BACHELOR'S THESIS

Degree in Mathematics

Title: Analysis of state-of-the-art deep generative models for images

Author: Víctor Salvia Punsoda

Advisor: Alberto Sanfeliu Cortés

Department: Mobile Robotics and Intelligent Systems, IRI

Academic year: 2019-2020

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat de Matemàtiques i Estadística

Universitat Politècnica de Catalunya
Facultat de Matemàtiques i Estadística

Degree in Mathematics
Bachelor's Degree Thesis

Analysis of state-of-the-art deep generative models for images

Víctor Salvia Punsoda

Supervised by Alberto Sanfeliu Cortés

June, 2020

First of all, I would like to thank Alberto for his guidance, passion, and trust. I will always remember how he introduced me to such an exciting and useful tool for the future. I also wanted to thank Javi and Albert for their help throughout the whole project. Finally, thanks to my family and friends for their unconditional support.

Abstract

In this work we will try to break down the fundamentals of deep generative models for image generation. For the sake of simplicity and understanding we will be using the MNIST dataset, which can be easily trained and understood. The algorithms that we will be explaining and comparing are an autoregressive model, a flow-based model, a variational autoencoder, and a generative adversarial network. Within each of these categories, we have chosen simple algorithms for the sake of clarity as well. Keep in mind that this is a wide but introductory work to the area of image generation. However, these algorithms were certainly state-of the-art around 2016.

Keywords

Machine Learning, Deep Learning, Artificial Neural Networks, Deep Generative models, Image Generation, Computer Vision, GANs, VAEs, Autoregressive Models, Flow-Based Models.

Contents

1. Introduction

In this section we will motivate the work done as well as present the specific problem that we want to solve.

1.1 Motivation of deep generative models

Figure 1: Faces generated by GANs [10]

We live in a world where data is becoming increasingly important. Classical statistical models help us with reasonable amounts of data, but when dimensions are high (images, voice, text...) they fail to capture meaningful features. This is the case for images and computer vision. So far, deep learning has had a big impact in **supervised learning** [1], where we basically prepare the data so that the algorithm knows what we want 'him' to learn. Then, showing 'him' these prepared examples it learns the specific task. For instance, we could train a supervised model to distinguish cats from dogs. Conversely, **unsupervised learning** makes no use of labelled data. This is much more real and practical, since we can use raw data, but way more challenging. One of the possible unsupervised tasks is to **model the probability distribution of a given dataset** - such as handwritten digits or faces. Some important examples are:

- Images (GANs)
- Speech (WaveNet)
- Texts (GPT-2)

These kinds of data are not normally labelled unless we invest a significant amount of time in doing so. Great achievements in the area of unsupervised learning are the generation of images with GANs, the generation of voice with WaveNet, and text generation with GPT2.

[1] Task of learning a function that maps an input to an output based on example input-output pairs.

1. First of all, we show in 1 the performance of GANs in image generation. This astonishing result is current state-of-the-art. The idea is that behind this high-dimensional data (we may be talking about $1024x1024x3 \approx 3x10^6$ pixels per face) there is an underlying distribution of faces. The problem is that working in this space is extremely challenging both computationally and mathematically. The models used in these situations are radically different from the ones that we may use in statistics to learn the distribution of some random variable.

2. Secondly, the example of voice generation is also surprising. The WaveNet model [17] does extremely well in this task, you can check it out at [2]. It is particularly interesting to see how good they are at music generation.

3. The last example is the latest language model, GPT-2 [19]. This can be used openly at [11]. You start writing some texts and it completes the rest of it. You can see an example in 2. In this case our input is the sentence *"Deep learning is an extremely cool technique"*, and the model completes the whole text. This is an astonishing result given all the restrictions and rules involved in language.

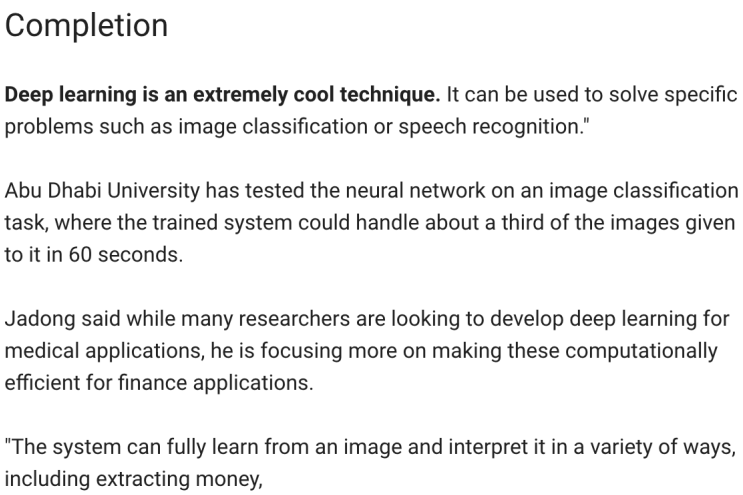

Figure 2: Text generated by GPT-2.

What these applications have in common is that we are modelling a certain high-dimensional distribution that captures the patterns of our data. The goal in generative models is three-fold:

- Density estimation (anomaly detection)
- Sampling (data generation)
- Compression (feature learning)

We will be working with handwritten digits, trying to approximate the specific density function so that we can compute the probability of an image - whether it comes from our real distribution or not. Also, we want to be able to sample from this distribution. Finally, we would like to be able to compress data so that we can represent high dimensional data in just a few bits. For example, if our image is 1000 dimensional, maybe we can find a representation of dimension 10 that captures the

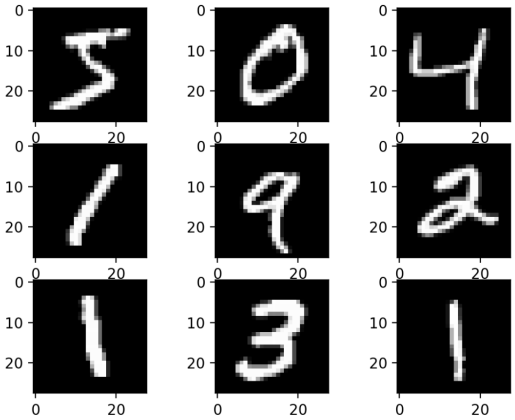

Figure 3: MNIST examples.

essential patterns in our data.
In conclusion, supervised learning is cool but quite limited. Not only because we need the data to be labelled but also because it is not the same distinguishing handwritten digits than to actually writing them. I think everyone would agree, in that knowing that a song is from Beethoven, is much easier than actually being able to compose a Beethoven-like song. This is exactly what we are doing here. We are modelling a distribution, in the space of all songs, that attributes more probability to the Beethoven-like songs and less to the others. Hence, if we sample from this distribution we will most likely get a Beethoven-like one.

1.2 Generation of images: Handwritten digits

Given a data set $\{\mathbf{x}^{(t)}\}_{t=1}^{N}$ with $N = 60.000$ images of handwritten digits and $\mathbf{x}^{(t)} \in \{0,1\}^n$, where $n = 784 = 28x28$ pixels, we wish to approximate the distribution $p_{\text{data}}(\mathbf{x})$ where the digits come from.
The dataset that we will be using is the MNIST dataset [14], which contains handwritten digits. The images are in grayscale and have dimension $28x28$. It consists of a training set of 60.000 examples and a test set of 10.000. In the original dataset the pixels have discrete intensities ranging from 0 to 255. An example of this dataset would be the images in 3. We will see how, even in this simple scenario, classical statistical methods fail to model the distribution. To see that, let us try with a histogram, the easiest way to model a distribution.

Histogram

This is the first approach one might come up with. It models the probabilities by counting the times that an event (image generated) occurs in our dataset. If we have images of dimension $28x28$ with binary pixels, there are $2^{784} \approx 10^{236}$ images (the probability of each image) to estimate. Using all the images of our dataset:

$$p_i = \frac{|\text{image}_i|}{|\text{images}|}$$

So the problem is twofold: first, we do not have enough space to store all these parameters. Second, if we had the space, given a training set of 60.000 images we would have one sample in each p_i, which means we would just be memorizing the whole thing rather than understanding what a digit 1 is, and giving higher probability to them. Our model would look something like this:

$$p_i = \frac{1}{N} \text{ or } p_i = 0$$

Notice that if the image is not in the dataset, the probability will be 0 even if it differs only in one pixel from a dataset image. This is clearly not what we were expecting to get. We want a model that is able to generalize, to learn the underlying patterns of data and **infer in the unseen data correctly**. Specifically, we want to be able to show him a new number and obtain its real density as well as sample new images from the modelled distribution.

Solution

The problem with the histogram is that each image influences only one parameter and there are as many parameters as images. This means that the fact that we are seeing a 1 only affects the specific 1 we are seeing. The 1 that looks identical with just one different pixel is not being "affected". Ideally, what we want is that the model learns what a 1 really is. The problem is that a histogram has no capacity whatsoever to generalize from what it is seeing in the training data because it is just storing the information that it has been shown.
To solve this problem we will try to somehow approximate the distribution with a parameterized family of probability distributions $p_\theta(\mathbf{x}) \approx p_{\text{data}}(\mathbf{x})$ over our data using neural networks. This is a very good at approximation as it can be seen in [16]. Now, the challenges are:

- How to define these functions $p_\theta(x)$ to effectively represent complex joint distributions?
- Define a distance so that we can optimize.
- Computational challenges due to high dimensional data.

To solve these problems we will use 4 different sets of algorithms: Autoregressive, Flow-based, VAEs, and GANs.

1.3 Goals of the work

There are several goals in this work:

1. Learn about deep learning and its applications in high-dimensional distribution modelling.
2. Review and compare the 4 most common approaches in the field of image generation.
3. Learn to program with PyTorch and develop deep learning projects.
4. Be used as an understandable and deep introduction to the fields of deep learning and generative models.

1.4 Notation

Some important notation to keep in mind:

- Capital letters ($A, B, ...$) for matrices.
- Bold for variables ($\mathbf{x}, \mathbf{h}, \mathbf{X}, ...$) if these are vectors or matrices. For parameters we will not use this because it can easily be inferred from equations, and it is not as relevant.
- $\mathbf{X} = \left[\mathbf{x}^{(1)}, ..., \mathbf{x}^{(N)}\right]$ is our dataset.
- $\mathbf{x} = [x_1, ..., x_n]$ will be the data point (image in our case).
- As a typical convention vectors (w,$\mathbf{x}$,...) are vertical. They will only be horizontal if we transpose them (w^t, $\mathbf{x}^t$,...).
- The element-wise multiplication will be $\odot$.
- The term $\mathbb{E}_x$ will mean that the expectation is with respect the the distribution of $\mathbf{x}$.
- The subindexes will never be bold, even if it is a multidimensional variable we will still use $p_x(\mathbf{x})$.

2. Background: Deep Learning and Machine Learning basics

In order to understand this work it is essential to tackle several concepts first. These will be the building blocks for deep generative models.

2.1 Introduction

The first question we should ask ourselves is: **"what is learning?"**. Learning can be understood as *"the acquisition of knowledge or skills through study, experience, or being taught"*. Different examples of skills may include:

- Identify cats.
- Diagnose skin cancer.
- Draw handwritten digits (or faces). (Ours!)
- Drive.
- Identify hate speech in texts.
- Compose music.

All these skills are learned through study, experience, or teaching. What they have in common is that they have already been tackled by AI with significant successes. We will now go through the landscape of machine learning tasks. The first categories we find are **supervised learning** - where we learn by being taught -, **unsupervised learning** - where we learn from data in the wild without indications -, and **reinforcement learning** - where we learn by using rewards if our actions are correct. Within supervised learning we find the tasks of **classification** and **regression**. In the case of unsupervised learning there is **clustering**, **dimensionality reduction** (PCA), and **generative models**.
The next step should then be to understand the relationship between machine learning, deep learning, and AI. Firstly, AI encompasses any program that is able to reason like a human. Inside this category we can find the field of machine learning, which are algorithms that learn specific skills form data. Within the machine learning tools we encounter deep learning, a set of algorithms that try to mimic real neural networks. We will first look at a single artificial neuron and then see how we can use them to build complex models.

2.2 Models: Artificial Neural Networks

This method is inspired in real neurons, where there are some inputs, some internal computations, and then outputs 4. This is exactly what an **artificial neuron** (logistic regression or perceptron) does.

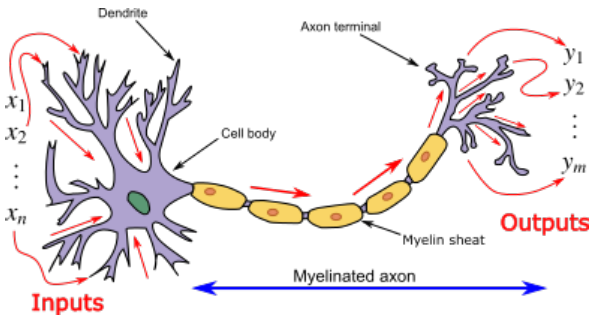

Figure 4: Inspiration for artificial neurons [22].

