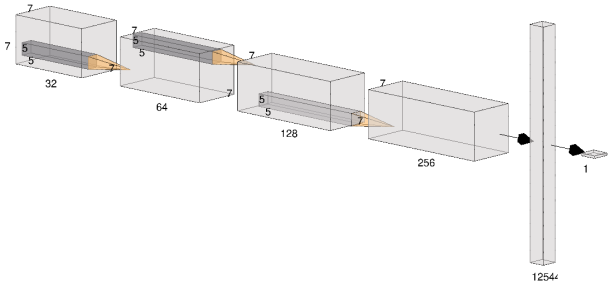


Figure 12: AlexNet style diagram that represents the discriminator network used in DCGlash. It show how the filter layers this time increase from 32 up until 256 followed up by a flattening layer and an output layer of one neuron that determines the probability of the sample being fake or real.

It is possible to find the source code of the development of the network in Listings 7 and 8.

4.3 Kuyf - Our variational autoencoder

Kuyf (Listing 10) is how we will be referring to our variational autoencoder throughout the experiments section. This network was mainly used as a second method to generate distributions and use its results for valuable comparisons with the outputs of Glash.

Kuyf is composed of two parts, the encoder and the decoder. The encoder section (Fig. 13) is made up of an input layer, a flatten layer, four dense layers, and a sampling layer. The input layer allows inputs with shape $(2, 100)$ where two is the number of dimensions of the data, and the hundred represents a hyperparameter involving the number of features. The flatten layer simply flattens the input. Following these two layers, we find the first dense layer that includes one hundred hidden neurons and uses the SeLU activation function [21]. The next dense layer halves the hidden neurons to fifty and again uses SeLU as its activation function. Now, the output of this last dense layer goes through two different dense layers that represent the *codings mean* and the *codings log of the variance* respectively. These values output from the two layers are the estimation of the mean and log of the variance made from the model and try to look for the optimal values that could represent a Gaussian distribution that fits the training data. These approximations are computed thanks to the help of the latent loss function. Lastly, the end layer is the sampling layer, which takes both the *codings mean* and the *codings log of the variance* to sample from the distribution that these two values generate.

The specific latent loss used by kuyf is:

$$\mathcal{L}_l = -\frac{1}{2}\sum_{i=1}^{n}[1 + \gamma_i - e^{\gamma_i} - \mu_i^2], \tag{4.17}$$

where γ is the log of the variance σ^2.

Getting to the decoder (Fig. 13), it is formed, again, by an input layer, three consecutive dense layers, and finally, one reshape layer. The input layer has a linear shape of (100), this depends on the coding size used previously in the encoder. This data follows through into a dense layer with fifty hidden neurons that uses SeLU as its activation layer. Next is another dense layer, this time with one hundred hidden neurons. After this, we get to our final dense layer which contains two neurons for each number of features set by the user, this hyperparameter was set to one hundred for these experiments. It is important to note that this last dense layer uses the Leaky ReLU activation function [2]. All of these layers conduct the information into one last layer that is simply a reshaping layer that leaves the output with the shape $(2, 100)$, here again, the hundred represents the number of features.

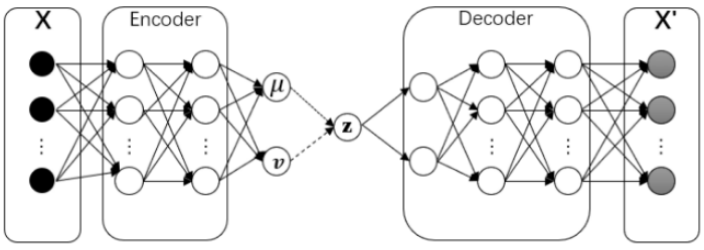


Figure 13: Structure of a variational autoencoder. Here, it is possible to see the encoder and decoder parts of the model. [40]

5 Experiments

In this section, we will portray the experiments we have used in order to tackle and understand the problems and some of the limitations of GANs and VAEs. We navigated a few approaches varying from applying transformations to the data distributions, to the use of one of the MNIST datasets (handwritten digits).

5.1 Data transformations

During the stage of literature review to understand the current situation of the research and state-of-the-art models, we realised that there was a common issue with the majority of the results. This problem was related to the general shape of the distributions to be generated. In nature as well as in the particle accelerators, showers generated from particle interactions tend to generally form Gaussian distributions. As a consequence of this, it was easy to spot how the quality of the generations in many of the academic papers reviewed was faithful to reality, however, as the data was further away from the mean of the distribution (tails) it started to lose sensitivity.

The first hypothesis studied in order to explain these poor generations of the tails of the distributions was assuming that the models being used were uncontrollably going through overfitting around the mean of the data, allegedly due to the unevenly larger amount of data around this area, and underfitting everywhere else. A potential method to investigate and could give different quality metrics of the results was applying transformations to the data distributions. The road-map to implement this was the following:

1. Start off with a real distribution, $p(x) \sim G(\mu, \sigma)$:

$$p(x) \sim \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}. \tag{5.1}$$

2. Fit this normally distributed data into a quadratic distribution by applying a logarithmic transformation to end up with a new distribution $p'(x)$:

$$p'(x) \sim c - \frac{(x-\mu)^2}{2\sigma^2}, \tag{5.2}$$

   where c is a constant that encloses the logarithmic of the normalisation constant.

3. Ideally, this would already be valid as one potential dataset to train the models on and measure performance. However, we can go further by applying an extra step with the aim of creating a linear fit and end up with our new distribution $p''(x)$.

$$p''(x) \sim \sqrt{\frac{(x-\mu)^2}{2\sigma^2}}. \tag{5.3}$$

4. Lastly, it might be interesting to evaluate an exponential distribution as well. It can be easily reached by fitting $p''(x)$ into an exponential function:

$$p'''(x) \sim e^{\sqrt{\frac{(x-\mu)^2}{2\sigma^2}}} \tag{5.4}$$

5. Train the model with each of the transformed distribution, measure performance, and take the generated distributions output of the models into a process of inverse transformation in order to achieve a normally distributed dataset that should share a common likelihood (or PDF) to the training distributions (true data).

This was achieved in practise with the code displayed in Listing 4.

Realistically after some tests, this leads to several problems. Firstly, if the initial distribution, $p(x)$, contained negative values, then the transformation series leads to an issue of multiplicity of these negative values. This means that, instead of getting target shapes such as "U" shapes or "V" shapes for the histograms, we would achieve just half of the previous shape, therefore, losing valuable information that could not be achieved from an inverse transform. This was an unacceptable loss that was a heavy reason to drop the idea.

Potential solutions were thought for this. The strongest was to use a constant value as part of the transformation with the objective of shifting the initial distribution to the positive axis. However, this approach had weaknesses in the cases for which the data is more complex than in one dimension, where correlations appear. As well, we felt like it was not a sensible idea to arbitrarily select a constant number to carry this shift as in a real use-case this process is not ensured to work. We did not find a reliable approach to do this either. Things like trying to extract inner features of the individual distributions were tested, however, this added a new level of complexity for the models to learn and it would have caused problems in the training and evaluation of the simulations. As well, the aim of this was to try to find proof and confirm our initial hypothesis, but it would have not been an effective solution in most of the cases as, even with the transformations, in each and every step, the distributions still had areas with a small number of samples. This would have led to the same performance issues and quality loss in these unrepresented areas.

5.2 Glash for "U" shape reproduction

Using the Glash based on a multilayer perceptron, we decided to carry out an experiment that would involve a dataset that models a chain of data points displaying a "U" shape (Fig. 14). This is a good starting point dataset created by fitting a randomly uniform number of points. It can also be thought of as a kind of complicated problem for which Glash will need to learn the symmetries of the data as well as its non-linearity.

For this, we built a setup for which the data mentioned above (Fig. 14) was employed as training data. It was divided into batches for later use of batch training in order to avoid memory overloads. The batches consisted of sets of 64 data points and a prefetch of 32.

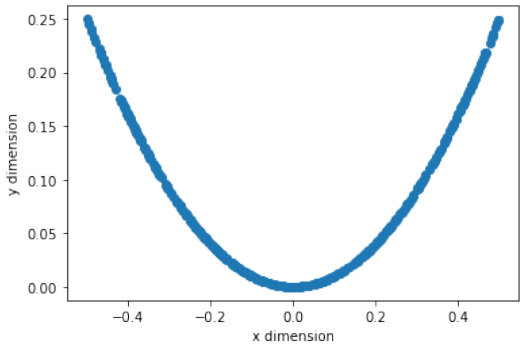


Figure 14: U-shaped data used for training of 2D-Glash. This data was generated by the function "toy data" found in Appendix C.

For this specific task, we modified the activation function in the generator network of Glash, this time *tanh* [37] was used as it was proven to output better results. The compilation of the model was using the ADAM optimiser [29] built into Keras Tensorflow [3] and using the binary cross-entropy function as the loss function (also built into Keras). In terms of training, it was done for five thousand epochs using a custom callback designed for the program to store images of the output of the generator for each epoch. This, together with our function "make_animation" (Listing 1) allowed us to track the progress of Glash throughout learning along with the loss plot.

5.3 Glash for simulation of 2D normal distributions

After eventually discarding the idea of distribution transformations (at least for the time being), we decided to use Glash to simulate two-dimensional normal distributions. A two-dimensional normal distribution is composed of data that is normally distributed in both dimensions, these two dimensions can be correlated or uncorrelated, and have the same or different means and/or standard deviations.

Because of the nature of the pseudo-random generation of Gaussian distributions using the Python module NumPy [4], the task of evaluating the results was not as simple as a direct comparison of two histograms. Although the previous is true, a direct comparison would be a good estimate of the simulation performance as the distance metrics would be fairly bad for simulations that are completely off, and decently good for simulations close to the true distributions. Note that it would rarely be a distance of zero as the task is not intended to be to learn how to **copy** the training data, also the pseudo-randomness of the generations would contribute to differences. In any case, the parameters that describe the distribution more accurately are the mean μ, the standard deviation σ, and the covariance in the case of multidimensional data. We also used these parameters in order to do comparisons between the data generated with NumPy and the simulated samples after training.

Firstly, we focused on making the model understand the concept of mean and be reactive to shifts along this axis. To do this we trained the model using normal data distributions with $\sigma^2 = 0$ and different values of μ. As an example, we will analyse the results given for

a two-dimensional Gaussian distribution with $\mu = 1$ (Fig. 15).

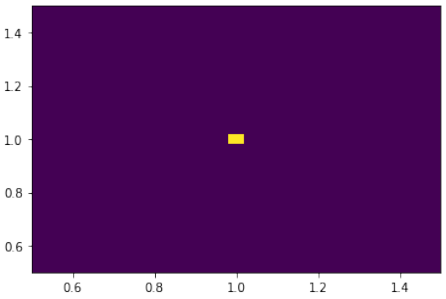


Figure 15: Gaussian distribution with mean $\mu_{x,y} = 1$ and variance $\sigma^2_{x,y} = 0$ generated using numpy random.normal and used as training data.

The version of Glash used can be seen in Listing 5. The training data for the initial steps of this experiment can be seen in Fig. 16, it was divided into batches of size 32 and prefetch 32 with the aim of using the batch training technique. It was compiled once again using the Adam optimiser for both the generator and discriminator networks and again, repeating the binary cross-entropy function for both networks. Also, early stopping [20] was used in order to avoid the model from getting an extremely divergent loss.

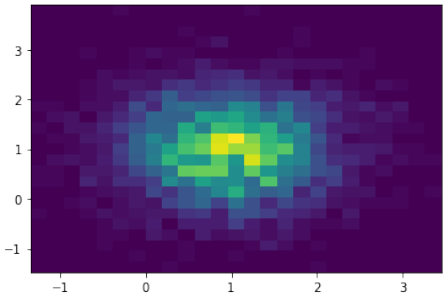


Figure 16: Example of a two-dimensional Gaussian distribution with mean $\mu_{x,y} = 1$ and variance $\sigma^2_{x,y} = 0.5$ generated using numpy random.normal that was used as training data.

For a numerical comparison between visually "close" histograms, we decided to use the chi-squared statistical test [11] (framework developed can be found in Listing 3) to measure the exact overall distance between the data points of each set. The chi-squared formula is the following:

$$\chi^2 = \sum_i^n \frac{(O_i - E_i)^2}{E_i}, \tag{5.5}$$

where i is just an index that runs through all of the elements in the dataset, n is the total number of data points, O_i are the observed values, and E_i are the expected values.

