POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Using Deep Generative Models for Representation Learning with Applications to AI Explainability

Supervisor
Prof. Tatiana Tommasi

Candidate
Davide Fiorino

Company supervisor
Clearbox AI
Luca Gilli Ph.D.

April 2021

Alla mia famiglia e a Ioanna, per aver sempre creduto in me.

Summary

In the last few years, the great potentiality of deep learning has been proven by cutting-edge artificial intelligence applications. Ranging from well-defined tasks like object recognition or speech translation to the more generic autonomous driving, deep learning has been proven to be an extremely powerful tool. One fundamental concept on which deep learning is implicitly built upon is representation learning: how we represent data matters in how we understand the world. In this work, we first study the concept of representation learning, the methods to perform it and its various applications. With the goal of contextualizing this framework to deep generative models, we study the most popular generative methods and make a qualitative comparison between them. In the last section, we will focus our attention on the InfoGAN approach, which imposes explicit conditions on the input representation and is based on the very-well performing Generative Adversarial Network model. Experiments are carried out with InfoGAN on increasingly complex datasets to assess its performance and limitations. We finally discuss how to use GANs in the rising popular field of AI explainability.

Acknowledgements

I want to thank my company supervisor Luca Gilli of Clearbox AI for his constant and valuable support and for the interesting conversations that produced the ideas illustrated in this work.

I also want to thank Prof. Tatiana Tommasi, for all the knowledge she transmitted us students in her Machine Learning course. I further want to thank all the great Professors at Politecnico di Torino that try to transmit us their experience and cutting-edge ideas.

Finally, I want to thank my student's association, the Mu-Nu-Chapter of IEEE HKN, for all the incredible people I met there.

Contents

Chapter 1

Introduction

Intelligence is arguably the most advanced capability that human beings have compared to animals and other living beings. For thousands of years, we have studied our brain and tried to explain how it can perceive, understand, make plans about the future, remember things, and feel emotions. The field of artificial intelligence (AI), goes even further by studying how to build intelligent entities at the service of humanity. Since the invention of digital computers in the 40s, the idea of building intelligent machines come back in fashion, but it was still far away. One of the first scientists to mention computer intelligence was Alan Turing, during a public lecture in London in 1947, saying, "What we want is a machine that can learn from experience" and invented the famous Turing test, to define the intelligent machine. Turing also proposed a goal for computers: being able to win in a chess game, a problem that if tackled as a search problem has a huge search space. About 50 years later, in 1997, the IBM computer Deep Blue, defeated the world chess champion, Garry Kasparov, in a six-game match. Even if this moment became historical, his success was almost entirely due to the progress of electronics, that allowed the computer to examine 200 million possible moves per second.

Artificial intelligence was still far away; many years of research with new algorithms and vast hardware improvements lead to where we are today. In 2016, Google AlphaGo defeated one of the strongest players of Go, Lee Sedol, using powerful hardware, deep neural networks and reinforcement learning. The game of Go is many order of magnitudes more complex than chess: the estimated number of possible board configuration in chess is 10^{120}, while is around 10^{174} for Go. Today, commercial products that use text, image, and speech recognition are of common use. Systems like automated customer support, personalized shopping experience, and drones are all based on AI. The fields of healthcare and finance are also benefiting from the technology, and self-driving cars are on their way. Furthermore, the research in AI is incredibly active with a number of publications that grow exponentially since the 2000s [1]. Yet, the concept of general artificial intelligence, like a robot that can think, perceive, and plan like a human is not a reality.

While in the early days, the field of AI was dominated by trying to solve formally-describable problems, like playing chess, more recently the focus has been to imitate tasks that are intuitive for humans, like recognizing faces, but almost impossible to describe formally. The solution found was to build machines that can

learn from experience, by extracting patterns from data, hence the field of machine learning (ML). The latest branch of ML is called deep learning, and it is based on the idea of learning complex abstract concepts by building them out of simpler ones. While in classical ML algorithms, the performance of the algorithms is highly dependent on the data representation given, deep learning algorithms can learn to extract a good representation and build understanding on top of it. This idea is so present in machine learning that is studied as a separate field, called representation learning. Nowadays, deep learning algorithms are able to reach human-level performance in some well-defined tasks like speech recognition, object recognition, semantic segmentation, but a broader and more general intelligence is not here yet. Another exciting challenge is being able to learn more from unlabeled data, which are readily available, leveraging what is called unsupervised learning.