Artificial Neuron

An artificial neuron takes some inputs, performs some computations, and yields an output. The computations are the following ones:

$$\hat{y} = g(\sum_{i=1}^{n} w_i x_i + b)$$

The g function (**activation function**) is an assumption we make. Typical choices for the activation function g are the following:

$$\text{Sigmoid: } g(x) = \frac{1}{1+e^{-x}} \in (0,1)$$

$$\text{ReLU: } g(x) = \max(0,x) = x^{+} \in [0,+\infty)$$

A lot of times the choice will depend on how we want our output to be interpreted. For instance, if we want it to be understood as a probability, then we should use sigmoid because the output will be between 0 and 1. The typical way to visualize this structure can be seen in figure 5.

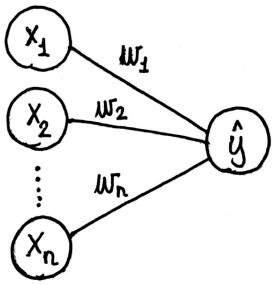

Figure 5: In the general case $\hat{y} = g(w^t \mathbf{x} + b)$.

The intuitive idea is that an artificial neuron learns a hyperplane $\sum_{i=1}^{n} w_i x_i + b = 0$. Then, if the data point $\mathbf{x}$ is in the hyperplane, we will have that $\hat{y} = g(0)$. If we use the sigmoid function, this will mean that the neuron is medium activated ($\hat{y} = 0.5$), and if we have the ReLU[2] function we will have null activation ($\hat{y} = 0$). In this latter case, the further we are from the hyperplane in the positive

[2]Stands for Rectified Linear Unit.

direction[3] the more activated the neuron. In the sigmoid case things are similar. If we get further in the positive direction the neuron gets more activated, whereas in the negative direction it loses signal.

To see the last statement in the sigmoid case, let us define

$$z = w_1 x_1 + \cdots + w_n x_n + b$$

$$\hat{y} = \frac{1}{1 + e^{-z}}$$

If we impose $z = 0$ we get $\hat{y} = 1/2$, which comes to say that the hyperplane $z = 0$ is the boundary of halfway activation. If $z > 0$ then we will have more signal, else ($z < 0$) we will have less. This can be visualized in figure 6.

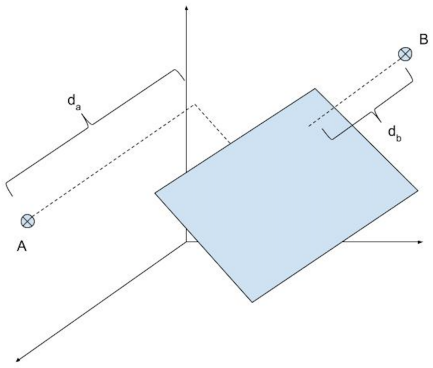

Figure 6: Hyperplane $z = 0$ that separates the data space into 2 regions, signals higher than 0.5 and lower. The distances to the hyperplane are the ones defining the signalling.

Fully Connected Neural Networks

In this setting we will basically stack together some neurons using layers. The architecture can be visualized in figure 7. In that case we only have one hidden layer, but there could be as many as we would like to. The notation will be very important to understand the equations in this thesis. We will use $W^{(i)}$ to denote the weights used to go from layer $i - 1$ to layer i. Also, keep in mind that normally we will use the same activation function for all the neurons of a given layer, and we will also have the bias $b_j^{(i)}$, where j indicates the position of the neuron in the layer and i the layer we are in. Clearly, with all this we will have a parametric model which can be written as

$$\hat{\mathbf{y}} = f(\mathbf{x}; \theta)$$

where θ will be used as the whole set of parameters and where the output can be a vector.

[3] This means the direction in which the normal vector is pointing towards, such that in that direction the weighted sum gets higher.

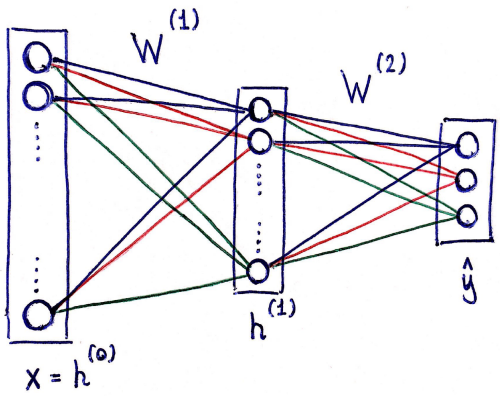

Figure 7: Fully connected neural network.

Other architectures

Using these as the building blocks there are also other architectures that are being used. The most typical ones are **convolutional neural networks** (CNNs) and **recurrent neural networks** (RNNs). Once the building blocks are clear both are very easy to understand. The former ones are used for images because the architecture is tailored for these kinds of applications. The basic idea is that they use filters (a small matrix of weights) and slide them through the whole image such that each filter learns to capture different shapes. If we wanted to improve the performance of our algorithms we could try to use CNNs in our architectures, but since the goal of this work was to introduce and explore deep learning in generative models we have kept it simple by ommitting the use of CNNs. In real world models the architectures are extremely complex, stacking all these architectures together. A typical model for images is to use some convolutional layers at the beginning and end with some fully connected ones (as well as **pooling layers**, **batch normalization** [9], etc.).

2.3 Loss function, optimization, and back-propagation

The first thing we should do if we want to learn something using Artificial Neural Networks is to define a **loss function**. A loss function is basically one which tells you how far you are from the desired outcome. This means that the higher the loss function, the worse your models are performing. In supervised learning for example we normally have our outputs $\hat{\mathbf{y}}$ and the real targets $\mathbf{y}$. Then, the function will compute some distance between those so that we have a value to optimize. In our case we don't have a target (unsupervised learning) so we could see our loss function as:

$$\mathcal{L}(f(\mathbf{x};\theta))$$

If we are using the whole dataset we will use the average

$$\frac{1}{N}\sum_{k=1}^{N}\mathcal{L}(f(\mathbf{x}^{(k)};\theta))$$

The way to define our loss in generative models will be based in principles related to **maximum likelihood**. Now, we will need a good algorithm to optimize (minimize in this case) the function.

We want it to be as low as possible so that we are close to the desired output.
In order to find the minimum of this function we could take the derivatives and equate them to zero. However, this has no closed form and this is why we must use an optimization algorithm. What an optimization algorithm does is basically take the function we are trying to minimize and roll downhill. The inevitable question now is how to find out the downhill direction. Easy, we just have to compute the gradient at the position we are in. The gradient is the direction of steepest ascent, which means we should take negative gradient and roll down a bit in that direction. This is the essence of all the optimization algorithms that we will see now. Let us introduce now different ways to approach this problem.

Gradient descent

Gradient descent is the classical method for optimization. In it we take the whole dataset (N data points), compute the loss function, and take its gradient. Then, we move in the opposite direction with a given rate (the **learning rate** μ):

$$\theta_{\text{new}} = \theta_{\text{old}} - \mu \nabla_\theta \left(\frac{1}{N} \sum_{k=1}^{N} \mathcal{L}(f(\mathbf{x}^{(k)}; \theta_{\text{old}})) \right)$$

Stochastic gradient descent (SGD)

Now, rather than computing the whole loss function we only evaluate it in a simple example. The way to update the weights would be, for every data point $\mathbf{x}^{(k)}$:

$$\theta_{\text{new}} = \theta_{\text{old}} - \mu \nabla_\theta \left(\mathcal{L}(f(\mathbf{x}^{(k)}; \theta_{\text{old}})) \right)$$

The advantages of this method is that for large datasets we do not have to wait to compute the function in the complete dataset, which can be very expensive computationally, and instead we can break the task into unit steps. Also, since this will not lead us straight towards the minimum as you will see in a later visualization 8, it can be very useful to avoid local minima.

Minibatch gradient descent

Now, we will do something in between the methods you have seen above. We will split our training data in a bunch of batches of datapoints. Once we have this, we will do the same as in the former methods. The direction we get will be more reliable than the SGD and at the same time quicker than gradient descent. This is a tradeoff we will always face. Find a great visualization of all these methods in 8.

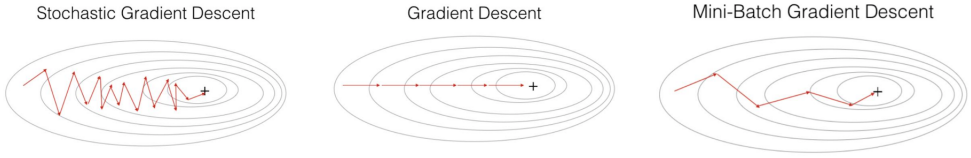

Figure 8: Visualization of the optimization algorithms [15].

Momentum

Momentum is something we can add in our optimization algorithm which basically saves all the past gradients. Adding it in our algorithms helps them converge faster. We will initialize a velocity parameter v, and at each step update it doing

$$v_{\text{new}} = \alpha v_{\text{old}} - \mu \nabla_\theta \left(\sum_{k=1}^{b} \mathcal{L}(f(\mathbf{x}^{(k)})) \right)$$

where b is the batch size. And now:

$$\theta_{\text{new}} = \theta_{\text{old}} + v$$

Clearly, the larger α is with respect to μ, the more previous gradients affect the current direction.

RMSProp: Root Mean Square Propagation

This algorithm incorporates an adaptative learning rate. The way it does this is by keeping an exponentially decaying average of the square of the gradients.

$$\mathbf{g} = \frac{1}{b} \nabla_\theta \left(\sum_{k=1}^{b} \mathcal{L}(f(\mathbf{x}^{(k)}; \theta)) \right)$$

$$\mathbf{r} = \rho \mathbf{r} + (1 - \rho) \mathbf{g} \odot \mathbf{g}$$

$$\Delta\theta = -\frac{\mu}{\sqrt{\delta + \mathbf{r}}} \odot \mathbf{g}$$

$$\theta = \theta + \Delta\theta$$

This helps in the sense that for the directions in which the last gradients have been very large it reduces the learning rate, but for the directions that have had little change it forces them to move faster. Notice that δ is just a small value to prevent dividing by zero. Also, we are incorporating a new hyperparameter[4] ρ. This one is called the **decay rate** and is basically how fast we forget past information. Empirically, this has been shown to be one of the best algorithms for deep learning.

Adam: Adaptative Moments Estimation [12]

In this final algorithm we will be using first and second order estimates.

$$\mathbf{g} = \frac{1}{b} \nabla_\theta \left(\sum_{k=1}^{b} \mathcal{L}(f(\mathbf{x}^{(k)}; \theta)) \right)$$

$$t = t + 1$$

$$\mathbf{s} = \rho_1 \mathbf{r} + (1 - \rho_1) \mathbf{g}$$

$$\mathbf{r} = \rho_2 \mathbf{r} + (1 - \rho_2) \mathbf{g} \odot \mathbf{g}$$

$$\Delta\theta = -\mu \frac{s}{\sqrt{r} + \delta}$$

[4] A hyperparameter is a parameter that we set and it is not learned.

where **s** is the first order estimate and **r** the second order. The interpretation here is that the first moment is basically taking the role of momentum and the second one is helping in adapting the learning rate. Another fair interpretation is that we are basically normalizing the steps with the variance of the gradients. In reality there is an intermediate step which is used to correct some bias but it has little effect in the long run and it will not help us understand what the algorithm is doing. Adam will be the algorithm that we will be using in most of the models.

Note: Something worth mentioning is that the initialization of the weights can be very important in training. Empirically, sampling them in the following way has been shown to be a good practice:

$$\text{std} = \frac{1}{\text{layer size}}$$
$$w \sim U[-\text{std}, \text{std}]$$

This is how PyTorch (the programming language used) automatically initializes the weights in fully connected layers.

Back-propagation

Now, how do we take derivatives? It turns out we can do it analytically. Recall that we have $\mathcal{L}(f(\mathbf{x}^{(k)}; \theta))$ for each data point. Basically we would like to see how to find the gradient with respect to all the weights so that we can update them. First of all the loss function has to be differentiable with respect to $\hat{y}_i = f_i(\mathbf{x}; \theta)$, the i-th components of the function. That way we would have

$$\frac{\partial L}{\partial \hat{y}_i}$$

where $L = \frac{1}{b}\sum_{k=1}^{b} \mathcal{L}(f(\mathbf{x}^{(k)}; \theta))$. Now, we have that $\hat{y}_i = g(z_i^{(l)})$[5], and since the activation function is differentiable and assuming we know the derivative we can do

$$\frac{\partial \hat{y}_i}{\partial z_i^{(l)}}$$

Finally, since

$$z_i^{(l)} = \sum_{j=1}^{n} w_j^{i(l)} h_j^{(l-1)} + b^{i(l)}$$

we would have that

$$\frac{\partial z_i^{(l)}}{\partial w_j^{i(l)}} = h_j^{(l-1)}$$

The notation $w_j^{i(l)}$ means that it is the weight that goes from poistion j to position i in layer l (in the matrix $W^{(l)}$ it is row i and column j). Following this same process we can go all the way backwards until we have all the gradients. This is how the library PyTorch we are using computes them. Once we have the gradients for each step, we can use the chain rule:

$$\frac{\partial L}{\partial w_j^{i(l)}} = \frac{\partial L}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial z_i^{(l)}} \frac{\partial z_i^{(l)}}{\partial w_j^{i(l)}}$$

[5]Recall that g is the activation function an $z_i^{(l)}$ is the result of the weighted sum in the i-th neuron before applying the activation in layer l.