Following the previous procedures discusses above, the next step was to expand the model in order for it to be able to extrapolate something aside from merely the mean. The ideal situation would be results that can correctly represent the distribution by matching both μ and σ for each dimension. In order to do this, the same training procedure was followed as above, this time changing the values of σ (Fig. 16) and relying upon hyperparameter tuning.

5.4 DCGlash with MNIST images

Lastly, the final experiment designed for the testing of the deep convolutional variation of Glash was the simulation of handwritten digits using the famous MNIST dataset contained in Tensorflow Keras (Fig. 17). The results from this experiment could be a nice proof-of-concept of the idea that convolutional models being trained on distributions that contain underpopulated areas might be a potential improvement to the results seen previously by the MLP-Glash. The most important parts of the code developed to carry out this experiment can be inspected in Listings 7-9.

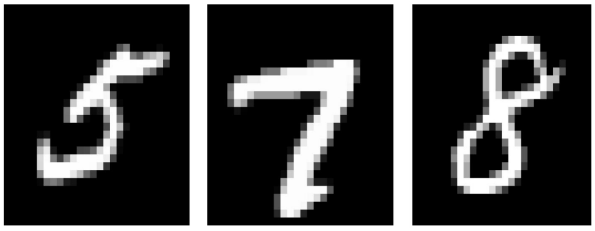


Figure 17: Three MNIST images containing the handwritten numbers five, seven, and eight.

We begin by setting up each individual network starting with the discriminator. The input shape for the discriminator will depend purely on the size of the training images, in this case, MNIST images are of size 28x28 meaning that the input layer shape, after resizing, will be $(7, 7, 1)$. This input layer is chosen to initialise the discriminator model that is compiled with a binary crossentropy loss, and a RMSprop (root mean squared propagation) optimiser [15] with a learning rate $\beta = 2{\cdot}10^{-4}$ and a decay of $\alpha = 6{\cdot}10^{-8}$.

Following up with the setup process, we initialise the generator network by using a one-dimensional input shape with a length directly proportional to the latent size of the input vector (shape $(100,)$). The optimiser function used for the generator is, again, an RMSprop with a learning rate and a decay that represent half of the values used for the discriminator model.

The training runs for fourty thousand epochs using batch training with batches of size sixty-four and following a fairly standard process similar to the one described in the section of "Learning" within section 3 of this document. We also implemented an extra step that aimed to store the output of the generator every five hundred epochs that was thought for visualising the progress of the network throughout learning.

6 Results

In this section, we will display the results obtained for each of the experiments discussed in the previous section. The necessary figures that will be required for a later discussion in the following section, will be exhibited here together with some of the technical details that, added on top of the methodologies explained above, should provide a full picture of how the experiments took place and the discoveries that are linked to them.

6.1 "U" shape reproduction

In order to get a different visualisation perspective and the data in a format in which we are able to directly compare it to the two-dimensional histograms used in other experiments. Fig. 18 is the visualisation of the data as a 2D histogram where each of the dimensions of the histogram represents one dimension of the arrays that the "toy_data" (Listing 1) function outputs.

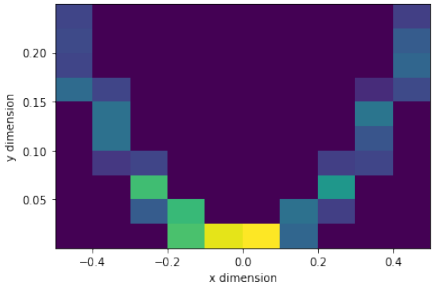


Figure 18: U-shape histogram used as training data.

From Fig. 18, it is possible to appreciate how the distribution used as training data can be related to a normal distribution in the sense that the majority of the data points (mean, μ) are located in the lowest points of the "U" (around the origin of Fig. 18) and the data scatters as we move towards the tails at which the events are scarcer (this represents the standard deviation, σ).

The training process was successful as Glash reached convergence before reaching five thousand epochs (Fig. 19). The losses of the networks slightly oscillated during this convergence process around the values of $0.68 \leq [\mathcal{L}^{(g)}, \mathcal{L}^{(d)}] \leq 0.71$ reaching exactly $\mathcal{L}^{(g)} = 0.6862$ and $\mathcal{L}^{(d)} = 0.6923$ in the last epoch.
The training ultimately translates into the results of the generations obtained by the model at the end of it. We kept track of the generations throughout training to try to understand the learning process of the model (Fig. 20). These results will be commented further in the following section.

However, due to the stochastic nature of the process, it took some runs to come up with a converging model, many times we found the situation in which the model, after seemingly reaching convergence, loses track and comes back to output poor results. The outputs achieved during training in these cases show solid progress until obtaining a good fit similar to the sub-figure (f) in Fig. 20, and after several epochs go back to results such as sub-figure

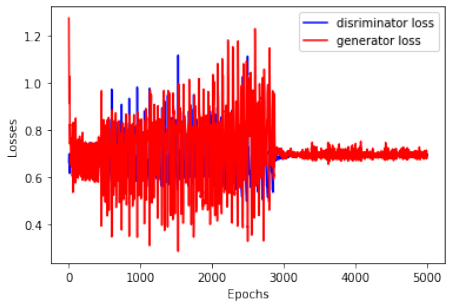
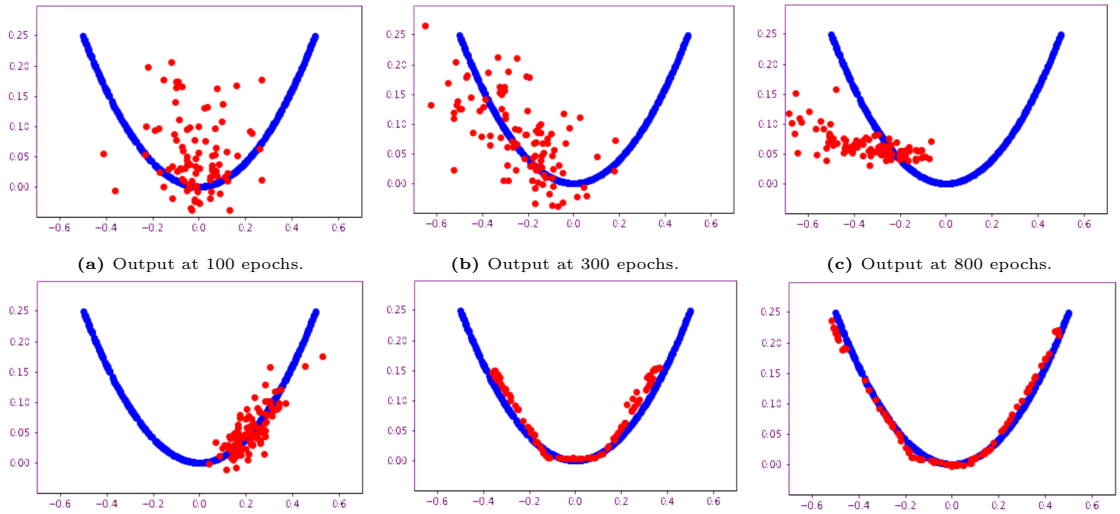


Figure 19: Displays the losses of both of the networks that form Glash, the discriminator and generator. Convergence is reached at around three thousand epochs and after a dangerously oscillating stage during learning.



(a) Output at 100 epochs. **(b)** Output at 300 epochs. **(c)** Output at 800 epochs.

(d) Output at 1500 epochs. **(e)** Output at 2500 epochs. **(f)** Output at 4000 epochs.

Figure 20: Learning progress of Glash being trained on the U-shaped data. The red dots represent the Glash-generated data points and the blue dots represent the true distribution.

(d). This could be methodically solved by applying early-stopping techniques. Not only from the compilation of outputs throughout learning is that we managed to realise this decay in performance after getting to a temporary convergence, but the plot of the losses from each of the networks also showed it clearly as well (Fig. 21).

6.2 2D normal distributions generation

Two-dimensional distributions happened to be a major challenge for our generative models. In this experiment, we were able to compare the results from both Kuyf and Glash in order to see which one is better performing. Now, the results will be shown and later on, in the next section, we will discuss our hypothesis to justify which one is better and why.

Starting with Glash, one of the first checks that were done was its capability of reacting to mean shifts. This was a minimum requirement in order to verify that the model is able to adapt to the complexity of generating global distributions. As mentioned in 5.3, to test this, we used a two-dimensional normal distribution reduced to the simplest form it can take, e.g. $\mu_{x,y} = (1, 1)$ and $\sigma^2_{x,y} = O$ (Fig. 22).

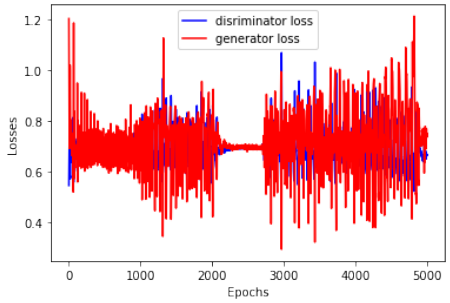
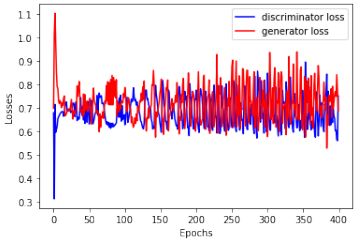
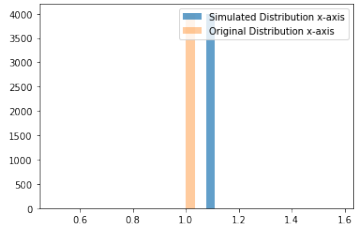


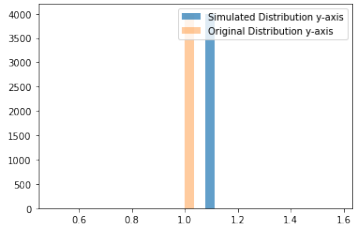
Figure 21: Plot corresponding to the losses of each of the networks that form Glash. It shows how after reaching what it seems convergence, if training is not stopped, there is the risk of going back to a poor performance.



(a) Losses of the discriminator (blue) and generator (red) networks throughout training.



(b) Simulated distribution (blue) versus original distribution (orange) in the **x** dimension



(c) Simulated distribution (blue) versus original distribution (orange) in the **y** dimension

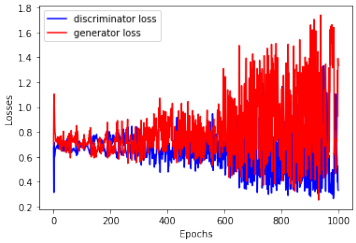
Figure 22: Results after 400 epochs including the losses for both networks during training, and the generated versus the original distributions in each dimension.

Three setups were used for this experiment each using a different number of epochs. Firstly, we run training for four hundred epochs and evaluate the results (Fig. 22). From Fig. 22, it is possible to see the results of the model in estimating the parameters of the distributions fed as training data. As mentioned above and as reflected in the table, the original distribution had a mean of $\mu = 1$ for both the x and y dimensions. As well, from the table (Fig. 23), we can see that the estimated mean by the model in the x dimension was $\mu_x = 1.077085$ having an error of $\epsilon_x \sim 7.7\%$, and similarly, in the y axis where $\mu_y = 1.077841$, this time having an error of $\epsilon_y \sim 7.8\%$. On the other hand, and as expected, the model understands that the distributions it should generate must not include standard deviation ($\sigma = 0$), this is demonstrated, again, in Fig. 23 where the table clearly shows how the standard deviation and therefore the variance and covariance are zero for both the original distribution and the generated sample.

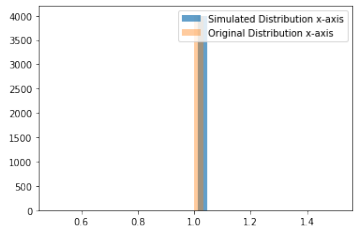
		original_x_dim	generated_x_dim	original_y_dim	generated_y_dim
0	Mean:	1.000000	1.077085	1.000000	1.077841
1	Standard Deviation:	0.000000	0.000000	0.000000	0.000000
2	Variance:	0.000000	0.000000	0.000000	0.000000
3	Covariance:	0.000000	0.000000	0.000000	0.000000

Figure 23: Table displaying the comparison of the statistics between each of the dimensions of the generated and original data distributions after 400 epochs.