1.1 Thesis Structure

The following work is articulated around three main related topics: representation learning, deep generative models and AI explainability. The experimental part is divided into three parts, regarding InfoGAN and possible applications to explainable AI. The structure of thesis is as follows:

- **Chapter 1** is an introduction to the work.
- **Chapter 2** discusses representation learning, the field of machine learning based on the concept that the performance of a model is heavily dependent on the representation used for data. We also discuss what makes a representation better than another one and why explicitly caring about the quality of this representation is important. The most popular representation learning approaches are presented.
- **Chapter 3**: with the goal of contextualizing the representation learning framework to deep generative models, we study the most popular generative methods and make a qualitative comparison between them. We distinguish between GANs, which work with an implicit density function, Variational Autoencoders that approximates an explicit density, and autoregressive models.
- **Chapter 4** present the experimental study. We start by describing InfoGAN, the main model we use for our experiments. In the first part of the study, we perform a benchmark of InfoGAN on increasingly complex datasets, evaluating image quality and variety both visually and objectively. We also evaluate the quality of the extracted representation in terms of interpretability. In addition, we investigate the usage of Dropout on training stability. In the second part, we employ the results of the previous part to create an explainability system that we call InfoGAN Explainability Grids. In the third experiment, we finish by employing InfoVAE to create Decision Maps that highlight the most important regions of an image for the classifier's decision.
- **Chapter 5** contains the conclusions and possible future developments.

Chapter 2

Representation Learning

The notion of data representation is pervasive in all the fields of computer science: there is always a way to represent data that is more convenient for the operations that we want to perform. As humans, we represent information in our brain in a way that is likely very different from the way we store them in computers. Moreover, even between humans, we represent the same concepts in different ways according to what is more convenient for us at the time. For example, we express temperature in Celsius degrees if we care about how we should dress that day, but we express it in Kelvin if we are working at a physics experiment at very low temperatures. We express distances in millimeters if we are talking about the size of a wrist-watch while using kilometers for road distances. In signal processing, we represent an audio signal as an ordered list of multi-dimensional vectors. We divide the input signal in chunks using sliding time windows, and to each chunk apply some filters, each outputting a number that we put together in a vector. The number of filters controls the dimensionality of the output space. Such a representation will be useful for many signal processing tasks.

For machine learning, more specifically, a good representation (or features) is one that eases the learning task at the end of the pipeline. Let us say we are trying to use a simple linear regression model; this will likely not be able to work directly on images represented as raw pixel values. If we apply a series of transformations to the images in order to represent it as high-level features, the same model might be able to obtain useful results. To make an elementary example, say we have a dataset of pictures of semaphores, and we want to create a classifier that discriminates the three different colors (red, yellow or green). We can now transform raw images (a very high dimensional object) into vectors in a 3-dimensional space, corresponding to the RGB color percentage in each image. With this new representation, even a simple linear classifier might be able to identify the color of the semaphore. In this example, we have a very efficient representation, but that is also extremely task-specific. As we will see, we usually strive to obtain a representation that is good for a multitude of tasks.

The starting hypothesis in representation learning is that the performance of machine learning methods is heavily dependent on the choice of data representation [2]. A good feature space might be crafted manually by using domain-specific expertise (feature engineering), or might be learned automatically by an algorithm.

A representation can be learned explicitly if we are using an algorithm with this exclusive purpose, or implicitly if we are using a deep learning algorithm with a specific task (like classification or regression) and feeding it raw data. Feature engineering was more common in the past for methods like Support Vector Machines (SVM) and decision trees. It is an activity that exposes the weakness of the learning algorithms in extracting such meaningful and discriminative features. Being the process based on the specific application domain is really time-consuming and requires expertise, and for some complex and high-dimensional datasets, it can be very challenging.

With the development of deep learning methods, the process of extracting interesting features and performing a learning task is usually integrated into a single model. If we talk about neural networks, we can see them as applying a series of parameterized non-linear transformations in order to produce a higher-level representation. The last layer of a neural network is usually a linear classifier (e.g. a softmax regressor) while the rest of the model learns to provide a useful representation to this classifier. In this case, since we are using a supervised training, the network is producing features that are explicitly crafted to benefit the task at the end. At every layer of the network, we have a different representation, that gets semantically more complex moving towards the last layers. This concept has been clearly observed in convolutional neural networks (CNN) for images, where the first layers are sensible to simple concepts like edges while the last layers might be able to recognize more complex shapes and even objects [3].