This gets more complex as we go backwards into the network but the idea is exactly the same.

2.4 Training, Validation, and Test sets

Now, how do we see whether our model is actually good or not?
The first thing we must check is that our loss function is decreasing. This indicates that our model is learning/improving under the metric we set with the data that we are using. However, this can be misleading. Imagine that you were doing well with your data but only because you memorized what you have to do. For example, in trying to distinguish cats from dogs you memorized which pictures are cats and which ones are dogs, rather than trying to capture meaningful features that define what is a cat and what is a dog. In that case, if shown new data of cats and dogs you would have no clue on how to identify them correctly. This is a pervasive problem in deep learning and is called **overfitting**. In order to check that our algorithm is **generalizing** knowledge and not **memorizing**, we will divide the initial dataset $\mathbf{X}$ in two subsets $\mathbf{X}_{train}$ and $\mathbf{X}_{test}$ (in the supervised setting we would do the same with the outcome variable y). Since we have only learned with the **training set**, the algorithm has not seen the **test set**. We will use this set to evaluate how well our model does with unseen data, which basically tells us whether it is really learning or just memorizing. Sometimes we can even divide into 3 sets: validation, training, and test. In that case, we train with the training set and make some evaluations with the validation that helps us tune the hyperparameters. Once this is done we use the test set. This allows us to avoid overfitting to the test set as well.

2.5 Deep generative models taxonomy

There are different types of generative models. All of them are somehow maximizing the likelihood of the training set coming from our distribution. The difference is that some of them are learning the explicit distribution while others are only learning to sample from the distribution - they don't have access to the density function (**Implicit density**: GANs). Within the first category we can find models that have access to the distribution (**Tractable density**: autoregressive models and flow models) and others that only have access to an approximation (**Approximate density**: VAEs). Figure 9 shows how Ian Goodfellow describes it.

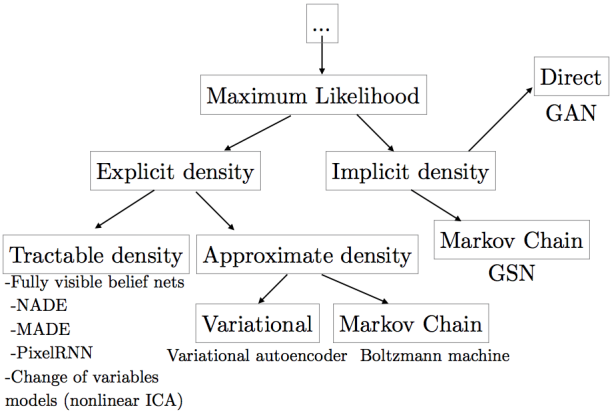

Figure 9: Taxonomy of deep generative models using maximum likelihood [6]

In this work we have used an autoregressive model (MADE), a flow-based model (NICE, change of variable models), a variational autoencoder, and a GAN.

3. Tractable density models: MADE and NICE

In tractable density models we have the analytical expression for p_θ and are able to evaluate it. These models are trained minimizing the KL divergence[6] between two distributions, or equivalently, maximizing the likelihood of the sample to occur. There are two questions to be answered now: how do we train and represent p_θ?
First we will see how to train our model using maximum likelihood. Then, we will see how to design p_θ. The requirements are the following ones:

- Compute $\log p_\theta(x)$ efficiently as well as its gradient.
- $\forall\theta, \sum_x p_\theta(x) = 1$ and $p_\theta(x) \geq 0, \forall x$

We will tackle this in two different ways. These are the most common ones in the tractable density paradigm: **Autoregressive models** and **Flow-based models**.

3.1 Training

In this section we will assume we have a given model $p_\theta(\mathbf{x})$ and want to find the best parameters $\tilde{\theta}$. In order to do so we have the following sample:

$$\mathbf{x}^{(1)}, ..., \mathbf{x}^{(N)} \sim p_{\text{data}(\mathbf{x})}$$

Where every point is a picture of the MNIST dataset.

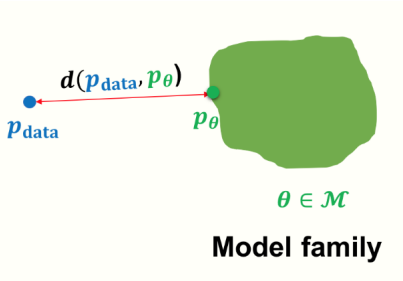

Figure 10: Clearly, we want to find the nearest point that minimizes the distances between models [20].

KL-divergence and Maximum Likelihood

Now, how do we measure closeness? This definition will allow us to find the parameters that minimize this distance. We will use KL-divergence, which is equivalent to maximum likelihood. KL-divergence measures the distance between two distributions:

$$D_{\text{KL}}(\, p || \, q) = \mathbb{E}_{x\sim p}\left[\frac{\log p(x)}{\log q(x)}\right]$$

[6] Explained bellow.

Clearly, if both distributions are the same $D_{\mathrm{KL}}(\,p||\,q) = 0$. Intuitively we can see that the distance will be low if in likely points of p, both distributions are similar. Notice as well that it is not a distance, since it is asymmetric. Also, we know that $D_{\mathrm{KL}}(\,p||\,q) \geq 0$, which will be very helpful in future applications. Finally, doing some simple maths and applying it to our model:

$$D_{\mathrm{KL}}(\,p_{\mathrm{data}}||\,p_\theta) = \mathbb{E}_{\mathbf{x}\sim p_{\mathrm{data}}}[\log p_{\mathrm{data}}(\mathbf{x})] - \mathbb{E}_{\mathbf{x}\sim p_{\mathrm{data}}}[\log p_\theta(\mathbf{x})]$$

And since the first term does not depend on θ, by minimizing the KL-divergence we are basically maximizing the expected log-likelihood. Using the a Monte Carlo estimator:

$$\mathbb{E}_{\mathbf{x}\sim p_{\mathrm{data}}}[\log p_\theta(\mathbf{x})] \approx \mathbb{E}_{\mathbf{x}\sim p_{\mathrm{data}}}[\log p_\theta(\mathbf{x})] = \frac{1}{N}\sum_{k=1}^{N} \log p_\theta(\mathbf{x}^{(k)})$$

We know that Monte Carlo estimators are unbiased, converge to the expected value, and the variance decreases as the number of samples increases.
In order to solve this non-convex optimization problem we used the Adam algorithm. Remember that we will not find the optimal point but we just want a "satisfying" solution. Understand that this will be the same process no matter which model we are using, that is why we explained first the training because it is common in all tractable density models.

3.2 Autoregressive models: MADE

Recall our MNIST case ($n = 784$):

$$\mathbf{x}^{(1)}, ..., \mathbf{x}^{(N)} \sim p_{\mathrm{data}}, \mathbf{x} \in \{0, 1\}^n$$

We want to find the explicit density function as well as be able to sample from it. We will try to tackle this problem using the following property:

$$p(\mathbf{x}) = \prod_{i=1}^{n} p(x_i|x_1, ..., x_{i-1})$$

This is a specific type of Bayes network which holds the autoregressive property, hence the name of the model. The autoregressive property states that the probability of the current state depends on the whole past states. The order we pick for the pixels will be an implicit assumption we are making on the given data. In our specific case of pictures we will typically order them left to right and top to bottom.

Note: The autoregressive property is much more natural in text or voice applications, where data is a real sequence.

The probabilistic model we are building for each pixel i is the following (Bernoulli distribution):

$$p_{\theta_i}(x_i|\mathbf{x}_{<i}) = \mathrm{Be}(f_i(\mathbf{x}_{<i}; \theta_i))$$

The fully expressive model would be a histogram, meaning that the function f would have a different parameter for every set of inputs indicating the probability. However, as we have seen in the introduction, it is not useful at all in our scenario. What we will do is to reduce the expressiveness of our function in order to have a feasible model.

Logistic Regression

We can first try with

$$f_i(x_1, ..., x_{i-1}) = \sigma(\alpha_0^i + \alpha_1^i x_1 + ... + \alpha_{i-1}^i x_{i-1}) \in [0, 1]$$

Since each f_i needs i parameters, then $|\theta| = \sum_{i=0}^{n} i = \frac{n(n+1)}{2} \approx O(n^2)$, which now gives us a more reasonable amount of parameters.

Single layer neural network

Going one step further we can try a single layer neural network, which would be much more expressive than a logistic regression.

$$\mathbf{h}_i = \sigma(A_i \mathbf{x}_{<i} + c_i)$$

$$f_i(x_1, ..., x_{i-1}) = \sigma((\alpha^i)^t \mathbf{h}_i + b_i)$$

With $\mathbf{h}_i \in \mathbb{R}^k$, we have now $O(n^2 k)$. Therefore we have increased both the parameters and the expressiveness of the model.

NADE: The Neural Autoregressive Density Estimatior [13]

Now, to reduce the number of parameters, we will tie the weights that come from every single input as done in figure 11. We will do the following:

$$\mathbf{h}_i = \sigma(W_{:,<i} \mathbf{x}_{<i} + c_i)$$

$$\hat{x}_i = p(x_i | \mathbf{x}_{<i}) = \sigma((\alpha_i)^t \mathbf{h}_i + b_i)$$

Notice that now the weight matrix W is shared among all $\mathbf{h}_i$. The number of parameters we now

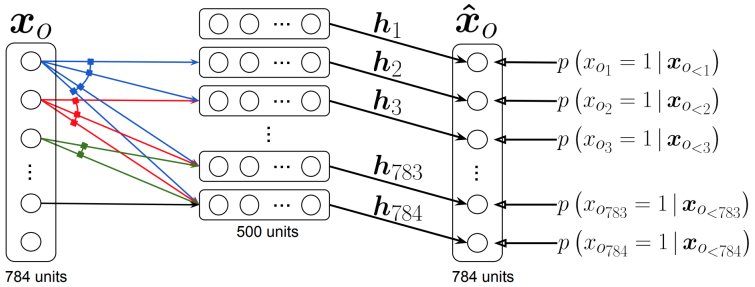

Figure 11: This picture from [21] shows the architecture that NADE follows.

have are $O(nd)$, which is an important improvement.

MADE: Masked Auto-encoder for Density Estimation [5]

Now, we want to join the idea behind NADE and an autoencoder structure for our network. Reducing dimensionality with an autoencoder helps learn meaningful features. In order to do so we will try to mask the necessary weights so that we can obtain this kind of structure with the autoregressive property. This has been our final choice for the experiments as the autoregressive model.
Recall the scenario:

- Given a dataset $\{\mathbf{x}^{(k)}\}_{k=1}^{N}$ with N images $\mathbf{x}^{(k)} \in \{0,1\}^{784}$ (28x28 binary pixels) we wish to approximate $p_{\text{data}}(\mathbf{x})$.

First of all let us see how to use autoencoders (AE) for this purpose. An AE has the following structure:

$$\mathbf{x} \to h(\mathbf{x}) = g(b + W^{(1)}\mathbf{x}) \to \hat{\mathbf{x}} = \sigma(c + W^{(2)}h(\mathbf{x})) \in [0,1]^{784}$$

In this case we have the input $\mathbf{x}$, the code $h(\mathbf{x})$ and the output $\hat{\mathbf{x}}$. This is why the first part is called the encoder and the second part the decoder. We will understand $\hat{\mathbf{x}}$ as the probability of each pixel being 1. This means that:

$$p(\{\text{d-th pixel prediction is correct}\}) = \hat{x}_d^{x_d}(1-\hat{x}_d)^{(1-x_d)}$$

Since the real d-th pixel is equal to x_d, which is either 0 or 1, if $x_d = 1$ then $p = \hat{x_d}$ and if $x_d = 0$ then $p = 1 - \hat{x}_d$. Overall, we can compute the probability of any image as:

$$p(\mathbf{x}) = \prod_{d=1}^{784} \hat{x}_d^{x_d}(1-\hat{x}_d)^{(1-x_d)}$$

Now, if we want the probability of the a whole batch (size b), we can compute it as $\prod_{k=1}^{b} p(\mathbf{x}^{(k)})$. However, as we usually do, we will train by minimizing the negative log-likelihood:

$$\mathcal{L}(\theta;\mathbf{x}) = \sum_{d=1}^{784} -x_d \log \hat{x_d} - (1-x_d)\log(1-\hat{x_d})$$

Which is basically the binary-cross-entropy formula.

Problem: we started explaining that an autoregressive model is one which satisfies $p(\mathbf{x}) = \prod_{d=1}^{n} p(x_d|\mathbf{x}_{<d})$. However, in this case every $\hat{x}_d$ seems to depend on the whole input. Therefore, we must prepare an architecture such that $\hat{x}_d$ depends only on the previous pixels. To do so we will use masks ensuring that this property holds true.

Masked Autoencoder

Since the output $\hat{x}_d$ must depend solely on the inputs $\mathbf{x}_{<d}$, we will zero out at least one connection for each of the unwanted paths. The masks will work as follows:

$$h(\mathbf{x}) = g(b + (W \odot M^W)\mathbf{x})$$

$$\hat{\mathbf{x}} = \sigma(c + (V \odot M^V)h(\mathbf{x}))$$

Remember that the code $h(\mathbf{x}) \in \mathbb{R}^K$ will be of lower dimension than $\mathbf{x}$. Now, how do we choose the right masks?