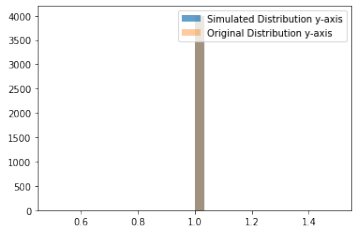
Moving forward to the next setup where the epochs were modified and set to one thousand iterations, we can see clear changes in the results. From Fig. 24, it is identifiable a non-



(a) Losses of the discriminator (blue) and generator (red) networks throughout training.



(b) Simulated distribution (blue) versus original distribution (orange) in the **x** dimension



(c) Simulated distribution (blue) versus original distribution (orange) in the **y** dimension

Figure 24: Results after 1000 epochs including the losses for both networks during training, and the generated versus the original distributions in each dimension.

convergent plot for the losses (a), and more accurate simulations in each of the dimensions (b) and (c). We are able to say more accurate simply basing ourselves on the results displayed in Fig. 25, where, using the same error computation as before, we get an uncertainty of $\epsilon_x \sim 1.1\%$ and $\epsilon_y \sim 0.9\%$ in the x and y dimensions respectively. Here again, the standard deviation and therefore the variance and covariance inferred values are zero just like the original training dataset.

		original_x_dim	generated_x_dim	original_y_dim	generated_y_dim
0	Mean:	1.000000	1.011277	1.000000	1.000908
1	Standard Deviation:	0.000000	0.000000	0.000000	0.000000
2	Variance:	0.000000	0.000000	0.000000	0.000000
3	Covariance:	0.000000	0.000000	0.000000	0.000000

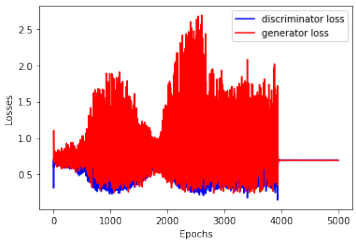
Figure 25: Table displaying the comparison of the statistics between each of the dimensions of the generated and original data distributions after 1000 epochs.

The last setup was increasing the epochs in order to test if these extreme oscillations seen in the losses during training would eventually fade away. The epochs, in this case, were extended to five thousand and the results are displayed in Fig. 27, and in the same format as a table in Fig. 26. For these outcomes, the errors in the standard deviation remained zero (e.g. the model perfectly predicts that $\sigma_{x,y} = 0$), and there is just a negligible variation in the errors for the means in each of the dimensions this time being $\epsilon_x \sim 1.2\%$ and $\epsilon \sim 0.3\%$. However, there is a potentially significant difference that can be spotted in plot (a) and (b) of Fig. 27, in which the losses seem to converge suddenly and pretty quickly after four thousand epochs.

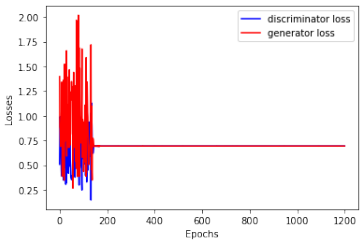
		original_x_dim	generated_x_dim	original_y_dim	generated_y_dim
0	**Mean:**	1.000000	1.011615	1.000000	0.997342
1	**Standard Deviation:**	0.000000	0.000000	0.000000	0.000000
2	**Variance:**	0.000000	0.000000	0.000000	0.000000
3	**Covariance:**	0.000000	0.000000	0.000000	0.000000

Figure 26: Table displaying the comparison of the statistics between each of the dimensions of the generated and original data distributions after 5000 epochs.

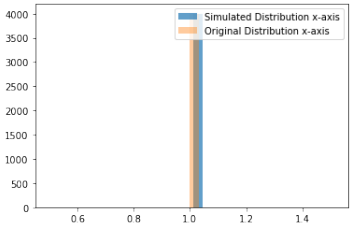
As well, just as a final check, we hoped the model to be able to adapt to μ changes and be able to predict the new means to a comparable degree of error. This was confirmed to work



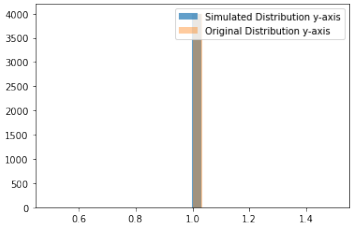
(a) Losses of the discriminator (blue) and generator (red) networks throughout training.



(b) Losses of the discriminator (blue) and generator (red) networks throughout training (zoom of the last portion).



(c) Simulated distribution (blue) versus original distribution (orange) in the **x** dimension



(d) Simulated distribution (blue) versus original distribution (orange) in the **y** dimension

Figure 27: Results after 5000 epochs including the losses for both networks during training, and the generated versus the original distributions in each dimension.

as expected although showing slightly higher errors (Fig. 28), more specifically, the values for the uncertainties are $\epsilon_x \sim 9.0\%$ and $\epsilon_y \sim 7.6\%$. As an example, we used a model that aimed to predict a simple distribution of mean $\mu = 3$ instead.

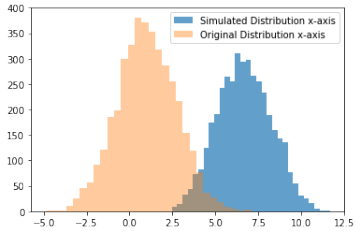
		original_x_dim	generated_x_dim	original_y_dim	generated_y_dim
0	Mean:	3.000000	3.271193	3.000000	3.234497
1	Standard Deviation:	0.000000	0.000000	0.000000	0.000000
2	Variance:	0.000000	0.000000	0.000000	0.000000
3	Covariance:	0.000000	0.000000	0.000000	0.000000

Figure 28: Table displaying the comparison of the statistics between each of the dimensions of the generated and original data distributions having shifted the mean μ to be equal to three.

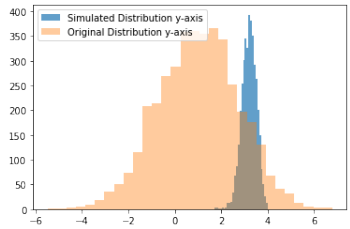
Once the previous steps were checked, it was an adequate time to proceed with the experimentation of more complex distributions and see how Glash and Kuyf react to the implementation of non-zero standard deviations in the training data. This means that from now on, we will be able to compare results from each of the models, and later on, evaluate the advantages and disadvantages of each while at the same time trying to think of potential causes of the problems.

Fig. 16 shows a two-dimensional Gaussian distribution that is namely composed of two single normal distributions in each of the axes. In the case of that figure, the standard deviation is not zero and it is possible to see the difference by a naked eye comparison between figures Fig. 16 and Fig. 15, the spread is seen in Fig. 16 is due to the standard deviation.

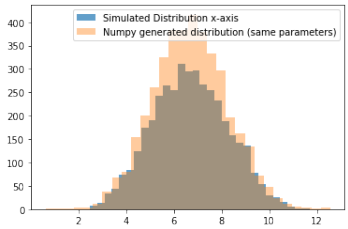
At this stage, the target histograms have therefore stopped being simple histograms contain-



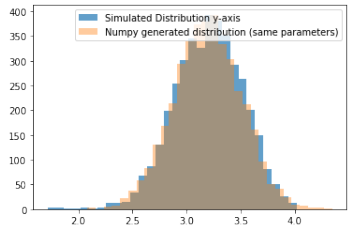
(a) Simulated x distribution (blue) in contrast to the original training data x distribution (orange).



(b) Simulated y distribution (blue) in contrast to the original training data y distribution (orange).



(c) Simulated x distribution (blue) compared to a numpy generated distribution with the same mean and standard deviation (orange).



(d) Simulated y distribution (blue) compared to a numpy generated distribution with the same mean and standard deviation (orange).

Figure 29: Results of the models after using more complex normal distributions as training data. More specifically, its parameters are: $\mu_{x,y} = 1$ and $\sigma^2_{x,y} = 3$.

Covariance, $\sigma^2_{x,y}$	χ^2 x-dimension	χ^2 y-dimension
0.5	926.3	415.2
3	288.8	546.7

Table 1: Layout of the results of the average measured chi-squared (across 10 measurements) between the generated samples and numpy generated normal distributions with the same parameters (μ, σ). These measurements were recorded as the average of ten different generated samples with the same parameters.

ing one bin that represents the mean and have become complex multidimensional normal distributions. In order to evaluate the results, we will separate each of the two dimensions and display the correspondent generated samples for each of these axes (Fig. 29). The results came together with a particularly worrying plot of the losses during training (Fig. 30 & 31) that will be analysed further in the next section. As well, the same tests were done using different values for σ^2, we tried to investigate if the model would show a better performance for any particular range of values for σ^2, an example can be seen in Figs. 32, 33, and 34.

As we will explain further in the "Discussion" section, it is remarkable for both setups that the generated samples are relatively far off from the original distribution for both dimensions. However, it is more encouraging to observe that the generated distributions offer a better chance to make comparisons with normal distributions with the same parameters (μ, σ^2) generated with NumPy. The numerical comparison done with the chi-squared method gave the results displayed in Table 1.

Lastly, the only results missing are the ones generated by Kuyf (Fig. 35). For these, we used a smaller number of different training setups as our purpose was broadly to evaluate

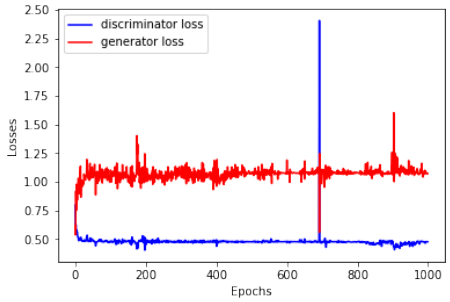
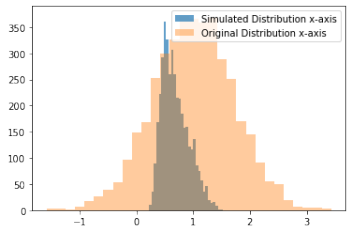


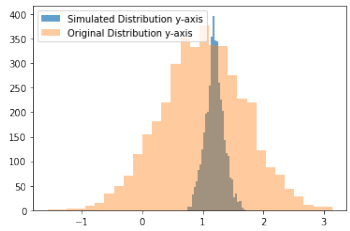
Figure 30: Standard plot displaying the losses of the discriminator (blue) and generator (red) networks throughout training.

		original_x_dim	generated_x_dim	original_y_dim	generated_y_dim
0	Mean:	1.002813	6.690219	1.017843	3.198754
1	Standard Deviation:	1.716526	1.566397	1.737220	0.317440
2	Variance:	2.946462	2.453598	3.017933	0.100768
3	Covariance:	2.947199	2.454212	3.018687	0.100793

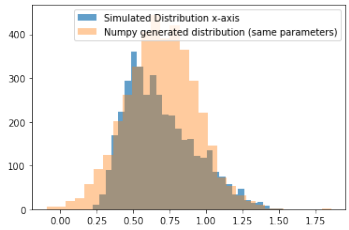
Figure 31: Table displaying the comparison of the statistics between each of the dimensions of the generated and original data distributions.



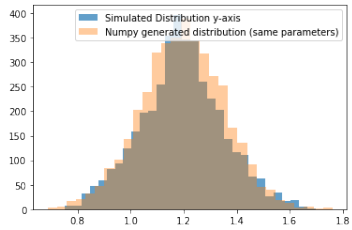
(a) Simulated x distribution (blue) in contrast to the original training data x distribution (orange).



(b) Simulated y distribution (blue) in contrast to the original training data y distribution (orange).



(c) Simulated x distribution (blue) compared to a numpy generated distribution with the same mean and standard deviation (orange).



(d) Simulated y distribution (blue) compared to a numpy generated distribution with the same mean and standard deviation (orange).

Figure 32: Results of the models after using more complex normal distributions as training data. More specifically, its parameters are: $\mu_{x,y} = 1$ and $\sigma^2_{x,y} = 0.5$.

its capability of generating significantly better samples than the GAN-based models. This means that we tested the necessary scenarios to check if there was a clear improvement in performance by Kuyf due to its probabilistic variational nature.