A significant distinction in all machine learning is made between supervised and unsupervised learning. In the supervised setting, we use prior knowledge of the output value that the algorithm should produce (labels), and we try to make the computed output close to the correct one. An example of this is a classification algorithm like linear regression. In the unsupervised setting, instead, we only leverage the data, and we try to learn interesting features about them without any specific control signal. An example of this approach is density estimation, like k-means clustering. While supervised training of a neural network does not involve explicitly imposing constraints on the representation at each layer, other kinds of representation learning algorithms are designed to shape the representation in some particular way. For example, a constraint that is usually imposed is the disentanglement of features, as will be analyzed later. Furthermore, representation learning provides an interesting way of applying semi-supervised learning: we can leverage the enormous amount of unlabeled data available to learn a good representation and then specialize the model in one task using a few labeled examples. An additional benefit of this approach is that it offers the chance to solve the overfitting problem by also learning from the unlabeled data [4, Chap. 15].

Summarizing, a challenge in machine learning is "learning representations of the data that make it easier to extract useful information when building classifiers or other predictors" [2]. A good representation makes learning easier since it brings out the actual degrees of freedom in the data; it captures relevant structure at multiple scales and filters out noisy and irrelevant structures. The process of a ML pipeline with explicit representation learning can be summarized in three simple steps (fig. 2.1): getting the raw data of interest, convert them into a better representation and performing the learning task.

Figure 2.1. The three logical steps of a machine learning pipeline with explicit representation conversion. The better representation can be obtained with feature engineering or through representation learning algorithms.

Applications of representation learning are the most disparate since all the modern deep learning approaches are based on this concept. Speech recognition, natural language processing (NLP), object recognition and image segmentation are all examples of tasks that display great accuracy with today's state-of-the-art methods. As a matter of fact, products based on the above-mentioned technologies are massively available in the consumer and industrial markets. More technical applications are in the related fields of transfer learning, domain adaptation and multi-task learning. As defined by [2], "transfer learning is the ability of a learning algorithm to exploit commonalities between different learning tasks in order to share statistical strength, and transfer knowledge across tasks". By learning a representation that is not task-specific, one can share it between different tasks with similar data, or similar tasks with the same kind of data. If we create a hierarchical representation, the lower layers might be shared between multiple tasks, while the upper ones might be learned in a supervised way for a specific purpose. In this way, multiple tasks can be learned together (multi-task learning). The concept is that, regardless of how representation was obtained, it should be possible to use it for a different task.

In most machine learning methods used today, feature learning is a theoretical concept embedded in the model, so it is easy to forget about it. That being said, caring explicitly about this aspect can bring noticeable advancements to state of the art, as deep-learning recent signs of progress have proven.

2.1 Obtaining a good representation

Since there is technically an infinite number of representations for any particular kind of data, how do we choose and build a good set of features? All the available methods deal with a trade-off between preserving as much information about the input as possible and having nice properties in the representation. One thing that can help is embedding some priors (or hints) in the learning algorithm; this is a way to help the learner discover the underlying causal factors of variations of the data. Representations can be crafted in order to express some general priors about the nature of the data space. General priors embed concepts that are useful for a variety of different tasks and distributions. Supervised learning labels are a solid clue about a particular feature of the data, but we also want to leverage unlabeled data by expressing some more subtle and generic clues. In order to make a learning algorithm generalize well, we use regularization, which is basically a way of expressing a constraint on the complexity of the model, to avoid overfitting. The following priors, introduced in [2] and extended in [4, Chap. 15], can also be seen as regularization strategies.

- **Smoothness**: assumes the function to be learned $f()$ is s.t. $x \approx y$ generally implies $f(x) \approx f(y)$. This assumption allows the generalization of results from training examples to nearby points in the input space; this is a basic prior upon which are based most machine learning models. The problem it has is the curse of dimensionality: generalization is achieved by doing a local interpolation between close training examples, but the abrupt variations in the target functions may grow exponentially with the number of input dimensions. Kernels machines and linear models are based on this assumption.

- **Linearity**: assumes a linear relationship between variables in the data. This is a strong assumption that allows making inference even in the space very far from most of the training points. Most simple machine learning algorithms make this assumption, that does not imply smoothness.

- **Multiple explanatory factors**: assumes that data distribution is generated by different underlying explanatory factors and that most tasks can be solved easily by retrieving at least some of those factors. As described before, this view justifies the semi-supervised learning approach. The features that are useful for learning $p(x)$ are also useful to learn $p(y|x)$ because they both refer to the same underlying explanatory factors. This idea is also behind the concept of distributed representations.