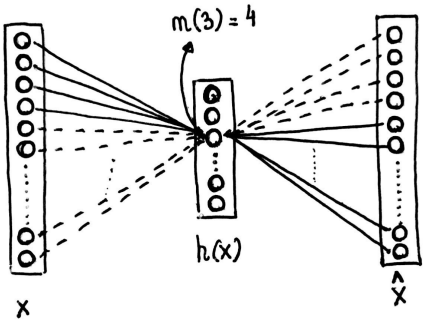

Figure 12: This picture shows the architecture that our MADE follows. The dotted lines signify connections that are being masked.

1. Assign to each of the K hidden units $m(k) \in \{1, ..., n-1\}$. This represents the inputs that will be connected. We will do it using a uniform discrete distribution.
2. Set $M^W_{k,d} = 1_{m(k) \geq d}$ (this means that the k-th hidden unit will only be connected to the $m(k)$ first inputs).
3. Set $M^V_{d,k} = 1_{d > m(k)}$ (this means that the d-th output will only be connected to the hidden units that are connected to at most the $d-1$ first inputs, fulfilling the autoregressive property).

This can be easily visualized in figure 12. There, we can see that by assigning a number to each hidden unit ($m(k) = d$) we can easily ensure that the autoregressive property holds.
To check the autoregressive property we can show that $M^{V,W} = M^V M^W$, which represents the final connectivity, is lower-diagonal. Doing some algebra we will show that $M^{V,W}_{d',d} = 0$ if $d' \leq d$:

$$M^{V,W}_{d',d} = \sum_{k=1}^{K} M^V_{d',k} M^W_{k,d} = \sum_{k=1}^{K} 1_{d' > m(k)} 1_{m(k) \geq d} = 0$$

Clearly $1_{d' > m(k)} 1_{m(k) \geq d} = 0$ if $d' \leq d$.

Note: Previous work found beneficial to add the following part to the algorithm:

$$\hat{\mathbf{x}} = \sigma(c + (V \odot M^V) h(\mathbf{x}) + (A \odot M^A)\mathbf{x})$$

Since the masks do not allow the d-th input to be connected with the d-th output (M^A will be lower diagonal), the identity function will not be learned and we can add the right term directly from the input. All this structure ensures that we can interpret each ouput as $\hat{\mathbf{x}}_d = p(x_d | \mathbf{x}_{<d})$ and learn the correct parameters to maximize this likelihood.

In our experiments we have used one hidden layer of dimension 500. The architecture is as drawn in figure 12. With all this we have the model $p_\theta(\mathbf{x})$ defined and we can use maximum likelihood as explained in the first section to find the best set of parameters.

3.3 Flow-based models: NICE

What else could we do to model this high dimensional distribution? In Flow-based models we attempt to approximate the whole distribution using invertible transformations.

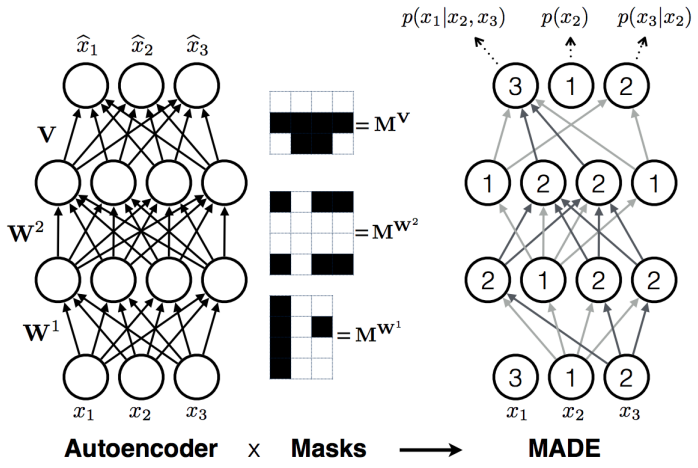

Figure 13: This picture from [5] shows how the masks work.

We will shift our perspective from looking at the PDF to looking at the CDF. Recall that sampling once we have the CDF is pretty straightforward:

$$z \sim U([0,1])$$

$$x = F_\theta^{-1}(z)$$

where F_θ is a CDF.
The key idea behind flows is to convert/map a simple (uniform, normal, etc.) distribution to the real data distribution. This is equivalent to training the PDF as we will see. The limitation is clear, since these transformations have to be invertible the spaces $\mathcal{Z}$ and $\mathcal{X}$ have to be of the same dimension, which means that there is no compression going on.

Flows in 1D

We will define **flow** as the differentiable and invertible function that maps $\mathcal{X}$, the data space,

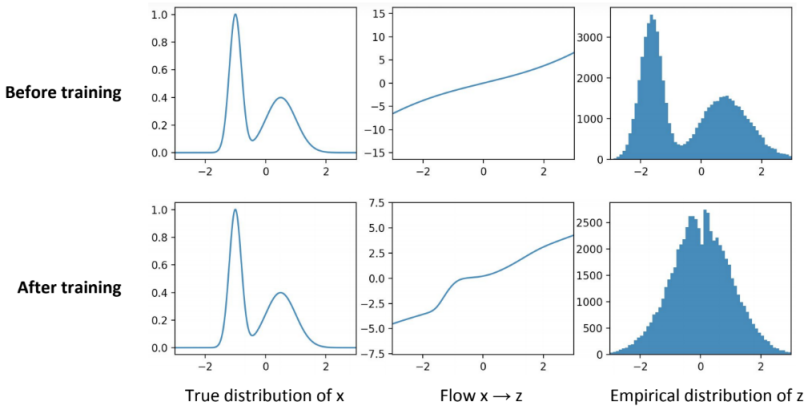

Figure 14: This picture from [**dul2019**] shows how flows work in one dimension.

to the noise space $\mathcal{Z}$.
We will train our model to turn the data distribution into the base distribution $p(z)$. In figure 14 we can see how the trained flow transforms a distribution in the data space into our base distribution (in this case a gaussian).
Now the problem is how to train these flows. We know we need p_θ in order to compute the likelihood, but we only have some transformation between two different spaces. Here is where the change of variables formula will be used.

Example and intuition behind the change of variable formula

We are going to explore an example of a flow. Imagine we have a prior distribution over $z \in [0, 1]$, $p(z) = 1$. Now, having defined the uniform distribution over the $\mathcal{Z}$ space, let us explore what happens if we map this space into another. For the sake of simplicity let us define $x = 2z$ as the mapping. Here, the flow would be $z = f(x) = \frac{x}{2}$. Figure 15 shows the mapping. We want to know which is

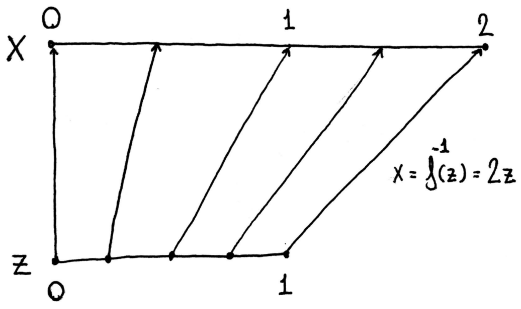

Figure 15: Here we can see the mapping from Z space to X.

the distribution for $x \in [0, 2]$ as a result of this function. We know intuitively what to expect from it, since it is uniform in $\mathcal{Z}$ it will also be in $\mathcal{X}$ because the mapping from one to another is equally distributed. Let us go through some interesting maths behind this:

$$p_x(x \in [x_0, x_0 + dx]) = p_z(z \in [f(x_0), f(x_0 + dx)])$$

Clearly, these intervals must have the same probability. We are just moving them from one space to another, but the likelihood of the event remains the same. Now, by some approximation:

$$p_x(x \in [x_0, x_0 + dx]) = \int_{x_0}^{x_0+dx} p_x(x)dx = F(x_0 + dx) - F(x_0) \approx F'(x_0)dx = p_x(x_0)dx$$

Where we used the Taylor polynomial for approximation - recall that $dx \to 0$. Now, doing the same thing twice:

$$p_z(z \in [f(x_0), f(x_0 + dx)]) = \int_{f(x_0)}^{f(x_0+dx)} p_z(z)dz = F(f(x_0 + dx)) - F(f(x_0))$$

$$\approx F'(f(x_0))[f(x_0 + dx) - f(x_0)] \approx p_z(f(x_0))f'(x_0)dx$$

Which finally shows the identity we were searching for, one that just by knowing the flow and the prior distribution we can compute the new distribution. In our case, we knew the mapping/flow $f(x)$ as well as the prior $p_z(z)$ and now we have a nice formula for $p_x(x)$. This formula is known as the change of variable formula and is stated as follows:

- **The change of variables formula:** Given an invertible and differential mapping (f_θ) from $\mathcal{X}$ to $\mathcal{Z}$, and a prior distribution over $\mathcal{Z}$ ($p_z(z)$), the resulting distribution on the $\mathcal{X}$ space is:

$$p_\theta(x) = p_z(f_\theta(x)) \left| \frac{\partial f_\theta(x)}{\partial x} \right|$$

Flows in higher dimensions

Since the change of variables formula also works for higher dimensonal spaces, using it we can train a neural network to learn the flow function. However, we will need two very strong requirements:

- The jacobian determinant must be easy to compute.
- The flow function must be invertible.

Going back to our goal of maximizing the log-likelihood we have the following expression for flows:

$$\text{argmin}_\theta \, \mathbb{E}_{\mathbf{x}}[-\log p_\theta(\mathbf{x})] = \text{argmin}_\theta \, \mathbb{E}_{\mathbf{x}}[-\log p_z(f_\theta(\mathbf{x})) - \log \det \left(\frac{\partial f_\theta(\mathbf{x})}{\partial \mathbf{x}} \right)]$$

NICE: Non-linear Independent Components Estimation [3]

We will build a flow $\mathbf{h} = f(\mathbf{x})$ such that the resulting distribution factorizes, which means that the components h_d are independent:

$$p_h(\mathbf{h}) = \prod_d p_{h_d}(h_d)$$

Considering this setting and using the change of variable formula:

$$p_x(\mathbf{x}) = p_h(f(\mathbf{x})) \left| \det \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}} \right| = \prod_d p_{h_d}(h_d) \left| \det \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}} \right|$$

To sample from this kind of model we will sample $\mathbf{h} \sim p_h(h)$ from the prior and then using the flow $\mathbf{x} = f^{-1}(\mathbf{h})$.
Now, we have to define the parametric family $\{p_\theta\}$.
Assuming a factorial distribution for our prior, the log-likelihood we want to maximize is the following:

$$\log p_x(\mathbf{x}) = \sum_{d=1}^{n} \log(p_{h_d}(f_d(\mathbf{x}))) + \log \left(\left| \det \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}} \right| \right)$$

Where $f(\mathbf{x}) = (f_1(\mathbf{x}), ..., f_n(\mathbf{x}))$. Notice several things:

- We are just learning an invertible transform of the dataset.
- The term $\det \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}}$ penalizes contraction and encourages expansion in the regions of high density (our data points).
- We will call f the encoder and f^{-1} the decoder.

Coupling layers, Rescaling, and Prior distribution

In this section we will explain the building blocks of the function we are going to build. These are **coupling layers**, **rescaling**, and the **prior distribution**.
First of all let us see what a coupling layer is. We will use a bijective transformations with triangular jacobian. Given $\mathbf{x} \in \mathcal{X}$, a partition I_1, I_2 of $[1, ..., n]$ with $|I_1| = d$, and a function m (the coupling function):

$$\mathbf{y}_{I_1} = \mathbf{x}_{I_1}$$
$$\mathbf{y}_{I_2} = g(\mathbf{x}_{I_2}; m(\mathbf{x}_{I_1}))$$

Here, g is the coupling law and it must be invertible with respect to the first variable. In our case we will use an additive coupling layer, which means $g(a; b) = a + b$. Now, the jacobian and the inverse are easily computed, where the former one is:

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{bmatrix} I_d & 0 \\ \frac{\partial \mathbf{y}_{I_2}}{\partial \mathbf{x}_{I_1}} & \frac{\partial \mathbf{y}_{I_2}}{\partial \mathbf{x}_{I_2}} \end{bmatrix}$$

Which means that $\det \frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \det \frac{\partial \mathbf{y}_{I_2}}{\partial \mathbf{x}_{I_2}}$. In the specific case of an additive coupling layer the determinant is 1.

Note 1: Since a coupling layer leaves part of the input unchanged we must exchange the role of each partition each time to add more expressiveness and complexity.
Note 2: Notice as well that in the case of additive layers the jacobian is 1, which means it preserves the volume. This is a very strict limitation for our model.

In order to solve the problem presented above, we will add a final layer to scale each one of the dimensions by doing $h_i = S_{ii} x_i$, allowing the learner to give more weight to some dimensions than others. This leads us to the final identity:

$$\log p_\theta(\mathbf{x}) = \sum_{i=1}^{n} [\log(p_{h_i}(f_i(\mathbf{x}; \theta))) + \log(|S_{ii}|)]$$

The S_{ii} values can be intuitively compared with the PCA eigenspectrum showing how much variation is present in each latent dimension. Since $x_i = \frac{h_i}{S_{ii}}$, the larger S_{ii} the less important the dimension i.