6.3 MNIST Handwritten digits with DCGlash

This experiment was simpler to evaluate and understand the results from the models. As mentioned in the "Experiments" section, results from the generator network were recorded

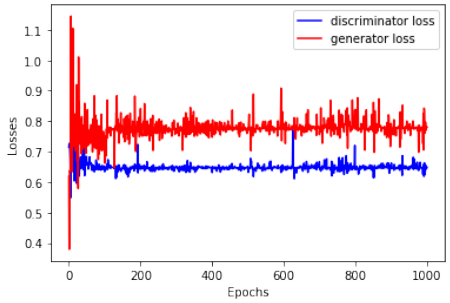


Figure 33: Losses of the networks during training for one thousand epochs with training data of covariance $\sigma^2_{x,y} = 0.5$.

		original_x_dim	generated_x_dim	original_y_dim	generated_y_dim
0	**Mean:**	1.003498	0.687828	1.003864	1.194201
1	**Standard Deviation:**	0.705707	0.236367	0.690033	0.155856
2	**Variance:**	0.498023	0.055869	0.476146	0.024291
3	**Covariance:**	0.498147	0.055883	0.476265	0.024297

Figure 34: Table of statistics comparing the simulated and real distributions in each of the dimensions. These specific results are for the setup with training data of covariance $\sigma^2_{x,y} = 0.5$.

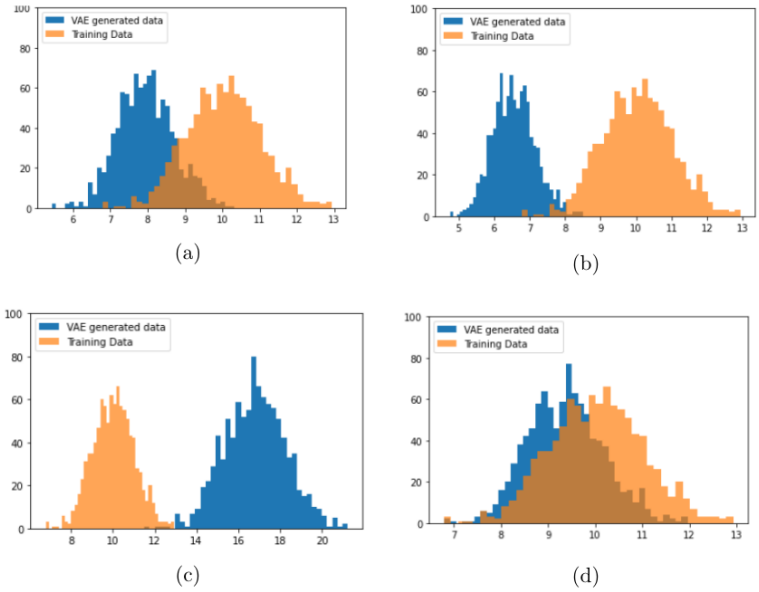
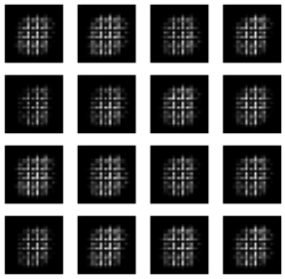


Figure 35: Some examples of how the VAE (Kuyf) generates Gaussian distributions after being train on, as well, Gaussian distributions

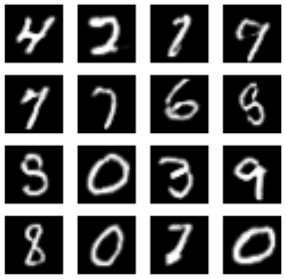
every five hundred epochs showing a clear indication of improvement and generations that could easily be confused with original samples (Fig. 36).

The training discriminator loss at the end of the training remained at $\mathcal{L}^{(d)} = 0.571940$ and accuracy of 0.734375. For this specific case, we decided to measure the adversarial loss of the model. The adversarial loss measures the difference between the ground truth and the generated data predicted by the generative model. On the final epoch, this loss was determined to be $\mathcal{L}_{adv} = 1.114590$.

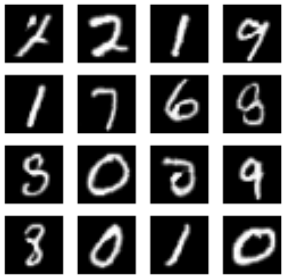


(a) Sixteen randomly selected simulated images at 500 epochs.

(b) Sixteen randomly selected simulated images at 7500 epochs.



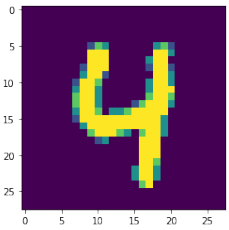
(c) Sixteen randomly selected simulated images at 15000 epochs.



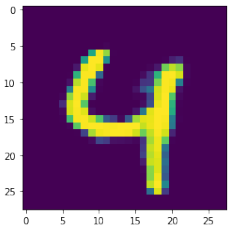
(d) Sixteen randomly selected simulated images at 40000 epochs.

Figure 36: Gathering of samples generated by DCGlash in different steps of the training.

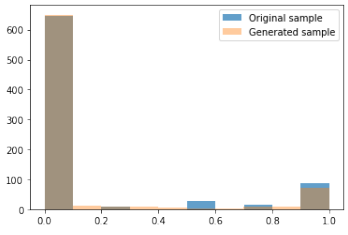
In order to get a closer look at a more specific case, we decided to carry out a head-to-head comparison of a generated sample and a real sample from the validation dataset both representing the handwritten digit four. We subjected the samples to the same comparison framework like the one used to calculate the chi-squared in the previous section (see Table 1) finally obtaining a representative value of this coefficient that was consistent with the already similarity observed in the images (a) and (b) in Fig. 37. The chi-squared result was $\chi^2 = 135.2$.



(a) True sample.



(b) Generated sample.



(c) Comparison of the histograms of (a) and (b).

Figure 37: Head-to-head comparison of an original sample and a generated sample representing the same number.

7 Discussion

From this point onward, we will stop simply displaying the results and we will aim to initiate an in-depth discussion on the meaning and significance of our findings. The purpose of this section will be to follow a logical path to the correct interpretation of the results that can be consulted throughout section 6. Here again, a one subsection will be used for each of the experiments.

7.1 Discussion of the U-shaped results

Fig. 20 is a good representation of the learning process that Glash underwent throughout training. The last sample of the results (f) in the same figure leaves a good impression of how our simplest model was able to learn a fairly good approximation of the true distribution underlying the training data. It is relevant to note that this training distribution was, as seen from Fig. 18, normally distributed in the sense that it had a clear location where most points accumulated and the number of data points decreased as you moved further from the mean, e.g. establishing a quantity like the standard deviation. This was an important step towards a proof-of-concept of a model that might be able to spot interesting distributions of different datasets that could be used as training data.

Some remarks that are worth talking about are things like the misleading measures of error of the networks at the end of the inference process. As mentioned above, these errors ($0.68 \leq [\mathcal{L}^{(g)}, \mathcal{L}^{(d)}] \leq 0.71$) were contained within a close interval due to the convergence seen in Figure 20. This convergence is a healthy indicator that suggests that training reached an appropriate stage where the weights and biases of both networks cannot be improved by any type of backpropagation. Theoretically, the ideal convergence should happen at $[\mathcal{L}^{(g)}, \mathcal{L}^{(d)}] = 0.5$, as this would indicate that the discriminator network would be guessing instead of classifying using solid criteria. However, in practise, situations like these are uncommonly seen. And, although our results do not match the theoretical perfect convergence, from the images recorded throughout training, it is possible to see how the model adequately learns while exploring several options and ends up finding a good fit of new data to the function that maps uniformly (or Gaussian) distributed to the shape seen in Fig. 14.

There is not a real consensus on what a "good" loss is, this is hard to define and it depends on the nature of the problem. For instance, it would not be fair to strictly compare the loss from a network trying to solve a regression problem with the loss of a different network carrying a binary classification task. In that case, regression is more likely to have a larger error as it is harder to predict an exact number from the spectrum of the real numbers than a class out of two possibilities. It is also true that the loss functions used for each of these problems will be different in most cases, this makes it even harder to evaluate. Something similar goes on in generative adversarial setups where the generator network is trained for regression and the discriminator for binary classification.

In any case, this experiment was designed as a first step to make before starting to explore formal normal distribution. This means that we were interested in seeing how GANs would

behave in environments where the data is in two different dimensions that are correlated to each other and where the data points are distributed in a Gaussian way, as well as to see how it reacts to the complexity that a "U" shape distribution might impose. We consider the results obtained as a solid base that demonstrates that GANs should be capable of solving, at least partially, the problem that we are ultimately trying to tackle.

7.2 Discussion of the 2D generated normal distributions

From the results displayed in section 6.2, it is possible to highlight a set of positive and negative findings. The positive results were the relative accuracy that our model has to infer the mean μ from the training datasets to a relatively low degree of uncertainty, at least when the standard deviation is small. As well, a remarkable success was seen in Figs. 28 and 31. This consists of the, generally, good understanding that the models have when it comes to identifying that the training datasets are Gaussian distributed datasets. Although the model might not estimate the parameters to an impressive level of accuracy, it was surprising to visualise the comparisons between the simulation samples and NumPy-generated normal distributions with the same mean and variance. This brings a positive view of the possibility that the models might be able to offer good performance with Gaussian distributed training data. To some degree, it contradicts our initial hypothesis of believing that the poor performing models of recent literature (at least towards the tails of the distributions) are due to overfitting and underfitting around specific areas. We also got the reassurance of confirming that the speed of simulations is unprecedented and incomparable to the commonly used Monte Carlo methods, meaning that, if better performance overall is achieved, the generative machine learning approach would be a fast enough solution that would meet all of the time necessities of future particle physics experiments.

Unfortunately, we also found some serious problems with the results. A common issue that was encountered was the faulty approximation of the standard deviation (or covariance for multiple dimensions) of the training distributions. More specifically, both Kuyf and Glash repeatedly throughout the experiments tended to **underestimate** of this quantity. In contrast to what has been discussed above, we believe this might back up our initial hypothesis. In fact, from Fig. 31, we saw that as the mean for both dimensions increased, the model tended to underestimate the standard deviation even more. A potential solution to this would be normalising the data and re-scaling it as required after training. Underestimating the standard deviation/variance and, at the same time, correctly localising the mean means that the simulated data will be sensible around the mean but will start losing significance as you get further from the peak of the distribution. In any case and as we will discuss in the section "Conclusion and future work", we are open to identifying issues with our specific models and we will even propose further steps that could significantly improve the quality of the simulations.

Furthermore, a hyperparameter that seemed to be significant was the number of epochs. By comparing the plots displaying the loss of each of the networks for each training setup (contrasted simply by the number of epochs), we could see how relevant the choice was. Convergence is an important sign of success during training and the results will, in most

cases, not be of good quality if the training plot was divergent or even too unstable. In addition to this, we will also note that the models showed a high response to slight changes in the hyperparameters or structure themselves making the training process and fine-tuning a tedious task.

There were two different approaches to this problem. During the early stages of development, we tried to train the generative models with relatively large datasets containing many individual histograms. However, we saw a lack of consistency in the results demonstrating that the models were incapable of not only not determining an accurate variance, but also that the errors for the estimated means were larger than the ones displayed previously in this report. Instead, we decided to use a single sample of a normal distribution generated with NumPy with the desired target parameters. Because this was an alternative approach we tried just at the beginning of the experimentation, we did not have the time to test it again once we had progressed in the development of the predictive algorithms. Therefore, it would not be appropriate to discard the solution and we will mention it again as part of the future work to be done.

Additionally, some comments on the comparison chi-squared metric we used. From the beginning, it is relevant to mention that the quantity is not absolute and, in fact, it is linked to considerable uncertainty that is directly proportional to the randomness of NumPy and the sampling of the models. This is because there is virtually an infinite number of ways of generating samples that have the same means and variances. Therefore the comparison was not as simple as blindly considering this chi-squared metric, the evaluation process also consisted of knowing to interpret the results by also backing the conclusion on a visual comparison. We were not able to find resources or academic papers of people doing these types of comparisons so there was not a set of predetermined values that indicate whether a simulation was accurate or not. However, we were able to clearly see the correlation between the comparison metric when applied to two histograms visually close together and two that were not.