- **Causal factors**: assumes that the factors of variations in the learned representation are the causes of the training data x, but not vice-versa. This is useful for methods like transfer learning and domain adaptation, where the distribution over the underlying causes changes or we use the model for a new task.

- **Hierarchical organization of explanatory factors**: the idea is that it is possible to express high-level concepts in terms of simple concepts and do this multiple times to build a hierarchy. This is exactly what happens in

deep feed-forward neural networks, where the number of layers determines the depth of this hierarchy and each layer work on the output of the previous one. This idea is at the core of the incredible deep learning progress.

- **Shared factors across tasks**: assumes that many different tasks are related in terms of the low-level explanatory factors. This is the hypothesis behind multi-task learning. The sharing of the internal representation corresponds to a sharing of statistical strength between different tasks. More formally, if we have input x, outputs y_i for the different tasks and a common set of factors h, we can learn each $P(y_i|x)$ via a shared intermediate representation $P(h|x)$.

- **Manifolds**: assumes that probability mass concentrates near regions that have a much smaller dimensionality than the original space of the input data. These regions are locally connected and occupy a tiny volume. Many machine learning algorithms model very well only in those regions. Autoencoders are explicitly based on this hypothesis since they compress a lot the dimensionality of the original representation.

- **Natural clustering**: assumes that the input space is made of many different manifolds and data that belong to each manifold have common characteristics so that it is possible to assign them a common label. Manifolds might be disconnected, but the label remains constant within each one of these. The manifold tangent classifier and adversarial training are based on this assumption.

- **Temporal and spatial coherence**: temporally consecutive or spatially nearby observations tend to be associated with the same value of categorical concepts, or they result in a small move on the surface of a high-density manifold. This prior can be enforced by penalizing fast changes in values over time or space.

- **Sparsity**: assumes that for any given input point x, only a small fraction of the possible factors are relevant. In terms of representation, this means having features that are often zero (sparse features). For example, a classifier trained on various kind of images might set most feature to zero when trying to classify images of a specific subcategory (like animals). It is, therefore, reasonable to enforce that any feature that can be interpreted as present or absent should be absent most of the time.

- **Simplicity of factor dependencies**: assumes that in good high-level representations, all the factors should be related to each other through simple relationships. Some examples are linear relationship, marginal independence or those captured by shallow autoencoders.

An important concept in evaluating representations is **expressiveness**: which is the ratio between the number of possible configurations and the size of the representation. Ideally, we want a small number of features to capture a vast amount of different abstract concepts. We can also define expressiveness as the number of parameters required, compared to the number of regions distinguishable in the

input space. Algorithms that learns one-hot representations like traditional clustering, Gaussian mixtures, nearest neighbor, decision trees, or Gaussian SVM all require *O(N)* parameters (and/or *O(N)* examples) to distinguish *O(N)* input regions. However, algorithms that learns sparse or distributed representations like Restricted Boltzmann Machines (RBM), sparse coding, auto-encoders or multi-layer neural networks can all represent up to $O(2^k)$ input regions using only *O(N)* parameters (with k the number of non-zero elements in a sparse representation, and $k = N$ in non-sparse RBMs and other dense representations) [2].

Nowadays, the most powerful machine learning methods are based on **deep network architectures**, whether they are standard neural networks, CNNs, recurrent neural networks (RNN) or sequence models. For deep networks, we intend with a large number of layers, in contrast to shallow networks. These kinds of models have the advantages of promoting feature re-usage and naturally leading to more abstract features as we go deeper. The re-usage of features allows building the powerful property of the hierarchy of concepts. On the other side, these models are hard to train, requiring many different techniques to obtain convergence in a reasonable time. The theoretical advantage of deeper models can be explained with the learning theory's concept of **VC-dimension**, whose definition is reported below. The VC-dimension of a hypothesis class gives the correct characterization of its learnability. To explain it, we first need to define the concept of Shattering [5, Chap. 6].

Definition 1. Shattering: Given $H[S]$ equal to the set of splittings of dataset S using concepts from H, we will say that H *shatters* S if $|H[S]| = 2^{|S|}$.

In other words, a set of points S is shattered by H if there are hypotheses in H that split S in all of the 2^S possible ways. This also means that all possible ways of classifying points in S are achievable using concepts in H.