Finally, we will use the logistic distribution as our prior for each one of the dimensions. We do not use gaussians because the gradient of the logistic behaves better. In this case, the log-likelihood can be easily shown to be:

$$\log p_{h_d}(h_d) = -\log(1 + e^{h_d}) - \log(1 + e^{-h_d})$$

Final architecture

We will use 4 additive coupling layers with a final scaling layer:

$$\text{1st layer: } \mathbf{h}^{(1)}_{I_1} = \mathbf{x}_{I_1},\ \mathbf{h}^{(1)}_{I_2} = \mathbf{x}_{I_2} + m^{(1)}(\mathbf{x}_{I_1})$$

$$\text{2nd layer: } \mathbf{h}^{(2)}_{l_2} = \mathbf{h}^{(1)}_{l_2},\ \mathbf{h}^{(2)}_{l_1} = \mathbf{h}^{(1)}_{l_2} + m^{(2)}(\mathbf{h}^{(1)}_{l_2})$$

...

The third and fourth layers are the same permutations. Finally, the scaling layer:

$$\hat{\mathbf{x}} = \mathbf{h}^{(5)} = e^{s} \odot \mathbf{h}^{(4)}$$

We have defined 4 functions $m^{(1)}, ..., m^{(4)}$ that will be learned with fully connected neural networks. All of them will have the same structure, 5 hidden layers with 500 units.

3.4 Results

We have trained both models using maximum likelihood and 100 epochs each. As it can be seen in figures 16 and 17 both models converge without important overfitting. It is interesting to see how the likelihood in both models is very different. This is due to the construction of both models. In the autoregressive model we have learned a discrete model in which pixels were binary. However, in flows we have learned it as if our pixels were continuous. This is why both likelihoods are very different and cannot be compared. The last thing has been sampling from both models, which can be seen in figure 18. We will compare both algorithms in the last section.

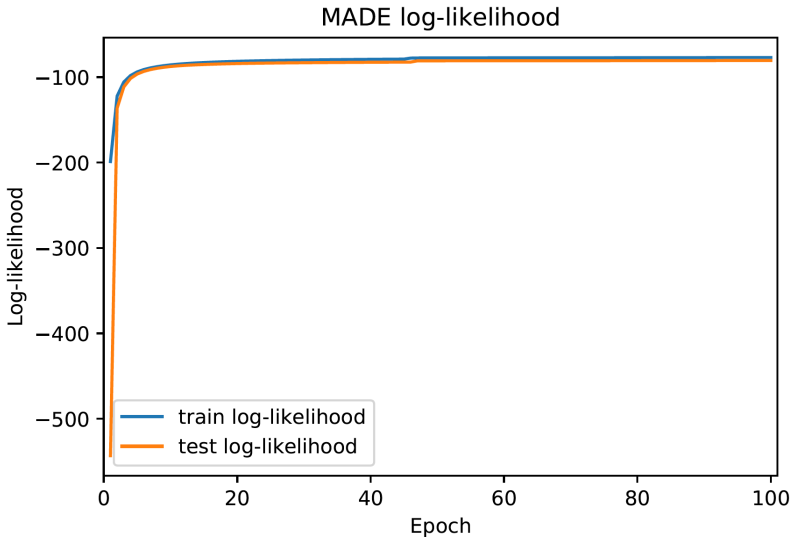

Figure 16: MADE losses.

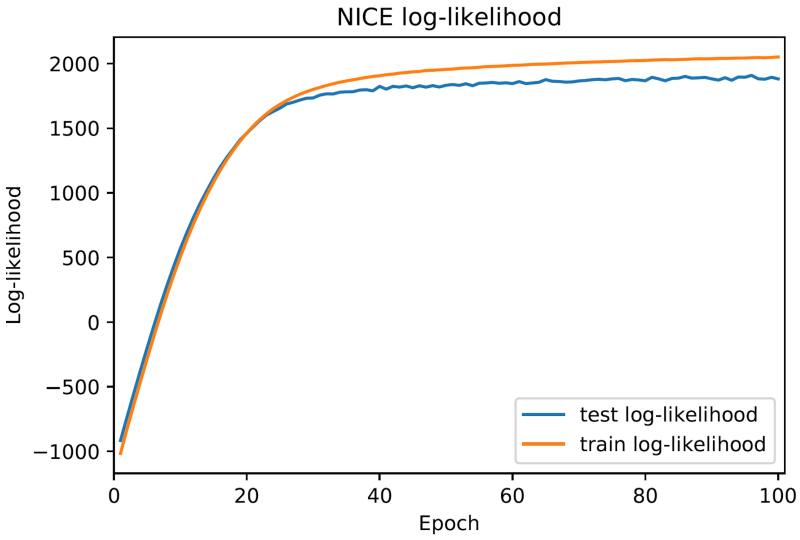

Figure 17: NICE losses.

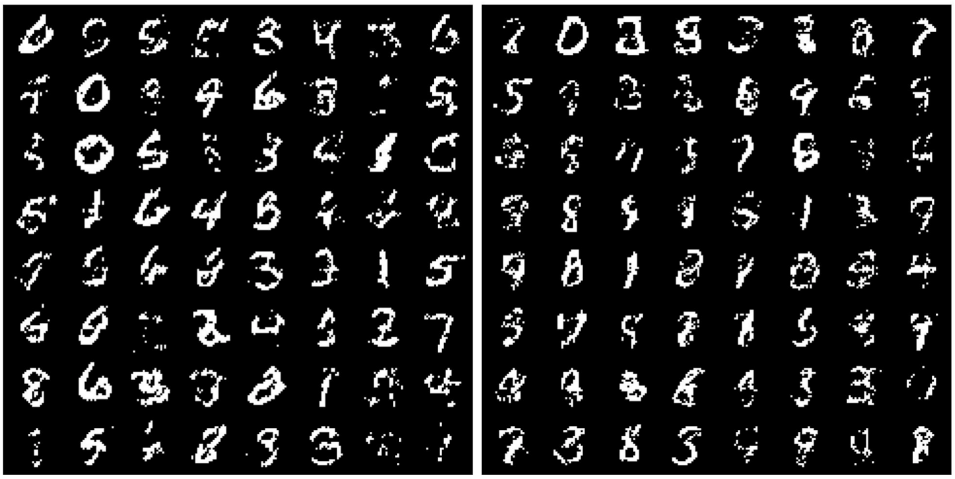

Figure 18: NICE in the left and MADE in the right.

4. Approximate density models: Variational Autoencoders (VAEs)

In this setting we assume that the data $\{\mathbf{x}^{(i)}\}_{i=1}^{N} \sim p_{\text{data}}(\mathbf{x})$ is generated from an unobserved latent variable $\mathcal{Z} = \mathbb{R}^2$. This hidden variable encodes the digit, the dimension of the number, the rotation, etc. You will see in figure 25 how this space will represent meaningful features.

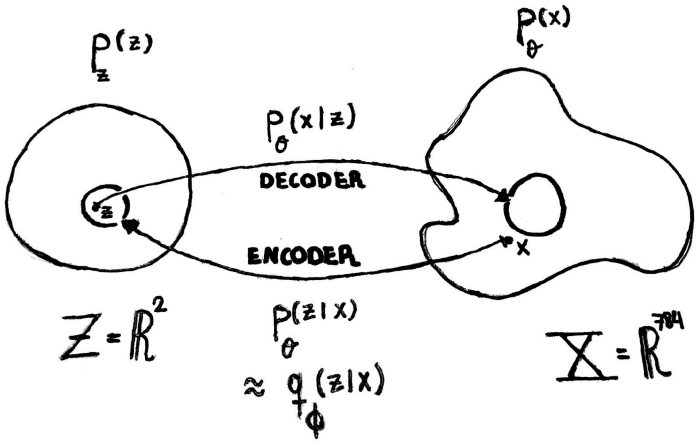

Figure 19: This is the model we are building. The circles within each space emphasize the fact that the image of the function (either the encoder or the decoder) is a probability distribution in the other space.

4.1 Model

Let us define first the following distributions:

- $p_z(\mathbf{z}) = \mathcal{N}(0, I)$ is the prior distribution on $\mathcal{Z}$.
- $p_{\text{data}}(\mathbf{x})$ is the real distribution of the images and $p_\theta(\mathbf{x})$ our approximation.
- $p_\theta(\mathbf{x}|\mathbf{z}) = \mathcal{N}(\boldsymbol{\mu}_x = f(\mathbf{z}; \boldsymbol{\theta}), \sigma^2 I)$ is a model we will learn (**decoder**), where $f_\theta : \mathbb{R}^2 \to \mathbb{R}^{784}$ is a neural network. Basically, given a code $\mathbf{z}$ we will output a distribution in the image space with mean $\boldsymbol{\mu}_x$. We will assume $\sigma^2 \approx 0$ to avoid noisy samples.

Having all this defined, it follows that:

- Implicitly, we are already defining the likelihood:

$$p_\theta(\mathbf{x}) = \int p_\theta(\mathbf{x}|\mathbf{z}) p_z(\mathbf{z}) d\mathbf{z}$$

- And the posterior (**encoder**):

$$p_\theta(\mathbf{z}|\mathbf{x}) = \frac{p_\theta(\mathbf{x}|\mathbf{z}) p_z(\mathbf{z})}{p_\theta(\mathbf{x})}$$

- As we will see, this posterior is intractable because p_θ is intractable. That is why we define an approximation $q_\phi(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\mu_z(\mathbf{x};\phi), \Sigma_z(\mathbf{x};\phi))$. Here, similarly, μ_z and Σ_z will be neural networks.

Keep in mind that all these are assumptions we are making in order to model the real distributions. The result will tell us if this were logical assumptions or not. We will attempt to model $p_\theta(\mathbf{x}|\mathbf{z})$ because it will be much easier than the whole $p_\theta(\mathbf{x})$, which will be approximated later on. Once we have the model, we will be able to generate new images using:

$$\mathbf{z} \sim p_z$$

$$\mathbf{x} \sim p_\theta(\mathbf{x}|\mathbf{z})$$

One can check how all this looks like in image 19.

4.2 Training

Problem:

As we have seen, the likelihood we would like to maximize under the training data is:

$$p_\theta(\mathbf{x}) = \int p_\theta(\mathbf{x}|\mathbf{z}) p_z(\mathbf{z}) dz$$

We could easily think that in order to approximate the integral we could use:

$$p_\theta(\mathbf{x}) = \frac{1}{N} \sum_{i=1}^{N} p_\theta(\mathbf{x}|\mathbf{z}_i), \text{ with } \mathbf{z}_i \sim p_z(\mathbf{z}) \tag{1}$$

However, in order for this to be accurate we would need a lot of samples. In practice this is an intractable problem, and this is why we will use variational inference. We will find a tractable lower bound of the likelihood and optimize it. Notice that given an image $\tilde{\mathbf{x}}$ there are a lot of $\mathbf{z}_i$'s that don't contribute to the likelihood whatsoever. If $p(\cdot|\mathbf{z}_i)$ is the distribution obtained from the code $\mathbf{z}_i$ given, but the image on the training $p(\tilde{\mathbf{x}}|\mathbf{z}_i) \approx 0$ is very unlikely, then it is not contributing to equation 1. It would be nice if we knew how to sample the "right way", meaning the $\mathbf{z}$'s that are more likely under the $\mathbf{x}$ we are seeing. Then, we could do the latter approximation without the need of many samples. This is why we introduced $q_\phi(\mathbf{z}|\mathbf{x})$, which approximates the intractable $p_\theta(\mathbf{z}|\mathbf{x})$.

Solution: variational inference

Now, how can we relate all this?

$$D_{\mathrm{KL}}\left(q_\phi(\mathbf{z}|\mathbf{x})||\, p_\theta(\mathbf{z}|\mathbf{x})\right) = \mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})}[\log q_\phi(\mathbf{z}|\mathbf{x}) - \log p_\theta(\mathbf{z}|\mathbf{x})]$$

$$= \mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})}[\log q_\phi(\mathbf{z}|\mathbf{x}) - \log p_\theta(\mathbf{x}|\mathbf{z}) - \log p_z(\mathbf{z})] + \log p_\theta(\mathbf{x})$$

Rearranging some terms and using that the KL divergence is non-negative:

$$\log p_\theta(\mathbf{x}) \geq \log p_\theta(\mathbf{x}) - D_{\mathrm{KL}}\left(q_\phi(\mathbf{z}|\mathbf{x})||\, p_\theta(\mathbf{z}|\mathbf{x})\right)$$

$$= \mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x}|\mathbf{z})] - D_{\mathrm{KL}}\left(q_\phi(\mathbf{z}|\mathbf{x})||\, p_z(\mathbf{z})\right) = \mathcal{L}(\theta, \phi; \mathbf{x})$$

It is very important to understand that this function is for a single image $\mathbf{x}$. We will call this term the variational lower bound (VLB) and we will define each term as:

- The **reconstruction loss**: $\mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x}|\mathbf{z})]$. This is reconstruction because we first sample an image $\mathbf{x}$, encode the image using q_ϕ, and go back to the image space with $p_\theta(\mathbf{x}|\mathbf{z})$. It is then evident that we want to maximize this term.
- The **regularization term**: $D_{\text{KL}}\left(\, q_\phi(\mathbf{z}|\mathbf{x})||\, p_z(\mathbf{z})\right)$. This term is the distance between the 2 distributions. It is called the regularization term because it is making sure that the new distribution behaves as similarly to the prior as possible (minimizing).