Lastly, comparing the Glash and Kuyf models was an important task in order to benchmark the results. From the relevant figures (Figs. 28, 31, and 34) we can note some important differences. Firstly, Kuyf can be seen to be worse in estimating the means of the true distributions, however, although there is a factor of underestimating the variance, its error in the variance-covariance estimation was not as significant as the one displayed by Glash. Both algorithms were capable to identify from the start that the training data they were being trained on were simple normal distributions in one or two dimensions, so individually, the samples were good examples of Gaussian distributions as one is defined. Overall there is not a clear winner and there can be a large amount of improvements to both models as both look prominent in the generative machine learning field showing promising results.

7.3 Discussion of the MNIST images generated by DCGlash

This experiment played a similar role as the experiment discussed in 7.1. It was the opening door to realising that deep convolutional techniques applied with GANs might be a potential

(and even essential) piece of work required to achieve the hoped performance for simulations in high energy physics.

In this experiment, as expected, we were able to see a consistent increase in the quality of the generations with respect to epochs. In fact, something remarkable that was also present in the experiment of section 5.2 is that the model kept learning important patterns in the data even after reaching convergence in terms of the losses during training. We trained our deep convolutional GAN for forty thousand epochs and kept seeing improved results recorded every five thousand epochs. Investigating into this further might lead to slight improvements for both experiments, however, due to the nature of these, the positive results achieved were already enough to prove the potential of the networks. In the end, the results achieved by the network are visually good enough to indicate that the deep convolutional model should become the new standard for the experiment discussed in section 7.2.

In addition, we tried to show the similarity by using the same framework of comparison using the chi-squared metric that we used for experiment 5.3 in one of the generated samples versus an original one representing the same digit. The lowest results (e.g. the closest) were achieved out of every other experiment. Only conceptually, one can imagine why the fact that DCGlash was able to generate realistic samples of handwritten digits is an important sign that indicates that convolution is a better choice. Just from visually comparing Figs. 16 (or even 15) and 17, it is possible to see how much more complex the MNIST images are, at least from a human perspective. Now, it is a different question to determine whether the underlying distributions represented in each of the digits of the MNIST dataset are actually harder to infer than the normal distributions represented by, for example, Fig. 17. We will discuss this further in section 9.

8 Limitations

Firstly, we will look into the drawbacks of the models from a more general perspective, looking as well into problems found in the current state-of-the-art results that indicate a frequent trend. Note that this is a valuable overview that helps understand the limitations of the models described in section 4 that run without any game-changing modifications. The idea is to provide a full picture of how far we currently are from optimal results and suggest a set of rational next steps.

8.1 State-of-the-art VAEs

One of the main limitations they have lies in the quality of the images generated, they tend to come up blurry. The causes of this phenomenon are not yet known. It is suspected that this blurriness level is just an intrinsic effect of maximum likelihood, which minimises $D_{KL}(p_{data}|p_{model})$. In short, this potentially indicates that the model assigns high probabilities to the points that occur in the training dataset, but not exclusively to those. Those other points include blurry images. Part of the reason that the model would choose to put probability mass on blurry images rather than some other part of the space is that the variational autoencoders used in practise, usually have a Gaussian distribution for $p_{model}(\mathbf{x}; g(z))$. Maximising a lower bound on the likelihood of such a distribution is similar to training a traditional autoencoder with MSE (mean squared error), in the sense that it tends to ignore features of the input that occupy few pixels or that cause only a small change in the brightness of the pixels that they occupy [21]. This is an issue shared with other generative models that try to optimise a log-likelihood. The blurriness phenomenon can be observed in terms of the mean variance from the images. The mean-variance of the pixels in the generated images is considerably lower than that for data used during training [8].

Several attempts to improve this issue based on reconstruction metrics have been considered. For example, the structural similarity is a procedure that is known to be effective whenever blurriness is found, however, it still was not useful for the case of VAEs [17]. Options using deep hidden features extracted from pre-trained classification models have been evaluated before, actually providing improved results [24]. As well, a variation of traditional VAEs was tried and successfully showed better results. This new model architecture, VAEPP (variational autoencoders with a pullback prior) [14], uses the discriminator to assess the quality of the data generated and that acts on the density of the prior.

8.2 Our VAE

As mentioned in the discussion section, the VAE model that we produced showed itself to be underestimating the standard deviation of the training data for all of the experiments carried with it. In some cases, it was seen that it confused the predicted mean, however, if the case is that it had a consistent systematic error estimating the mean, then a simple solution to this would consist of manually shifting the mean to the desired value. In this case, this might propose a problem as we are uncertain of the systematic behaviour of this error when estimating means and therefore we would not be able to define an algorithmic approach to correcting the peak of the distributions for every case. The more complicated

question, nevertheless, arises if one seeks a definitive answer to determine how important the errors found in the standard deviation estimations are.

Difficulties were found at the hyperparameter tuning stage concluding with the hypothesis that the optimal training setup was reached by decreasing the variance of the sampling distribution produced. This decrease made the VAE produce more spread out distributions (larger standard deviations) approximating slightly better the true training distributions. Although this improvement was found, it still remains questioned why this happened and, in the future, it could become a problem of AI explainability might be an obstacle towards a further development or scalability of the model.

The last point that we consider important to mention is the limitation of Kuyf to learn curved distributions. This was an experiment that was not discussed in this document but it was studied in the "On the use of Generative Machine Learning Methods for particle collision simulations" dissertation by Mr Martin Fanev. We will refer to that document to warn that, from the results obtained after using reasonably curved data (such as Gaussian data mapped into a ringed distribution in two dimensions) Kuyf performed poorly (Fig. 38). This fact might be helpful to evaluate for future work and it could be a problem in some particle physics applications where cylindrical cross-sections are involved.

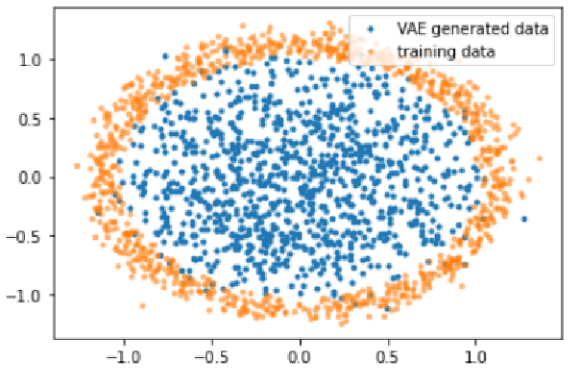


Figure 38: Ringed data experiment with Kuyf where the model seems incapable to infere a function to map noise into the ring shape.

8.3 State-of-the-art GANs

In contrast to section 3.2.2, here we will be reviewing the problems associated to recent state-of-the-art results instead of checking the overall general limitations of the model structure.

Fundamentally, GANs are not guaranteed to learn the **true distribution** of the data we are looking after. More precisely, it will try to approximate the distribution of the purely training set, which might not agree with the true underlying distribution. Bringing this issue closer to our problem means understanding that a GAN will not learn to mimic the true event generator, it will mimic the dataset it was trained on. This implies that, for our use

case where we have troubles with data shortage, we should not be tempted to expand our training dataset with more GAN-generated data, real performance will not improve.

It can be deduced that, if the systematic uncertainties do not decrease in proportion to the quantity of the sample size for training, a GAN cannot either be used to improve the statistical error of the true dataset it was trained on. Fig. 39 will help with the understanding of this:

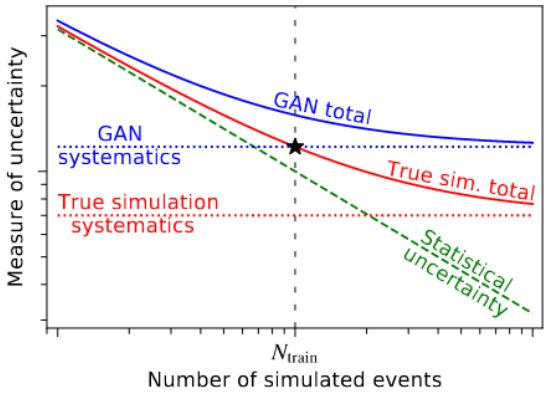


Figure 39: The green dashed line corresponds to the statistical uncertainty from simulation, while the red dotted and blue dotted lines represent the systematics uncertainty of the true simulation (MCM generated) and the GAN simulation respectively. The red solid line represents the total uncertainty of the true simulation. The solid blue is the same but for the GAN simulation. N_{train} is the number of samples the GAN was trained on. Finally, the star symbol marks the uncertainty at the moment where the GAN stopped being trained on the true simulation data and began to converge. [35]

Lastly, we see it relevant to mention that, again, there is an important fact to be aware of. This is that learning in GANs can be difficult in practice when the generator g and discriminator d are neural networks, and $max_d v(g, d)$ is not convex. This is a problem that can lead to underfitting independently of the complexity of the model. It is a hard issue to spot and it makes irrelevant the use of better performing models.

8.4 Our GAN

Here, we will focus on our MLP-GAN as this is the one we used for the most significant experiments and, hence, the one we had the opportunity to explore its limitations. In this case, most of its limitations will be linked to the limitations of the multilayer perceptrons (MLPs) themselves.

Although MLPs are able to learn non-linear models, this is not enough to get the results and the performance we seek. An important characteristic of the MLP model that made hard the experimentation is the fact that the hidden layers within the model have a non-convex loss function, and, in the hyperspace to minimise, there exist several local minimum points. In other words, due to the random nature of the weights initialisation, this can lead to different accuracy values in validation. This is a problem that implies a factor of trial and error to hope for a good weights initialisation that, in some cases, might not even be representative of what the model is actually capable of achieving.

Another limitation of the individual models inside Glash is that the networks are highly sensitive to feature scaling. This makes necessary a process of data pre-processing that would need to include scaling and normalising and could affect in questionable ways the quality of the simulated samples. Furthermore, a bigger problem than we have investigated was the limitations our model suffered from the variation of the total number of parameters in training. While it was able to output good approximate results for simpler cases (less amount of parameters), it was evident how underperforming the model became for situations where the training data had more parameters. This issue prevented Glash from being truly scalable.

One last point to note is that the evaluation of the objective advantages of the MLPs, in general, is not sufficiently helpful for our purpose. This could mean that there is not enough objective evidence that indicates that we should use MLP based GANs over any other types. In general, these models work well with large input data, provide quick predictions once training is over, and can achieve the same accuracy ratio with smaller data. However, these advantages are helping issues that our training setups do not have. They are not the only alternative if the aim is to predict with low run times or work with large amounts of data, most generative (machine learning) models are able to do this. Ultimately, big high energy physics laboratories do not have a problem of scarcity of data so in most cases, it will not be necessary to sacrifice any amount of accuracy to save on training resources.

9 Conclusion and future work

9.1 Conclusions of the experiments

The experiments that have been studied throughout this investigation were useful as an introduction to the use of two of the most promising generative machine learning methods, e.g. GANs and VAEs, in particle physics. Due to the open-ended nature of the challenge and the lack of time that precluded the inclusion of more experiments involving actual real particle collisions datasets, we did not achieve a convincing answer to whether these methods are totally suitable for things such as collision generations. However, it was possible to get several insights into the inner workings of basic and some more convoluted architectures of the models, a fact which will surely be useful in future implementations of such. In this section, we pretend to summarise our findings and try to lay a response on how performing these models got to be with simple problems and generations of two-dimensional Gaussian distributions.

Overall, within the time frame of our study, we were able to experiment with mainly MLP-based models which were considered accurate when solving mapping tasks with Gaussian distributed data. The example of this that we analysed was the task given to Glash of learning to map randomly generated noise into Gaussian data in two dimensions that were distributed as a "U" type of shape. After a large number of iterations, we saw in Figure 20 how the model was able to infer a correct estimated latent function that maps this data into the desired shape. As well, towards the end of the project, we found the time to develop a model that made use of the famous convolution technique to shift the paradigm of the project and start considering the two-dimensional distributions as images. We backed up previous experiments by demonstrating that it is indeed possible to generate high-quality "complex" (or at least more complex than the images representing the 2D normal distributions [Fig. 16]) images, in this case, hand-written digits, with the use of deep convolutional GANs. This was a key step that establishes a clear path to follow for further work.

It is proudly that we say that we were able to successfully carry out these experiments and avoid principal issues related to GANs training such as mode collapse or extreme oscillations of the losses during training (refer to section 3.2.2). Additionally, as mentioned in the early sections of this paper, we would focus on the experimentation and results of Glash (the family of models built with GANs), however, there were more positive results that are discussed in Mr Martin Fanev's dissertation - *On the use of Generative Machine Learning Methods for particle collision simulations* - with the further use of Kuyf and a few variations. In any case, the conclusions are similar to the ones we will discuss next.