Definition 2. Vapnik-Chervonenkis dimension: The VC-dimension of a hypothesis space H is the cardinality of the largest set S that can be shattered by H. If H can shatter sets of arbitrarily large size we say that H has infinite VC-dimension.

Under the definition of VC-dimension, in order to prove that $VC(H)$ is at least d, we need to show only that there is at least one set of size d that H can shatter. As an example, consider H to be the set of all linear classifiers in 2 dimensions (fig. 2.2). In this case, $VC(H) = 3$ since there is at least a set of 3 points that an hypothesis from H can classify in all possible ways. On the contrary, there is no set of 4 points that any hypothesis from H can classify in all possible ways.

Computing the VC-dimension for a neural network is a lot more complex, but the following result holds true:

Theorem 1. The class of functions computed by multi-layer neural networks with binary as well as linear activations and ρ weights has VC dimension $O(\rho^2)$ [6].

So the VC dimension of a neural network depends on the number of parameters. According to learning theory, if a family of functions can be represented with a smaller VC-dimension than it can be learned with fewer examples. A smaller VC-dimension means a model with less capacity, that can cover the desired functions

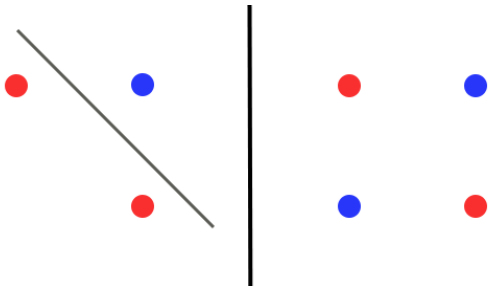


Figure 2.2. Two set of points of two classes (red and blue) in two dimensions. On the left set it is also illustrated a possible linear classifier, and different linear classifiers can classify these 3 points in all possible ways. On the right set there is a configuration of 4 points that is not possible to model for any linear classifier.

with less risk of overfitting. There are also advantages related to improvements in computational and statistical efficiency and the re-use of parameters over many different kinds of inputs [2]. Deep architectures have a massive number of parameters, but thanks to some theoretical results, we know that for the same number of parameters, a deep architecture is exponentially more efficient than a shallow one [7]. This is due to the fact that deep architectures can exploit the powerful concept of hierarchical representations.

Another interesting concept that was used more in the past when describing a good representation for pattern recognition, speech recognition and computer vision is the idea of **invariance**. Having invariant features means that we are interested in features that are sensitive to the aspects of the data we care about and insensitive to the others. With the hierarchy of concepts of deep networks, we have that the more abstract concepts are generally invariant to most local changes of the input. In contrast, the opposite is exact for low-level features. The one above is a powerful idea, but it is not enough for unsupervised learning, where we do not know in advance the applicative task, so we would like to keep all the crucial factors.

The alternative to this approach, **disentanglement**, means learning to disentangle the causal factors of variations in the input data, which can intuitively provide a high-level understanding of them with "a few" separate features. If we think, for example, about the image in fig. 2.3, we can describe it in its entirety with words like "Abbey Road album cover" or "the Beatles crossing the street", and this description would be enough for another human that knows this image to have it in his mind. What an efficient representation! Nevertheless, if the other person does not know the picture, we might describe it as "four men crossing a street on the crosswalks, in the '60". We can also go even lower level and describe all the objects in the picture, and even how each of this object is made in terms of shape and color.

In the above example, we are going from a more abstract high-level representation to a lower-level one. The other thing is that all of these descriptions contain disentangled ideas. However, how can we disentangle the concept of cars, from the concept of men or lighting in the scene? In the recent and powerful approaches of deep learning, the idea is to leverage the data itself, by using vast quantities of

Figure 2.3. The Beatles: Abbey Road. An image like this can be described at many level of abstraction. (source: Reuters).

unlabeled data. This is the most effective way of learning to separate the various explanatory factors in a robust and flexible representation. Nowadays using features learned in an unsupervised way to initialize a neural network has become common practice (unsupervised pre-training). Seeing machine learning in the optic of modeling dependencies between variables, disentanglement makes these dependencies much simpler because the data is projected into an abstract space of high-level concepts. In such a space, even the curse of dimensionality is hugely alleviated. The central difference between invariance and disentanglement is the preservation of information: while invariant features discard some information by definition, with disentangled features, we try to keep as much information as possible [2]. Regarding the above-mentioned priors, we can view many of them as ways to help the learner to discover and disentangle the underlying causal factor of input data.