By maximizing the whole loss function we are both maximizing the likelihood as well as minimizing the distance between the two distributions ($q_\phi(\mathbf{z}|\mathbf{x})$, $p_\theta(\mathbf{z}|\mathbf{x})$). Hence, we are not only learning a model $p_\theta(\mathbf{x})$ but also approximating $p_\theta(\mathbf{z}|\mathbf{x})$ with $q_\phi(\mathbf{z}|\mathbf{x})$, which can give us further information about the latent space.

Loss function

Now, we can break down the optimization objective. First of all, the KL-divergence has a closed form because it is comparing two gaussians:

$$D_{\text{KL}}\left(\, q_\phi(\mathbf{z}|\mathbf{x})||\, p_z(\mathbf{z})\right) = D_{\text{KL}}\left(\, \mathcal{N}(\mu_z(\mathbf{x};\phi), \Sigma_z(\mathbf{x};\phi))||\, \mathcal{N}(0, I)\right)$$

$$= \frac{1}{2}[tr(\boldsymbol{\Sigma}_z) + \boldsymbol{\mu}_z^t\boldsymbol{\mu}_z - k - \log(\det(\boldsymbol{\Sigma}_z))]$$

The reconstruction term, $\mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x}|\mathbf{z})]$, will be approximated with just one example (stochastic approximation). How to compute $\log p_\theta(\mathbf{x}|\mathbf{z})$? Knowing that the dimension of the space is $n = 784$, $\boldsymbol{\Sigma}_x = \sigma^2 I_n$, and $\det \boldsymbol{\Sigma}_x = \sigma^{2n}$:

$$p_\theta(\mathbf{x}|\mathbf{z}) = \frac{1}{(2\pi)^{n/2}|\boldsymbol{\Sigma}_x|^{1/2}} e^{\frac{-1}{2}(\mathbf{x}-\boldsymbol{\mu}_x)^t\boldsymbol{\Sigma}_x^{-1}(\mathbf{x}-\boldsymbol{\mu}_x)} = \frac{1}{(2\pi)^{n/2}\sigma^n} e^{\frac{-1}{2\sigma^2}(\mathbf{x}-\boldsymbol{\mu}_x)^2}$$

Now, doing the log-probability we get that it is proportional to the mean square error, which is widely used in these kinds of tasks.

$$\text{Reconstruction loss } = \lambda(\mathbf{x} - \boldsymbol{\mu}_x)^2$$

Therefore, the loss function for the whole dataset or batch $\mathbf{X} = [\mathbf{x}^{(1)}, ..., \mathbf{x}^{(N)}]$ will be:

$$\mathcal{L}(\theta, \phi; \mathbf{X}) = \frac{1}{N}\sum_{i=1}^{N}\mathcal{L}(\theta, \phi; \mathbf{x}^{(i)})$$

$$\mathcal{L}(\theta, \phi; \mathbf{x}^{(i)}) = \log p_\theta(\mathbf{x}^{(i)}|\mathbf{z}) - D_{\text{KL}}\left(\, q_\phi(\mathbf{z}|\mathbf{x}^{(i)})\Big|\Big|\, p_z(\mathbf{z})\right)$$

This is the approximation we are making, which converges to the real gradient we aim for. In order to train our model we just have to use an optimization algorithm.

Gradients

Given the random nature of these mappings, it is unclear how we should be taking gradients. In figure 20 one can see how the losses are computed through the process. The problem is that there

are random processes which do not allow gradients to be taken. What we will do is known as the **reparameterization trick**. Instead of sampling from $\boldsymbol{\mu}_x$ and $\boldsymbol{\Sigma}_x$, we will do the following:

$$\mathbf{z} = \boldsymbol{\mu}_x + \boldsymbol{\Sigma}_x \epsilon, \text{ where } \epsilon \sim \mathcal{N}(0, I_n)$$

Other than that all are differentiable functions, which means we can take gradients without any problem.

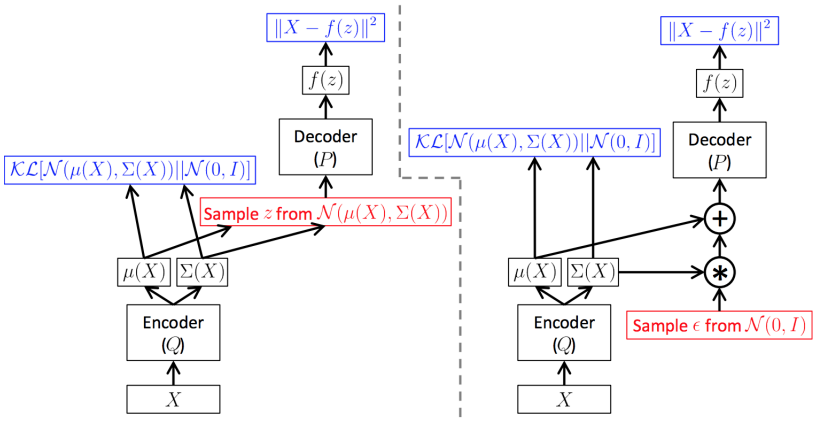

Figure 20: In this image we can see how the losses are computed in the process. One can clearly see that in order to take gradients from the reconstruction loss to the encoder q_ϕ we have to go through a random process (sampling $\mathbf{z}$). However, in order to be able to take gradients we can implement the reparameterization trick. The image can be found in [4].

4.3 Architecture

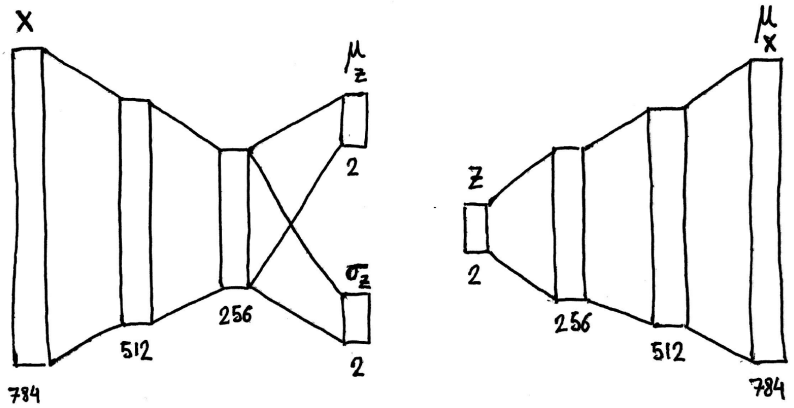

Figure 21: VAE architecture. All the layers are fully connected ones.

One can see the architecture used in figure 21. As you can see, we are using fully-connected layers both in the encoder and the decoder. Also, notice that the latent space is very low dimensional (2). This means that we are compressing a lot the initial space, learning a very meaningful representation.

4.4 Results

We ran the algorithm for 100 epochs and used a batch size of $N = 100$. As you can see in figure 22, we stopped at epoch 100 to avoid overfitting. Although the gap between the training and test performance was considerable, the test performance was still as good as in epoch 60. This is why as the final model we used the one at epoch 100. One can see the resulting images in figure 23.

Latent space interpolation and 2D visualization

In figure 24 we can see what happens if we interpolate two random points in the latent space and project the line between them. Since the prior is a normal, we expect that the line between the points must be very likely as well, and this is why the images are still digits. In this case we go from a 7 to a 9.
Another great visualization can be seen in figure 25. In it we are basically taking some images and using the encoder to know which point in $\mathbb{R}^2$ corresponds to each of them. Once this is done we can see how different numbers are distributed in the $\mathcal{Z}$ space. Notice that 7 and 9 are close in this space and this is why when doing the interpolation we did not go through any other numbers.

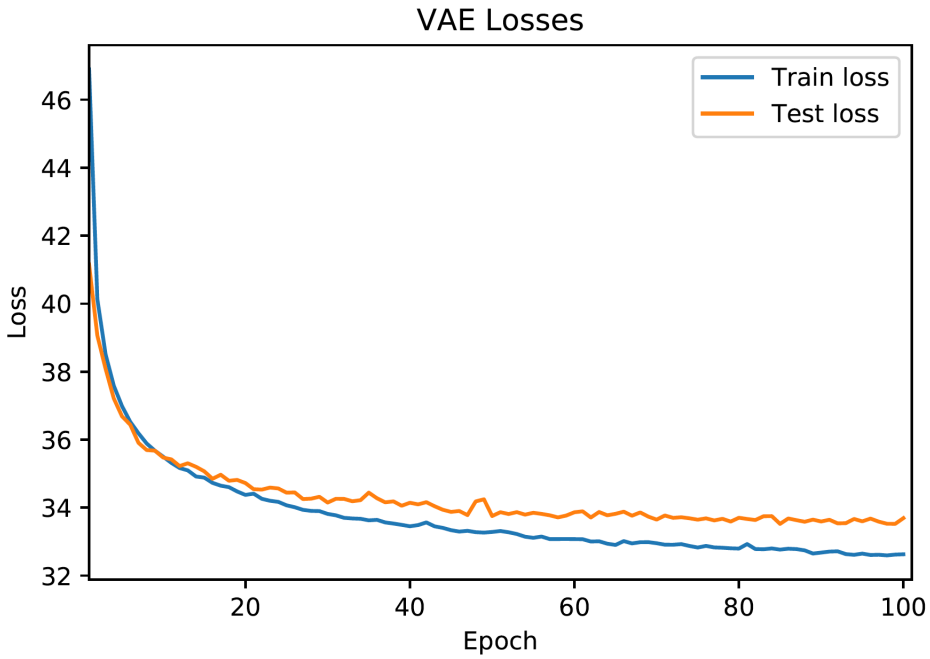

Figure 22: Losses for variational autoencoders.

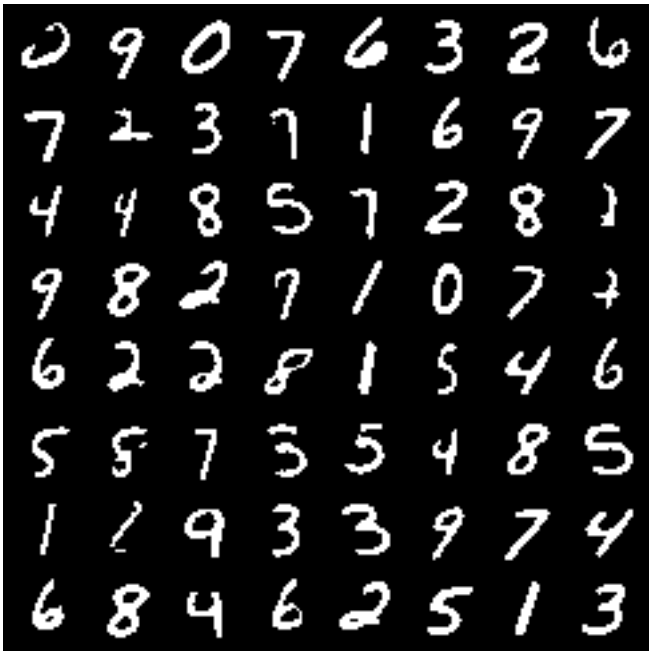

Figure 23: VAEs generated digits.

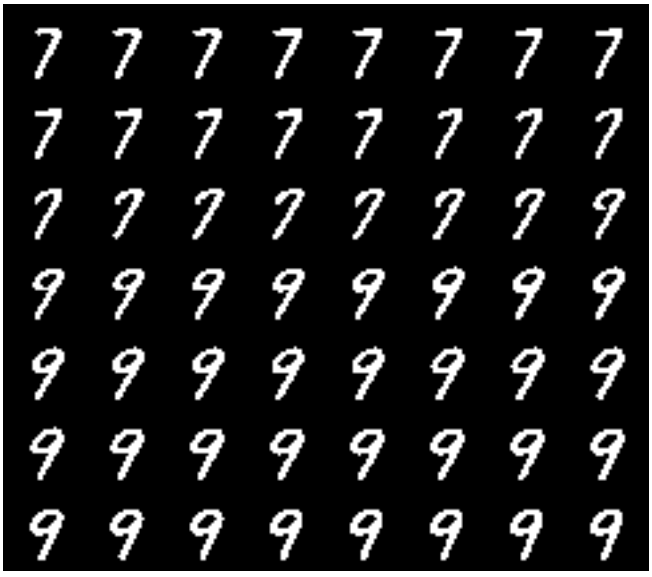

Figure 24: This is the interpolation of 2 random points in the latent space projected in the image space.