Probably the most relevant experiment was the one that consisted of simulating two-dimensional Gaussian distributions with the MLP-Glash and Kuyf. This is the closest we looked into to what it would be a real experiment simulating true particle collisions data. In nature, the data provided by the detectors in the particle accelerators after particles collide, involve data distributions that are normally distributed. They have a defined mean and variance, and therefore, a sensible previous step to moving forwards to working with these datasets

would be to experiment with general and unrepresentative Gaussian distributed (in this case generated with numpy). The results were inconclusive although they made clear some faults in our models. Neither of the models tested for this experiment seemed to be able to approximate accurate variances for any of the dimensions involved in the datasets. In terms of the mean of the simulated distributions, it was close to the one displayed by the training dataset and never off by an error larger than ten per cent being independent of any type of mean shift that one could apply to the original data before training. To conclude discussing this experiment on a positive note, a good sign of these results was that the simulations represented actual normal distributions with the correct shape and some parameters for the mean μ and the variance σ^2. This suggests that the models, although they seemed to be incapable of inferring the appropriate variances, they were able to learn the overall shape of the target distributions.

9.2 Future work

For open-ended problems, it is important to contribute not only with results but it is as important to propose ideas that might push the boundaries of the experiments that one has done. For this reason, we composed a list of elements that should be worth investigating.

- First and foremost, changing the training set up for the experiment as it might be possible to obtain better performance by using a large dataset containing many individual data points where each of these points would represent an entire histogram. This is the standard in many other experiments of, for instance, computer vision for image classification or even for training a GAN to learn to simulate a specific class of image.
- Following the optimistic results of the experiment in section 5.4, DCGlash demonstrated a good capability of generating the MNIST hand-written digits. We believe it might be possible to extrapolate this performance to generate images representing 2D normal distributions instead. Coming back to the evaluation of the data transformations discussed in section 5.1 with this model might be an interesting addition.
- Even if deep convolution GANs are not enough, they are a model developed in 2014. Eight years in the field of artificial intelligence can be thought of as a long period of time. In the meantime, many other models have been proposed, and the state-of-the-art GANs nowadays seem to be able to do impressive things with usual images. Attempting to use more complex models, although it might be overcomplicating the setup and lead to non-sensical results, might be a sensible alternative. We believe WGANs could solve some of the problems we encountered.
- It might be possible that leaving aside from the fact that they share the Gaussian nature, the datasets we employed throughout the experiment might be problematic for generative machine learning methods to generate. This is remarkable because given that it looks like the simplest datasets to generate out of the three experiments, no matter the hyperparameter tuning options, our models seem incapable of producing high-quality simulations. We propose that it might be wise to, after understanding properly how the algorithms work, skip the step of using toy data for training and attempt to see the results produced by the models trained on real particle collisions.

References

[1] MCatNLO online documentation accessed in 2022. URL: https://www.hep.phy.cam.ac.uk/theory/webber/MCatNLO/.

[2] Short description of the leaky ReLU activation function. Accessed in April 2022. URL: https://paperswithcode.com/method/leaky-relu.

[3] Official online documentation and tutorials for the Tensorflow framework. Accessed in April 2022. URL: https://www.tensorflow.org/api_docs/python/tf/keras.

[4] Official online documentation including updates of the open-source framework, numpy. Accessed in April 2022. URL: https://numpy.org/.

[5] Abien Fred Agarap. "Deep learning using rectified linear units (relu)". In: *arXiv preprint arXiv:1803.08375* (2018).

[6] Johan Alwall et al. "MadGraph 5: going beyond". In: *Journal of High Energy Physics* 2011.6 (2011). DOI: 10.1007/jhep06(2011)128.

[7] Martin Arjovsky, Soumith Chintala, and Léon Bottou. *Wasserstein GAN*. 2017. DOI: 10.48550/ARXIV.1701.07875. URL: https://arxiv.org/abs/1701.07875.

[8] Andrea Asperti. *Variance Loss in Variational Autoencoders*. 2020. DOI: 10.48550/ARXIV.2002.09860. URL: https://arxiv.org/abs/2002.09860.

[9] R. Atienza. *Advanced Deep Learning with TensorFlow 2 and Keras: Apply DL, GANs, VAEs, deep RL, unsupervised learning, object detection and segmentation, and more, 2nd Edition*. Packt Publishing, 2020. ISBN: 9781838825720. URL: https://books.google.co.uk/books?id=68rTDwAAQBAJ.

[10] Manuel Bähr et al. "Herwig++ physics and manual". In: *The European Physical Journal C* 58.4 (2008), pp. 639–707. DOI: 10.1140/epjc/s10052-008-0798-9.

[11] A. Bevan. *Statistical Data Analysis for the Physical Sciences*. Cambridge University Press, 2013. ISBN: 9781107067592. URL: https://books.google.co.uk/books?id=tdAAYT7J71AC.

[12] Enrico Bothmann et al. "Event generation with Sherpa 2.2". In: *SciPost Physics* 7.3 (2019), pp. 852–867. DOI: 10.21468/scipostphys.7.3.034.

[13] Andrew Brock, Jeff Donahue, and Karen Simonyan. *Large Scale GAN Training for High Fidelity Natural Image Synthesis*. 2018. DOI: 10.48550/ARXIV.1809.11096. URL: https://arxiv.org/abs/1809.11096.

[14] Wenxiao Chen et al. "VAEPP: Variational Autoencoder with a Pull-Back Prior". In: *Neural Information Processing: 27th International Conference, ICONIP 2020, Bangkok, Thailand, November 23–27, 2020, Proceedings, Part III*. Bangkok, Thailand: Springer-Verlag, 2020, pp. 366–379. ISBN: 978-3-030-63835-1. DOI: 10.1007/978-3-030-63836-8_31. URL: https://doi.org/10.1007/978-3-030-63836-8_31.

[15] Soham De, Anirbit Mukherjee, and Enayat Ullah. *Convergence guarantees for RMSProp and ADAM in non-convex optimization and an empirical comparison to Nesterov acceleration*. 2018. DOI: 10.48550/ARXIV.1807.06766. URL: https://arxiv.org/abs/1807.06766.

[16] Riccardo Di Sipio et al. "DijetGAN: a Generative-Adversarial Network approach for the simulation of QCD dijet events at the LHC". In: *Journal of High Energy Physics* 2019.8 (2019). DOI: 10.1007/jhep08(2019)110.

[17] Alexey Dosovitskiy and Thomas Brox. *Generating Images with Perceptual Similarity Metrics based on Deep Networks*. 2016. DOI: 10.48550/ARXIV.1602.02644. URL: https://arxiv.org/abs/1602.02644.

[18] D. Foster. *Generative Deep Learning: Teaching Machines to Paint, Write, Compose, and Play*. O'Reilly Media, 2019. ISBN: 9781492041894. URL: https://books.google.co.uk/books?id=RqegDwAAQBAJ.

[19] David Foster. *Generative deep learning : teaching machines to paint, write, compose, and play / David Foster*. eng. 2019. ISBN: 9781492041948.

[20] Federico Girosi, Michael Jones, and Tomaso Poggio. "Regularization Theory and Neural Networks Architectures". In: *Neural Computation* 7.2 (Mar. 1995), pp. 219–269. ISSN: 0899-7667. DOI: 10.1162/neco.1995.7.2.219. eprint: https://direct.mit.edu/neco/article-pdf/7/2/219/812917/neco.1995.7.2.219.pdf. URL: https://doi.org/10.1162/neco.1995.7.2.219.

[21] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. The MIT Press, 2017.

[22] Ian J. Goodfellow et al. *Generative Adversarial Networks*. 2014. DOI: 10.48550/ARXIV.1406.2661. URL: https://arxiv.org/abs/1406.2661.

[23] Simon Haykin. *Neural networks: a comprehensive foundation*. Prentice Hall PTR, 1994.

[24] Xianxu Hou et al. *Deep Feature Consistent Variational Autoencoder*. 2016. DOI: 10.48550/ARXIV.1610.00291. URL: https://arxiv.org/abs/1610.00291.

[25] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. DOI: 10.48550/ARXIV.1502.03167. URL: https://arxiv.org/abs/1502.03167.

[26] Tero Karras, Samuli Laine, and Timo Aila. *A Style-Based Generator Architecture for Generative Adversarial Networks*. 2018. DOI: 10.48550/ARXIV.1812.04948. URL: https://arxiv.org/abs/1812.04948.

[27] Tero Karras et al. *Progressive Growing of GANs for Improved Quality, Stability, and Variation*. 2017. DOI: 10.48550/ARXIV.1710.10196. URL: https://arxiv.org/abs/1710.10196.

[28] Diederik P Kingma and Max Welling. *Auto-Encoding Variational Bayes*. 2013. DOI: 10.48550/ARXIV.1312.6114. URL: https://arxiv.org/abs/1312.6114.

[29] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. DOI: 10.48550/ARXIV.1412.6980. URL: https://arxiv.org/abs/1412.6980.

[30] S. Kullback and R. A. Leibler. "On Information and Sufficiency". In: *The Annals of Mathematical Statistics* 22.1 (1951), pp. 79–86. DOI: 10.1214/aoms/1177729694. URL: https://doi.org/10.1214/aoms/1177729694.

[31] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". In: *nature* 521.7553 (2015), p. 436.

[32] Yann A. LeCun et al. "Efficient BackProp". In: *Neural Networks: Tricks of the Trade: Second Edition*. Ed. by Grégoire Montavon, Geneviève B. Orr, and Klaus-Robert Müller. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 9–48. ISBN: 978-3-642-35289-8. DOI: 10.1007/978-3-642-35289-8_3. URL: https://doi.org/10.1007/978-3-642-35289-8_3.

[33] Ming-Yu Liu and Oncel Tuzel. *Coupled Generative Adversarial Networks*. 2016. DOI: 10.48550/ARXIV.1606.07536. URL: https://arxiv.org/abs/1606.07536.

[34] Xudong Mao et al. *Least Squares Generative Adversarial Networks*. 2016. DOI: 10.48550/ARXIV.1611.04076. URL: https://arxiv.org/abs/1611.04076.

[35] Konstantin Matchev, Alexander Roman, and Prasanth Shyamsundar. "Uncertainties associated with GAN-generated datasets in high energy physics". In: *SciPost Physics* 12.3 (2022). DOI: 10.21468/scipostphys.12.3.104.

[36] Mehdi Mirza and Simon Osindero. *Conditional Generative Adversarial Nets*. 2014. DOI: 10.48550/ARXIV.1411.1784. URL: https://arxiv.org/abs/1411.1784.

[37] Chigozie Nwankpa et al. *Activation Functions: Comparison of trends in Practice and Research for Deep Learning*. 2018. DOI: 10.48550/ARXIV.1811.03378. URL: https://arxiv.org/abs/1811.03378.

[38] C. Oleari. "The POWHEG BOX". In: *Nuclear Physics B - Proceedings Supplements* 205-206 (2010), pp. 36–41. DOI: 10.1016/j.nuclphysbps.2010.08.016.

[39] Martin J. Osborne. *Introduction to Game Theory: International Edition*. OUP Catalogue 9780195322484. Oxford University Press, 2009. ISBN: ARRAY(0x500d83c8). URL: https://ideas.repec.org/b/oxp/obooks/9780195322484.html.

[40] Juan Qiu, Qingfeng Du, and Chongshu Qian. "KPI-TSAD: A Time-Series Anomaly Detector for KPI Monitoring in Cloud Applications". In: *Symmetry* 11.11 (2019). ISSN: 2073-8994. DOI: 10.3390/sym11111350. URL: https://www.mdpi.com/2073-8994/11/11/1350.

[41] Alec Radford, Luke Metz, and Soumith Chintala. *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks*. 2015. DOI: 10.48550/ARXIV.1511.06434. URL: https://arxiv.org/abs/1511.06434.

[42] F. Rosenblatt. "The perceptron: A probabilistic model for information storage and organization in the brain." In: *Psychological Review* 65.6 (1958), pp. 386–408. ISSN: 0033-295X. DOI: 10.1037/h0042519. URL: http://dx.doi.org/10.1037/h0042519.

[43] Torbjörn Sjöstrand, Stephen Mrenna, and Peter Skands. "A brief introduction to PYTHIA 8.1". In: *Computer Physics Communications* 178.11 (2008), pp. 852–867. DOI: 10.1016/j.cpc.2008.01.036.

[44] A. Valassi, E. Yazgan, and J. McFayden. *A. Valassi – MC generators challenges and strategy towards HL-LHC LHCC – 01 Sep 2020 1 Monte Carlo generators challenges and strategy towards HL-LHC*. 2020.

[45] S. Vallecorsa. "Generative models for fast simulation". In: *Journal of Physics: Conference Series* 1085 (2018), p. 022005. DOI: 10.1088/1742-6596/1085/2/022005.

[46] Han Zhang et al. *Self-Attention Generative Adversarial Networks*. 2018. DOI: 10.48550/ARXIV.1805.08318. URL: https://arxiv.org/abs/1805.08318.

[47] Jun-Yan Zhu et al. *Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks*. 2017. DOI: 10.48550/ARXIV.1703.10593. URL: https://arxiv.org/abs/1703.10593.

A Algorithms [21]

Algorithm 1: Forward propagation through a typical deep neural network and the computation of the cost function. The loss $L(\hat{\mathbf{y}}, \mathbf{y})$ depends on the output $\hat{\mathbf{y}}$ and on the target $\mathbf{y}$. To obtain the total cost J, the loss may be added to a regulariser $\Omega(\theta)$, where θ contains all the parameters (weights and biases). Algorithm 2 shows how to compute gradients of J with respect to parameters $\mathbf{W}$ and $\mathbf{b}$. For simplicity, this demonstration uses only a single input example $\mathbf{x}$. Practical applications should use a minibatch.

Require: Network depth, l
Require: $\mathbf{W}^{(i)}, i \in 1, ..., l$, the weight matrices of the model
Require: $\mathbf{b}^{(i)}, i \in 1, ..., l$, the bias parameters of the model
Require: $\mathbf{x}$, the input to process
Require: $\mathbf{y}$, the target output
$\mathbf{h}^{(0)} = \mathbf{x}$
for $k = 1, ..., l$ **do**
$\mathbf{a}^{(k)} = \mathbf{b}^{(k)} + \mathbf{W}^{(k)}\mathbf{h}^{(k-1)}$
$\mathbf{h}^{(k)} = f(\mathbf{a}^{(k)})$
end for
$\hat{\mathbf{y}} = \mathbf{h}^{(l)}$
$J = L(\hat{\mathbf{y}}, \mathbf{y}) + \lambda\Omega(\theta)$

Algorithm 2: Backward computation for the deep neural network of algorithm 1, which uses, in addition to the input $\mathbf{x}$, a target $\mathbf{y}$. This computation yields the gradients on the activation $\mathbf{a}^{(k)}$ for each layer k, starting from the output layer and going backwards to the first hidden layer. From these gradients, which can be interpreted as an indication of how each layer's output should change to reduce error, one can obtain the gradient on the parameters of each layer. The gradients on weights and biases can be immediately used as part of a stochastic gradient update (performing the update right after the gradients have been computed) or used with the other gradient-based optimisation methods.

After the forwards computation, compute the gradient on the output layer:
$\mathbf{g} \longleftarrow \nabla_{\hat{y}} J = \nabla_{\hat{y}} L(\hat{\mathbf{y}}, \mathbf{y})$
for $k = l, l-1, ..., 1$ **do**

Convert the gradient on the layer's output into a gradient on the pre-nonlinearity activation (element-wise multiplication if f is element-wise):

$\mathbf{g} \longleftarrow \nabla_{\alpha^{(k)}} J = \mathbf{g} \otimes f'^{(\mathbf{a}^{(k)})}$
Compute the gradients on weights and biases (including the regularisation term, where needed):
$\nabla_{\mathbf{b}^{(k)}} J = \mathbf{g} + \lambda\nabla_{\mathbf{b}^{(k)}}\Omega(\theta)$
$\nabla_{\mathbf{W}^{(k)}} J = \mathbf{g}\mathbf{h}^{(k-1)^T} + \lambda\nabla_{\mathbf{W}^{(k)}}\Omega(\theta)$
Propagate the gradients w.r.t. the next lower-level hidden layer's activations:
$\mathbf{g} \longleftarrow \nabla_{\mathbf{h}^{(k-1)}} J = \mathbf{W}^{(k)^T}\mathbf{g}$
end for

Listing 1: Functions contained in the file helpers.py - Part 1.

```
# Note that we are skipping the lines that import the necessary packages

def toy_data(n, rmin, rmax):
    """
    Function to generate toy data. Random uniform data is generated in the
    range provided and then it goes through a mathematical function to generate a second
    dimension of data. This function is for 2D data.
    Parameters:
        - n: Number of data points to generate
        - rmin: Minimum value of the range
        - rmax: Maximum value of the range
    """
    x1 = np.random.uniform(rmin, rmax, n)
    x2 = x1 ** 2
    x1 = x1.reshape((n, 1))
    x2 = x2.reshape((n, 1))
    samples = np.hstack((x1, x2))

    return samples


def show_samples(generated_point_list, epoch, generator, data, n=100, l_dim=5):
    """
    Function that shows the generated samples every 20 epochs and compares
    them with the real data.
    Parameters:
        - generated_point_list: List of generated points
        - epoch: Current epoch
        - generator: Generator model
        - data: Real data
        - n: Number of samples to show
        - l_dim: Dimension of the latent space
    """
    if epoch % 20 == 0:
        noise = tf.random.normal(shape=(n, l_dim))
        generated_data = generator(noise)
        generated_point_list.append(generated_data)


def make_animation(real_data: tuple, generated_point_list):
    """
    Function that makes an animation of the generated data.
    It can be used to evaluate how the generator is doing with respect to time
    or epochs.
    Parameters:
        - real_data: Real data as a tuple
        - generated_point_list: List of generated points
    """
    real_x, real_y = real_data

    camera = Camera(plt.figure())

    plt.xlim(real_x.min() - 0.2, real_x.max() + 0.2)
    plt.ylim(real_y.min() - 0.05, real_y.max() + 0.05)

    for i in range(len(generated_point_list)):
        plt.scatter(real_x, real_y, c='blue')
        fake_x, fake_y = generated_point_list[i][:, 0], generated_point_list[i][:, 1]
        plt.scatter(fake_x, fake_y, c='red')
        camera.snap()

    animation = camera.animate(blit=True)
    plt.close()
    animation.save('animation.gif', fps=10)
```

Listing 2: Functions contained in the file helpers.py - Part 2.

```
def stats_dist(dist, dist2=None, prnt=True):
    """
    Calculates the mean, standard deviation, variance, and covariance matrix of a
    distribution. The function also includes the option to calculate the covariance
    matrix of two distributions. To do this, pass the second distribution as the
    second parameter.
    Parameters:
        - dist (numpy array): The distribution to be analyzed.
        - dist2 (numpy array): The second distribution to be analyzed.
        - prnt (bool): Whether or not to print the results. If print is False,
        the function will simply return the results.

    Output:
        - Mean, standard deviation, variance, covariance of the first distribution.
        If a dist2 is included then, the covariance takes into account dist2 as well.
    """
    if prnt:
        print("--------------------------------")
        print("Mean:", np.mean(dist))
        print("--------------------------------")
        print("Standard deviation:", np.std(dist))
        print("--------------------------------")
        print("Variance:", np.var(dist))
        print("--------------------------------")
        if dist2 is not None:
            print("Covariance matrix:", np.cov(dist, dist2))
            return np.mean(dist), np.std(dist), np.var(dist), np.cov(dist, dist2)
        else:
            print("Covariance:", np.cov(dist))  # Will be a scalar
    else:
        if dist2 is not None:
            return np.mean(dist), np.std(dist), np.var(dist), np.cov(dist, dist2)

    return np.mean(dist), np.std(dist), np.var(dist), np.cov(dist)


def df_generator(stats_or_x, stats_gen_x, stats_or_y, stats_gen_y, rows: np.ndarray,
df_style):
    """
    Generates a dataframe with the given parameters.
    Parameters:
        - stats_or_x: The stats of the x-dimension from the train data.
        - stats_gen_x: The stats of the x-dimension from the generated data.
        - stats_or_y: The stats of the y-dimension from the train data.
        - stats_gen_y: The stats of the y-dimension from the generated data.
        - rows: Numpy array with the rows of the dataframe as strings.
        - df_style: The style of the dataframe. Should be coming from a function
        like "df_style"
    """
    stats_df = pd.DataFrame()
    stats_df[" "] = rows
    stats_df["original_x_dim"] = np.array([stats_or_x[0], stats_or_x[1],
    stats_or_x[2], stats_or_x[3].item()])

    stats_df["generated_x_dim"] = np.array([stats_gen_x[0],
    stats_gen_x[1], stats_gen_x[2], stats_gen_x[3].item()])

    stats_df["original_y_dim"] = np.array([stats_or_y[0], stats_or_y[1],
    stats_or_y[2], stats_or_y[3].item()])

    stats_df["generated_y_dim"] = np.array([stats_gen_y[0],
    stats_gen_y[1], stats_gen_y[2], stats_gen_y[3].item()])

    stats_df = stats_df.style.applymap(df_style, subset=[" "])

    return stats_df
```

Listing 3: Functions contained in the file helpers.py - Part 3.

```
def chi_squared(h1, h2):
    """
    Computes the chi squared between two histograms. For the computation, it is designed
    to just take into consideration the heights of each of the bins.

    Parameters:
        - h1: array with the heights of each of the bins of histogram 1.
        - h2: array with the heights of each of the bins of histogram 2.

    Output:
        - Chi squared coefficient
    """
    coeff = np.sum(((h2-h1)**2)/h1)

    return coeff


def align_hist(plot_ht, plot_ht_2, nbins):
    """
    Function that returns a list with new values for the bins' bounds. This is done by
    redefining the range of both histograms, generalise it, and divide that range over
    the number of desired bins. This list can be used further in the function
    "norm_hist".

    Parameters:
        - plot_ht: the output of the function plt.hist(HIST_1)
        - plot_ht_2: the output of the function plt.hist(HIST_2)
        - nbins: number of bins desired
    """
    h1 = plot_ht[0]; x_val = plot_ht[1]
    h2 = plot_ht_2[0]; x_val_2 = plot_ht_2[1]

    min_point = min([min(x_val), min(x_val_2)])
    max_point = max([max(x_val), max(x_val_2)])
    rge = max_point - min_point

    bin_sep = rge/nbins
    bins_val = [min_point]
    for i in range(nbins):
        point = bins_val[i] + bin_sep
        bins_val.append(point)

    return bins_val

def norm_hist(h1, bins_val):
"""
Function that gets rid of the bins with 0 height by joining them together with the
closest non-zero bin. This avoids getting an undefined result when "chi_squared" is used.

Parameters:
    - h1: height of the histogram 1 (Expected values/real histogram)
    - bins_val: aligned list of values corresponding to the bounds of the bins.
    (Generally, output of align_hist)

Output:
    - List with the new bins with non-zero height.
"""
zero_idx = np.where(h1 == 0)[0]
for i in zero_idx:
    non_zero_idx = i
    while non_zero_idx in zero_idx:
        non_zero_idx += 1
    bins_val[i] = bins_val[non_zero_idx]
bins_val = list(dict.fromkeys(bins_val))

return bins_val
```

Listing 4: Code used in the attempt of transforming the space of the true distributions.

```
import numpy as np

class simple_generation:
    """
Class containing the necessary methods in order to be able to generate
toy data that will be fed into the variational  autoenconder and GAN.
It has several methods used to create and transform data distributions
from one space to others being able to go back.
    """

    def __init__(self, mean, std, no_samples):
        """
        Object initiates creating a gaussian distribution of data taking the inputs:

        - mean (float/int)
        - std (float/int): standard deviation.
        - no_samples (int): number of samples that go into the distributions.
        """
        self.mean = mean
        self.std = std
        self.no_samples = no_samples
        self.init_dist = np.random.normal(mean, std, no_samples)

    def init_dist(self):
        """
        Method that returns the initial gaussian distribution.
        """
        return self.init_dist

    def quad_fit(self):
        """
        Method that fits the normally distributed data into a quadatric distribution.
        """
        c = np.log(1/(np.sqrt(2*np.pi)*self.std))
        self.c = c
        quad_function = lambda x: c - ((x-self.mean)**2)/(2*self.std**2)
        quad_dist = quad_function(self.init_dist)

        self.quad_function = quad_function

        return quad_dist

    def linear_fit(self):
        """
        Method that fits the normally distributed data into a linear distribution.
        """
        lin_function = np.sqrt(-self.quad_fit() + self.c)
        return lin_function

    def c_return(self):
        """
        Simple method that returns the value of the constant used to do
        the data transformations, c.
        """
        return self.c
```