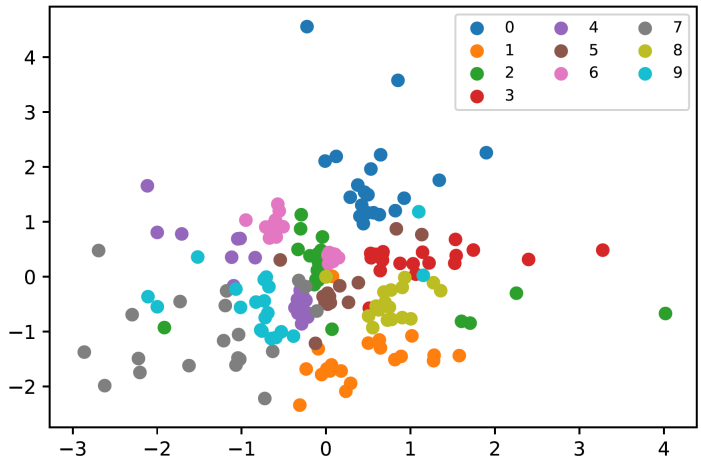

Figure 25: Here one can visualize the digits in the latent space. It is very interesting to see how similar digits are clustered next to each other (1 and 7 for instance).

5. Implicit density models: Generative Adversarial Networks (GANs)

So far we obtained good likelihood-based models, but what if we tried likelihood-free learning? In likelihood-based models we maximize the probability of observing the training set given a parametric probability function - we optimize the parameters of this function to fit the data. Once this is done, we have a tractable distribution where we can sample from. Keep in mind that a good generative model is one which has good sample quality and high likelihood. However, the highest likelihood does not necessarily mean the best sampling. We will now switch our approach by focusing on the sampling process - we will learn a sampler.

We will define an agent (G, generator) that generates new images, and another (D, discriminator) that discriminates whether an image is real or fake. These two agents will play against each other the following game several rounds. Each round will consist of the following steps.

1. G generates new images.
2. D will be given several images, both generated and real.
3. D will classify these images as real or fake.
4. G and D will learn from the errors they have made and improve their methods/systems.

Following this process, G and D will become stronger and stronger. Notice that if they end up being perfect, the generated images will be indistinguishable from the real ones and D will be unable to discern real from fake. Once this happens, G will be the sampler we are looking for.

5.1 Model: generator and discriminator agents

We will define these agents using neural networks (G and D are neural networks). The discriminator will learn a mapping from the image space to the $[0, 1]$ space.

$$\begin{aligned} D_{\theta_d} : \mathbb{R}^{784} &\longrightarrow [0,1] \\ \mathbf{x} &\longrightarrow D_{\theta_d}(\mathbf{x}) = p(\text{real}|\mathbf{x}) \end{aligned}$$

As you can see the discriminator depends on the weights θ_d of the neural network that defines the agent. From now on we will just use $D = D_{\theta_d}$. The generator will learn a mapping from a latent space $\mathcal{Z} = \mathbb{R}^{64}$ to the image space.

$$\begin{aligned} G_{\theta_g} : \mathcal{Z} &\longrightarrow \mathbb{R}^{784} \\ \mathbf{z} &\longrightarrow G_{\theta_g}(\mathbf{z}) = \textbf{fake image} \end{aligned}$$

Similarly, we will use $G = G_{\theta_g}$. We will define a prior distribution in $\mathcal{Z}$ (multivariate normal) where we will sample noise from. The sampling process goes as follows:

$$\mathbf{z} \sim p_z$$

$$\mathbf{x} = G(\mathbf{z})$$

Finally, the architecture used for the networks is very straightforward, you can check it in section 5.3. This is not the most optimal one, we could have used CNNs which give better results with images. The reason we chose not to is to keep it simple, the main goal of this document is to introduce deep generative models and compare their performance.

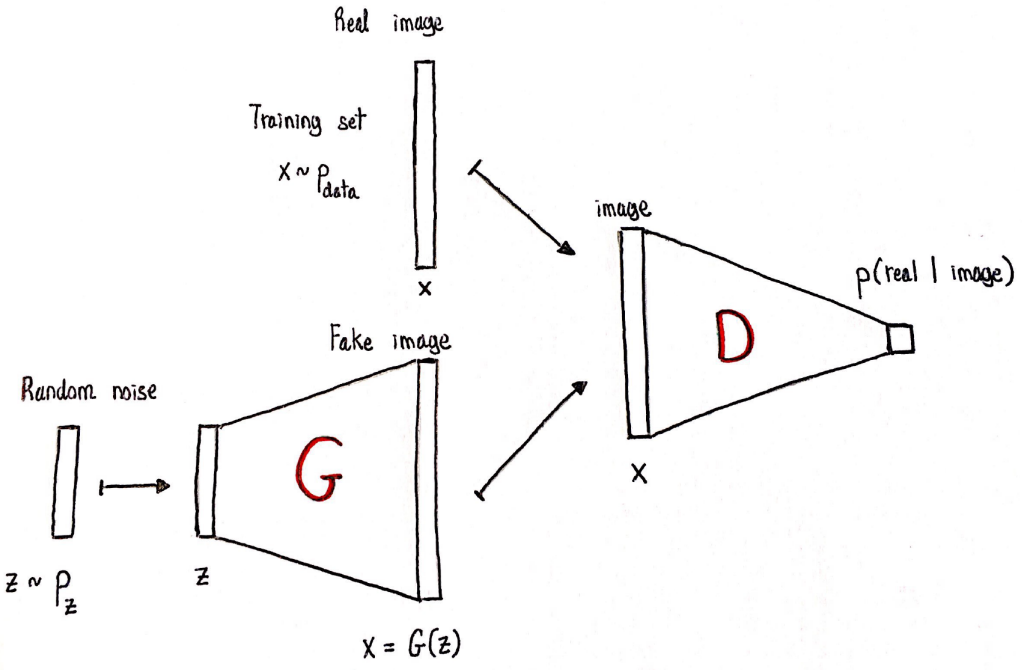

Figure 26: GANs structure.

5.2 Training

The game described before can be formally written as a minimax game:

$$\min_G \max_D V(D, G)$$

$$V(D, G) = \mathbb{E}_{x \sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))]$$

Now, let us see why this is the same as the game. The first term of the V function is 0 if the discriminator identifies that x is real ($D(x) = 1$) and $-\infty$ if it identifies it as fake ($D(x) = 0$). Hence, since the images come from the real distribution, the goal of D is to maximize it (classify real images as real). The second term is 0 if the discriminator finds out it is a fake sample and $-\infty$ if the generator fools the discriminator. Now again, D wants to maximize and G to minimize. Therefore, the goal of the generator in this game is to minimize $V(D, G)$ and the goal of the discriminator is to maximize it, as written in the formula. As seen in algorithm 1 the training will be done alternating one agent and the other. Notice a few things: firstly, we can use any optimization algorithm we want just by changing the step function. In this case, we have used the Adam algorithm [12], widely accepted in the deep learning community. Secondly, we are not optimizing the generator as done in the minimax

Algorithm 1 GANs training:

Initialization
N = batch size
for number of training iterations **do**
  for k steps **do**
    Sample minibatch $\{\mathbf{x}^{(1)}, ..., \mathbf{x}^{(N)}\}$ from p_{data}
    Sample minibatch $\{\mathbf{z}^{(1)}, ..., \mathbf{z}^{(N)}\}$ from the prior distribution p_z
    $----$ **TRAIN D** $----$
    $L_f = -\frac{1}{N}\sum_{i=1}^{N} \log(1 - D(G(\mathbf{z}^{(i)}))) \approx -\mathbb{E}_{p_z}[\log(1 - D(G(\mathbf{z})))]$
    $L_r = -\frac{1}{N}\sum_{i=1}^{N} \log D(\mathbf{x}^{(i)}) \approx -\mathbb{E}_{p_{data}}[\log D(\mathbf{x})]$
    $\text{loss} = L_f + L_r$
    $d\theta_d = \nabla_{\theta_d}(\text{loss})$
    $\theta_d = \text{step}(\theta_d, d\theta_d)$
    $----$ **TRAIN G** $----$
    $\text{loss} = -\frac{1}{N}\sum_{i=1}^{N} \log D(G(\mathbf{z}^{(i)})) \approx -\mathbb{E}_{p_z}[\log D(G(\mathbf{z}))]$
    $d\theta_g = \nabla_{\theta_g}(\text{loss})$
    $\theta_g = \text{step}(\theta_g, d\theta_g)$
  end for
end for

game. Here, instead of minimizing $\mathbb{E}_{p_z}[\log(1 - D(G(z)))]$ we are maximizing $\mathbb{E}_{p_z}[\log D(G(z))]$. This is because it is more stable in training according to the authors of [8].

5.3 Architecture

The discriminator:

$$\begin{aligned}
\text{1st layer: } \mathbf{a}^{(1)} &= \text{LeakyRelu}_{0.2}(W^{(1)}\mathbf{x} + b^{(1)}) \\
\text{2nd layer: } \mathbf{a}^{(2)} &= \text{LeakyRelu}_{0.2}(W^{(2)}\mathbf{a}^{(1)} + b^{(2)}) \\
\text{3rd layer: } D(\mathbf{x}) = \mathbf{a}^{(3)} &= \text{Sigmoid}(W^{(3)}\mathbf{a}^{(2)} + b^{(3)})
\end{aligned}$$

And the generator:

$$\begin{aligned}
\text{1st layer: } \mathbf{a}^{(1)} &= \text{Relu}(U^{(1)}\mathbf{z} + c^{(1)}) \\
\text{2nd layer: } \mathbf{a}^{(2)} &= \text{Relu}(U^{(2)}\mathbf{a}^{(1)} + c^{(2)}) \\
\text{3rd layer: } G(\mathbf{z}) = \mathbf{a}^{(3)} &= \text{Tanh}(U^{(3)}\mathbf{a}^{(2)} + c^{(3)})
\end{aligned}$$

The discriminator starts with the size of the image, the following two layers have the same hidden size (256) and the last one is a real number. For the generator it starts with the latent size (64), it maps it to the same hidden size as before and in the last layer it has the image size (784).

5.4 Results

We ran algorithm 1 for 300 epochs and with a batch size of $N = 32$.
One can see in figure 27 that up until the 100th epoch the generator is progressively fooling more and more the discriminator. However, at that point, it starts to decrease slowly and the discriminator is increasingly better able to distinguish real from fake. It is important to understand that this does not mean that the generator is not learning. It just means the discriminator is learning "more" than the generator. However, it is clear that the agent G has some limitations (very simple network) which might be the reason why it does not converge towards the perfect sampler.

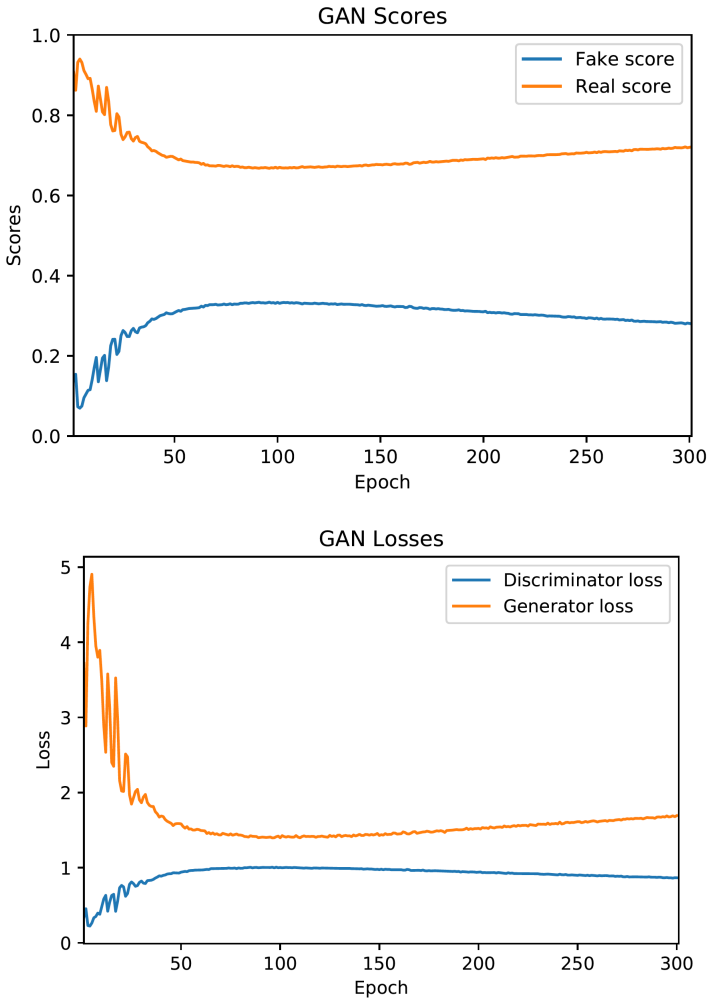

Figure 27: GANs results. Above one can see the scores. This is basically $\mathbb{E}_{p_z}[D(G(z))]$ for the fake score and the same but with real images for the real score. Below, one can see the losses. In both cases, G and D want to minimize. Originally, G wanted to minimize and D to maximize.

Notice as well that we are talking about 70% vs 30% accuracy, meaning the generator will fool the discriminator 3 times out of 10 in average. Now, the discriminator is able to identify patterns in the generated images which make them different from real ones, but are we humans able to? In figure 28 you can see samples in different stages of training. The interesting result we find in the next

section is that humans do not perceive any difference between generated and real. This might not be surprising for handwritten digits, but these algorithms are also used in face generation for instance, where this result is much more powerful.