Listing 5: Code that defines the generator and discriminator networks in the Glash based on a multilayer perceptron

```
def glash_discriminator(n=2, act_fun_1=tf.keras.activations.relu):
    """
    Discriminator network side of the GAN. It consists of a sequence of
    fully connected dense layers that output a
    probability of the input being real (1) or fake (0). It aims to
    distinguish between real and fake distributions
    (generation samples from the generative network).

    Arguments:
        - n: number of samples it takes as an input in 1D
    """
    model = tf.keras.Sequential([
        tf.keras.layers.Input(shape=(n,)),
        tf.keras.layers.Dense(15, activation=act_fun_1,
                              kernel_initializer='he_uniform'),
        tf.keras.layers.Dense(10, activation=act_fun_1,
                              kernel_initializer='he_uniform'),
        #tf.keras.layers.Dense(50, activation=act_fun_1,
        #                      kernel_initializer='he_uniform'),
        tf.keras.layers.Dense(1, activation='sigmoid')
    ], name='glash_discriminator')

    return model


def glash_generator(latent_dim=5, act_fun_1=tf.keras.activations.relu,
act_fun_2=tf.keras.activations.relu):
    """
    Generative network component of the GAN. It consists of a sequence of
    fully connected dense layers that output a
    newly generated distribution. It aims to generate a distribution that
    is close to the input distribution improving
    its weights and biases until the discriminator network is no longer
    able to recognise fake from real data.

    Arguments:
        - latent_dim: number of dimensions of the latent space
    """
    model = tf.keras.Sequential([
        tf.keras.layers.Input(shape=(latent_dim,)),
        tf.keras.layers.Dense(10, activation=act_fun_1,
                              kernel_initializer='he_uniform'),
        tf.keras.layers.Dense(10, activation=act_fun_1,
                             kernel_initializer='he_uniform'),
        tf.keras.layers.Dense(2, activation=act_fun_2)
    ], name='glash_generator')

    return model
```

Listing 6: Python class used to initialise Glash

```
class Glash(tf.keras.Model):

    def __init__(self, discriminator, generator, latent_dim=5):
        super(Glash, self).__init__()
        self.discriminator = discriminator
        self.generator = generator
        self.latent_dim = latent_dim

    def compile(self, optimizerD, optimizerG, loss_fn, loss_fn_2):
        super(Glash, self).compile()
        self.optimizerD = optimizerD
        self.optimizerG = optimizerG
        self.loss_fn = loss_fn
        self.loss_fn_2 = loss_fn_2

    def train_step(self, real_data):
        if isinstance(real_data, tuple):
            real_data = real_data[0]

        # Generate fake data with the appropriate shape and batch size
        batch_size = tf.shape(real_data)[0]
        noise = tf.random.normal(shape=(batch_size, self.latent_dim))
        fake_data = self.generator(noise)

        # Combine real and fake data for training the discriminator
        combined_data = tf.concat([real_data, fake_data], axis=0)
        labels = tf.concat([tf.ones((batch_size, 1)),
            tf.zeros((batch_size, 1))], axis=0)

        # Training the discriminator
        with tf.GradientTape() as tape:
            predictions = self.discriminator(combined_data)
            d_loss = self.loss_fn(labels, predictions)

        # Update the weights of the discriminator
        grads = tape.gradient(d_loss, self.discriminator.trainable_weights)
        self.optimizerD.apply_gradients(zip(grads,
            self.discriminator.trainable_weights))

        # Generate fake labels to train the generator
        misleading_labels = tf.ones((batch_size, 1))
        noise = tf.random.normal(shape=(batch_size, self.latent_dim))

        # Training the generator
        with tf.GradientTape() as tape:
            fake_predictions = self.discriminator(self.generator(noise))
            g_loss = self.loss_fn_2(misleading_labels, fake_predictions)

        # Update the weights of the generator
        grads = tape.gradient(g_loss, self.generator.trainable_weights)
        self.optimizerG.apply_gradients(zip(grads, self.generator.trainable_weights))

        return {'d_loss': d_loss, 'g_loss': g_loss, 'fake_pred': fake_predictions}

    def call(self, noise):
        return self.generator(noise)
```

Listing 7: Development code of DCGlash - the deep convolutional GAN used the MNIST experiment.

```
def build_generator(inputs, image_size):

    image_resize = image_size // 4
    # network parameters
    kernel_size = 5
    layer_filters = [128, 64, 32, 1]

    x = Dense(image_resize * image_resize * layer_filters[0])(inputs)
    x = Reshape((image_resize, image_resize, layer_filters[0]))(x)

    for filters in layer_filters:
        # First two convolution layers use strides = 2
        # the last two use strides = 1
        if filters > layer_filters[-2]:
            strides = 2
        else:
            strides = 1

        x = BatchNormalization()(x)
        x = Activation("relu")(x)
        x = Conv2DTranspose(filters=filters,
                            kernel_size=kernel_size,
                            strides=strides,
                            padding='same')(x)

    tx = Activation('sigmoid')(x)
    generator = Model(inputs, x, name='generator')

    return generator

def build_discriminator(inputs):

    kernel_size = 5
    layer_filters  = [32, 64, 128, 256]

    x = inputs
    for filters in layer_filters:
        #first 3 convolution layers use stride = 2
        #last one uses stride = 1
        if filters == layer_filters[-1]:
            strides = 1

        else:
            strides = 2

        x = LeakyReLU(alpha=0.2)(x)
        x = Conv2D(filters=filters,
                   kernel_size=kernel_size,
                   strides=strides,
                   padding='same')(x)

    x = Flatten()(x)
    x = Dense(1)(x)
    x = Activation('sigmoid')(x)
    discriminator = Model(inputs, x, name='discriminator')

    return discriminator
```

Listing 8: Part 2 of the code used for the MNIST generator. This contains a function that merges the networks and calls for training.

```
def build_and_train_models():
    # Load MNIST dataset
    (x_train, _), (_, _) = mnist.load_data()

    # Reshape data for CNN as (28, 28, 1) and normalise
    image_size = x_train.shape[1]
    x_train = np.reshape(x_train, [-1, image_size, image_size, 1])
    x_train = x_train.astype('float32')/255

    model_name = 'dc_glash'
    # Network parameters
    # the latent or z vector is 100-dim
    latent_size = 100
    batch_size = 64
    train_steps = 10000
    lr = 2e-4
    decay = 6e-8
    input_shape = (image_size, image_size, 1)

    # Build discriminator model
    inputs = Input(shape=input_shape, name='discriminator_input')
    discriminator = build_discriminator(inputs)

    # Original paper uses Adam, but the discrinator converges easily with RMSprop
    optimizer = RMSprop(lr, decay=decay)
    discriminator.compile(loss='binary_crossentropy',
                          optimizer=optimizer,
                          metrics=['accuracy'])
    discriminator.summary()

    # Build generator model
    input_shape = (latent_size, )
    inputs = Input(shape=input_shape, name='z_input')
    generator = build_generator(inputs, image_size)
    generator.summary()

    # Build adversarial model
    optimizer = RMSprop(lr=lr * 0.5, decay=decay * 0.5)
    # Freeze the weights of discriminator during adversarial training
    discriminator.trainable = False
    # adversarial = generator + discriminator
    adversarial = Model(inputs,
                        discriminator(generator(inputs)),
                        name=model_name)
    adversarial.compile(loss='binary_crossentropy',
                        optimizer=optimizer,
                        metrics=['accuracy'])
    adversarial.summary()

    # train discriminator
    models = (generator, discriminator, adversarial)
    params = (batch_size, latent_size, train_steps, model_name)
    train(models, x_train, params)
```

Listing 9: Custom training sequence created for DCGlash on MNIST handwritten digits.

```
def train(models, x_train, params):


    # The GAN component models
    generator, discriminator, adversarial = models
    # network parameters
    batch_size, latent_size, train_steps, model_name = params
    # the generator image is saved every 500 steps
    save_interval = 500
    # noise vector to see how the generator output evolves during training
    noise_input = np.random.uniform(-1.0, 1.0, size=[16, latent_size])

    # number of elements in train dataset
    train_size = x_train.shape[0]
    for i in range(train_steps):
        # train the discriminator for 1 batch
        # 1 batch of real (label=1) and fake images (label=0)
        # randomly pick real images from dataset
        rand_indexes = np.random.randint(0, train_size, size=batch_size)
        real_images = x_train[rand_indexes]

        # generate fake images from noise using generator
        # generate noise using uniform distribution
        noise = np.random.uniform(-1.0, 1.0, size=[batch_size, latent_size])

        # generate fake images
        fake_images = generator.predict(noise)
        # real + fake images = 1 batch of train data
        x = np.concatenate((real_images, fake_images))
        # label real images is 1
        y = np.ones([2 * batch_size, 1])
        # fake images label is 0
        y[batch_size:, :] = 0
        # train discriminator network, log the loss and accuracy
        loss, acc = discriminator.train_on_batch(x, y)
        log = "%d: [discriminator loss: %f, acc: %f]" % (i, loss, acc)

        # train the adversarial network for 1 batch
        # since the discriminator weights are frozen in adversarial
        network only the generator is trained
        noise = np.random.uniform(-1.0, 1.0, size=[batch_size, latent_size])

        y = np.ones([batch_size, 1])
        # train the adversarial network
        loss, acc = adversarial.train_on_batch(noise, y)
        log = "%s [adversarial loss: %f, acc: %f]" % (log, loss, acc)
        print(log)

        if (i + 1) % save_interval == 0:
            # plot generator images on a periodic basis
            plot_images(generator,
                        noise_input=noise_input,
                        show=False,
                        step=(i + 1),
                        model_name=model_name)

        # save the model after training the generator
        # the trained generator can be reloaded for future MNIST digit generatio
        generator.save(model_name + ".h5")
```

Listing 10: Development code of Kuyf - the variational autoencoder model used for 2D histogram generation.

```
import tensorflow as tf
from tensorflow import keras
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd


class Sampling(keras.layers.Layer):
    """
    samples codings from a normal distribution.samples a random vector from a normal
    distribution with mean 0 and standard deviation of 1
    """
    def call(self, inputs):
        mean, log_var = inputs
        return tf.random.normal(tf.shape(log_var)) * tf.exp(log_var / 2) + mean


# Encoder part

inputs = keras.layers.Input(shape=[2,features_num])#2D data"
z = keras.layers.Flatten()(inputs)
z = keras.layers.Dense(100, activation='selu')(z)
z = keras.layers.Dense(50, activation='selu')(z)
codings_mean = keras.layers.Dense(coding_size)(z)
codings_log_var = keras.layers.Dense(coding_size)(z)
codings = Sampling()([codings_mean, codings_log_var])
encoder = keras.Model(inputs=inputs, outputs=[
                            codings_mean, codings_log_var, codings])

# Decoder part

decoder_inputs = keras.layers.Input(shape=[coding_size])
x = keras.layers.Dense(50, activation='selu')(decoder_inputs)
x = keras.layers.Dense(100, activation='selu')(x)
x = keras.layers.Dense(2*features_num,activation=tf.keras.layers.LeakyReLU())(x)
outputs = keras.layers.Reshape([2,features_num])(x)
decoder = keras.Model(inputs=decoder_inputs, outputs=outputs)

# Combine Encoder and Decoder

_, _, codings = encoder(inputs)
reconstructions = decoder(codings)
kuyf = keras.Model(inputs=inputs, outputs=reconstructions)

latent_loss = -0.5 * tf.reduce_sum(1 + codings_log_var - tf.exp(
codings_log_var) - tf.square(codings_mean))

kuyf.add_loss(latent_loss)
kuyf.compile(loss=tf.keras.losses.KLDivergence(),
                        optimizer=keras.optimizers.Adam())
```