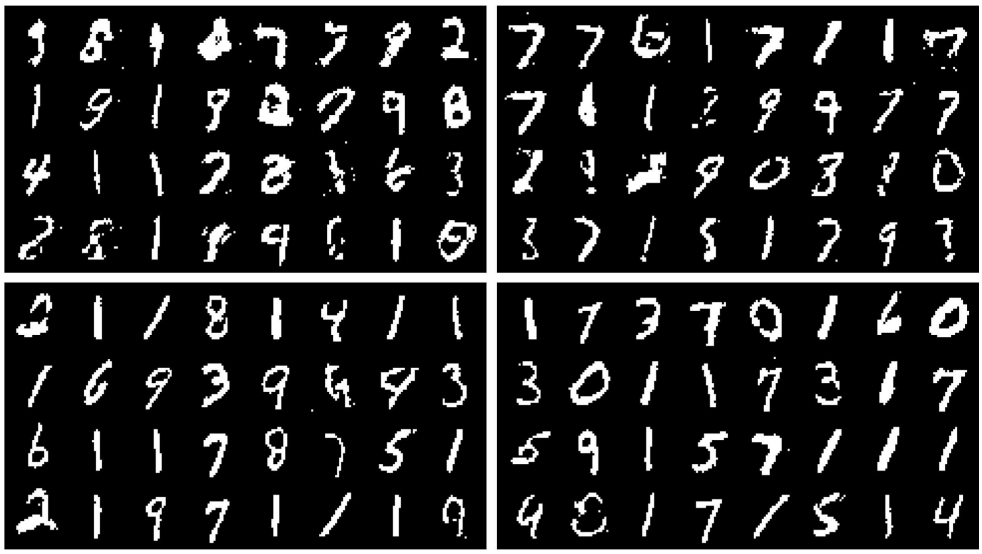

Figure 28: Generated images by GANs. The top left is epoch 50, top right 100, bottom left 200, and bottom right 300.

Although we see that the results are astonishing, we must keep in mind that a histogram would look perfect to human eyes as well, but it would not be accomplishing our goal of generalization - it would just be memorizing. In order to test whether we are overfitting or not, we will search for the nearest neighbor of each one of the generated images. This can be seen in figure 29, where there are clear differences between the generated image and its nearest neighbor. We cannot conclude we are not overfitting, but at least we see that we are learning something meaningful.

Figure 29: On the left are the generated images and on the right the corresponding nearest neighbor according to the euclidean distance.

The reality is that the mapping from the latent space $\mathcal{Z}$ to the image space $\mathbb{R}^{784}$ is continuous, which means that close images in one space will also be close in the other space. Now, since we are using a multivariate normal as the prior distribution (uni-modal), if we sample 2 vectors in the latent space and interpolate them, the resulting line will also be very likely, which suggests that the images should be handwritten digits as well. All this means that the image of this interpolation should be a continuous transformation between different hand-written digits. This is further explored in figure 30.

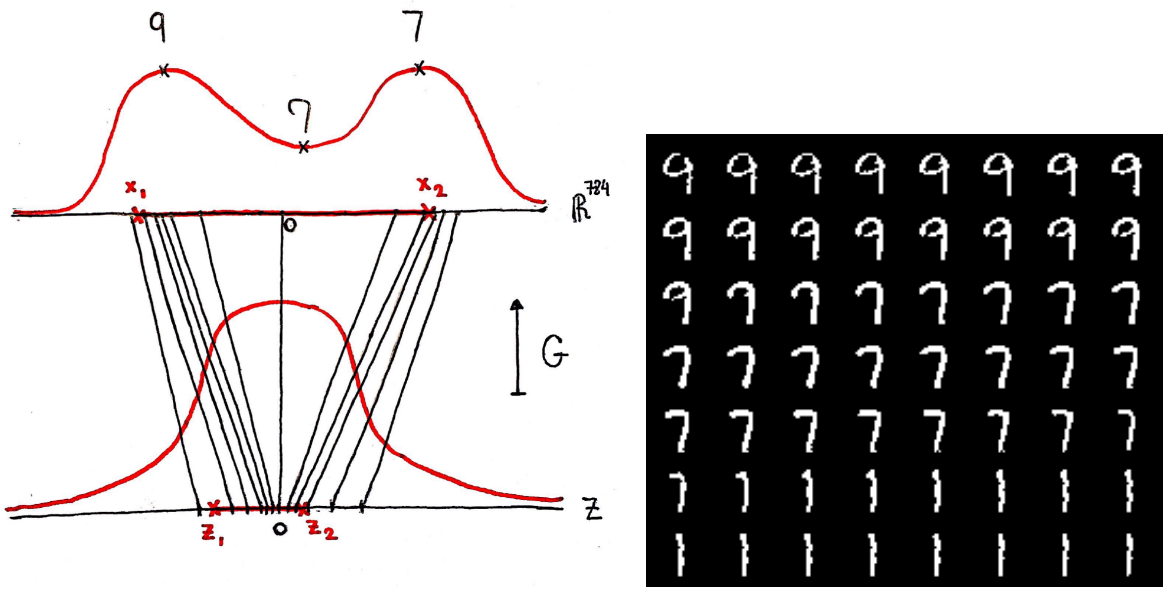

Figure 30: In the figure above one can see what is really going on through this transformation of the space using G. In this example, we see how we can go from a uni-modal distribution to a multi-modal one, which is basically what is going on in reality but in higher dimensions - where each "mountain" is a different number. Furthermore, one can see what happens if we sample 2 points in that latent space, interpolate them, and project the image. The figure below is the image of the interpolation in the latent space between 2 different points, sampled from the prior (multivariate normal distribution).

6. Conclusions, Comparisons and Future Work

Comparison between the 4 models

Comparing generative models is very challenging because there is no clear metric to do so. There are 2 main aspects we should analyse: likelihood estimates and sampling quality.

First, we want this models to give us the probability of an expected outcome. In this case, GANs are useless because they learn the likelihood implicitly. VAEs can be used but they are only approximating the likelihood function. Therefore, tractable density models are the best option because they are tailored for that. Keep in mind that we cannot compare the likelihoods because flows are trained in continuous data and autoregressives have been trained in discrete data. However, we can mention a few things we should consider when choosing between them:

- Autoregressive models might work better in applications where the data is **sequential**, such as speech or text.
- An advantage of flow-based models in comparison to autoregressives is that they are **invertible** and we are able to recuperate the exact information from the latent space, which could be useful in some applications.
- We have barely tested **computational costs** because the data we are using is very simple. However, two things are clear. First, flow's convergence was much slower. Second, sampling with autoregressive models is very expensive because they have to sample pixel by pixel, computing the whole model each time.
- Both models are very bad in sampling. This might be due to the fact that they are specifically trained to maximize the likelihood and not the sampling quality. In comparison, GANs are specifically trained to yield good samples and VAEs use the reconstruction loss which bias the model towards good samples as well.

In order to test the second aspect we will prepare a survey on image quality in which the aim is to see which algorithm produces more real numbers. The survey we did to test the models had 15 questions. In each question the user was asked to order the images from more "real" (1) to less (5). The term real here stands for the likelihood of it coming from a real distribution of handwritten digits. The options were one image from each of the 4 algorithms and one from the training set (Ground-truth). These are the results we obtained after 37 responses:

Alg.	Ord.
GAN	2.3
VAE	2.5
GT	2.5
MADE	3.4
NICE	3.5

It is very interesting to see that GANs and VAEs outperform the groundtruth in this little **turing test**. Clearly, VAEs and GANs outperform tractable density models by far in the sampling aspect. A few considerations between VAEs and GANs:

- In real applications GANs have lately outperform any other model in terms of sampling.
- An advantage of VAEs over GANs is that with VAEs we have an approximation of the likelihood, which might be worth using in some applications.

Note: A very important advantage in using VAEs is that they learn a low-dimensional representation which can help us find very interesting features that other algorithms might not. In GANs we also have a low dimensional representation but it is not as accessible as the one in VAEs.

To sum up, each algorithm has its strengths and weaknesses and we should be aware of them in order to make the best possible choice. Depending on the task we want to implement we will decide which one is best.

Future Work

There are several things we could do as a natural continuation of this work.

1. Try to build conditional models. These are the same ones we have built but with the ability of telling them which number you want to sample. The models in which this feature has been less explored are flows. There is some research being done such as [18], conducted by researchers at IRI as well.
2. Once the models have been tested and understood in the MNIST dataset, we could go on and compare them in very high dimensional spaces (the most natural continuation would be to apply it in better quality images). This would allow us to have a much more accurate comparison of each model in real examples.
3. We could also change the architectures of the networks being used and see how much our models could improve. As we have mentioned throughout the thesis, CNNs are the most suited architectures for images. These, combined with some final fully connected layers and other typical features could certainly improve performance.
4. As a final suggestion, we could also do some research and see where we could apply these generative models. As a personal preference I would start looking at medical image applications.

Conclusions

We will go through the goals we had for this work.

1. The first goal has been accomplished by far. I feel very comfortable now with reading deep learning papers and thinking about possible applications of this incredible tool. I even had the pleasure to attend a conference in Barcelona with top researchers in the field. Also, I understand how high-dimensional distributions are modelled and most importantly why these models are used.
2. Also, we succeeded in reviewing and comparing these 4 approaches in the field of image generation. We compared their ability to yield quality samples, their ability to evaluate the likelihood function, and their ability to learn a low dimensional representation.

3. I have learned PyTorch, the most promising deep learning library, as well as how to develop and structure a machine learning project in general (using GitHub as well).

4. Finally, I have written this thesis in a way that it could be understood by a wide audience without renouncing the use of advanced mathematics. My hope is that this will be used as an introduction to the field of deep learning and deep generative models.

7. Bibliography

The most important sources of knowledge for this work have been the Deep Learning Book [7], two different online courses on Deep Generative Models [20] and [1], and the Deep Learning Specialization on Coursera [15].

References

[1] Pieter Abbeel. *Deep Unsupervised Learning*. Online Course. 2019. URL: https://sites.google.com/view/berkeley-cs294-158-sp19/home.

[2] DeepMind. *WaveNet*. Audio generation. 2016. URL: https://deepmind.com/blog/article/wavenet-generative-model-raw-audio.

[3] Laurent Dinh, David Krueger, and Yoshua Bengio. *NICE: Non-linear Independent Components Estimation*. 2014. arXiv: 1410.8516 [cs.LG].

[4] Carl Doersch. "Tutorial on variational autoencoders". In: *arXiv preprint arXiv:1606.05908* (2016).

[5] Mathieu Germain et al. *MADE: Masked Autoencoder for Distribution Estimation*. 2015. arXiv: 1502.03509 [cs.LG].

[6] Ian Goodfellow. "NIPS 2016 tutorial: Generative adversarial networks". In: *arXiv preprint arXiv:1701.00160* (2016).

[7] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[8] Ian Goodfellow et al. "Generative adversarial nets". In: *Advances in neural information processing systems*. 2014, pp. 2672–2680.

[9] Sergey Ioffe and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *arXiv preprint arXiv:1502.03167* (2015).

[10] Tero Karras, Samuli Laine, and Timo Aila. "A style-based generator architecture for generative adversarial networks". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2019, pp. 4401–4410.

[11] Adam King. *Talk To Transformer*. Text generation. URL: https://talktotransformer.com/.

[12] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[13] Hugo Larochelle and Iain Murray. "The neural autoregressive distribution estimator". In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. 2011, pp. 29–37.

[14] Yann LeCun, Corinna Cortes, and CJ Burges. "MNIST handwritten digit database". In: *AT&T Labs [Online]. Available: http://yann. lecun. com/exdb/mnist* 2 (2010), p. 18.

[15] Andrew Ng. *Deep Learning Specialization*. Deep Learning course. URL: https://www.coursera.org/specializations/deep-learning.

[16] Michael Nielsen. *A visual proof that neural nets can compute any function*. Neural Networks and approximation. URL: http://neuralnetworksanddeeplearning.com/chap4.html.

[17] Aaron van den Oord et al. *WaveNet: A Generative Model for Raw Audio*. 2016. arXiv: 1609.03499 [cs.SD].

[18] Albert Pumarola et al. *C-Flow: Conditional Generative Flow Models for Images and 3D Point Clouds*. 2019. arXiv: 1912.07009 [cs.CV].

[19] Alec Radford et al. "Language models are unsupervised multitask learners". In: *OpenAI Blog* 1.8 (2019), p. 9.

[20] Aditya Grover Stefano Ermon. *Deep Generative Models*. Course offered by top researchers in this field. URL: https://deepgenerativemodels.github.io/.

[21] Benigno Uria et al. *Neural Autoregressive Distribution Estimation*. 2016. arXiv: 1605.02226 [cs.LG].

[22] Wikipedia. *Artificial Neuron*. Used for an image. URL: https://en.wikipedia.org/wiki/Artificial_neuron.

A. Programs

One can find the 4 codes in my github `https://github.com/VictorSP17`. The repositories are: **Simple GAN MNIST**, **VAE MNIST**, **MADE MNIST**, and **NICE MNIST**. In each of them there is a jupyter notebook with all of the code.