All the ideas we have discussed so far have no practical utility if they cannot be translated into a **training objective**, like a cost function for an optimization problem. Defining a criterion to keep some properties in the representation is not as easy as defining it for a specific task, like classification or regression. Often we want a unique model that performs a specific task while learning a good set of features, so two objectives must be combined. This problem has been investigated extensively in recent research. The first thing that one can do is provide some form of measurement of disentanglement between features and try to maximize this quantity.

In [8], Glorot et al. used stacked denoising autoencoders to extract high-level features from the unlabeled text of product reviews. They later trained a classifier to perform sentiment analysis and successfully performed domain adaptation on a large-scale dataset of 22 different domains. By evaluating a deep architecture on sentiment analysis, they have been able to evaluate disentanglement. If the model is able to somewhat disentangle the underlying factors, it will improve cross-domain transfer since there exist generic concepts that characterize product reviews across many domains. Besides, because some of these factors about the source dataset were known, they could measure how the same factors are disentangled in the learned representation.

An interesting visual analysis is reported in fig. 2.4 where the number of features used for two families of tasks is plotted as a color map, varying the task. The same plot is reported using raw input data, and data with features transformed with the proposed method (SDAsh). Comparing the two graphs, we can appreciate how relevant features for domain recognition and sentiment analysis are far less overlapping in the case of SDAsh. This means that the features extracted are a lot more disentangled since their re-usage is more distinctly distributed between the two families of tasks. Other approaches will be analyzed more extensively in the rest of this thesis.

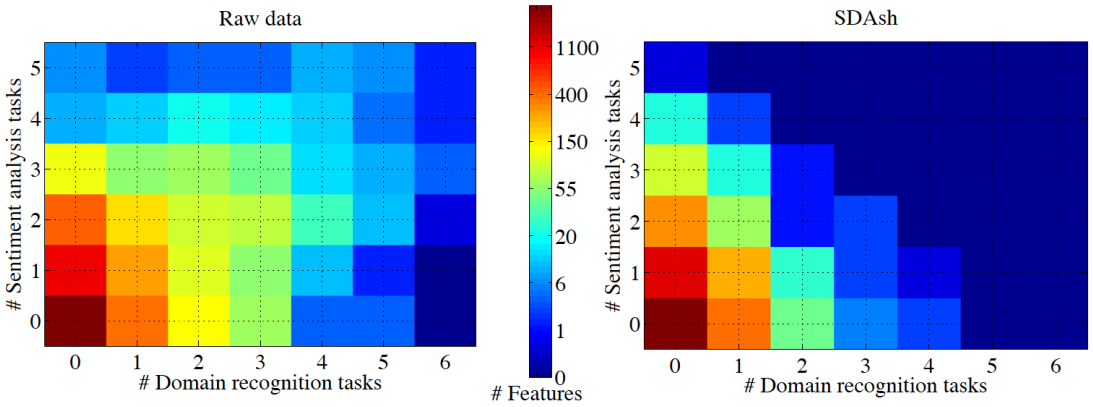


Figure 2.4. Number of tasks of domain recognition (x-axis) vs sentiment analysis (y-axis). The color level at coordinate $(n;m)$ indicates the number of features that have been re-used for n sentiment analysis tasks and for m domain recognition tasks. Each feature is re-used by classifiers trained on raw features (left) or features transformed by SDAsh (right) [8].

Another critical point is being able to **evaluate the quality of a representation**. Since the goal of the representation is usually to improve performances of a following task, the most logic evaluation can indeed be done on the accuracy of a task, like classification. This is enough if we only care about using the representation on a specific task. However, it is a non-practical method to evaluate features that should be used in a variety of tasks, since for every change in the representation we should train and evaluate a large number of models. Furthermore, this is always an incomplete evaluation of the features since it is task-dependent.

The better alternative is to evaluate unsupervised performances. Autoencoders and its variants are one of the most popular ways of feature learning. Intuitively the representation created by the encoder is good if the decoder is able to accurately reproduce the original data. A way to evaluate this is to measure the test set reconstruction error, which can easily be computed. This evaluation, though, is dependent on model capacity, since a more capable model will likely have a lower reconstruction error. One can also use denoising reconstruction error in denoising autoencoders, that by definition is immune to this problem. For methods based on undirected graphical models (like Boltzmann machines), the evaluation is more complex [2]. More recently, a lot of metrics for evaluating disentanglement were invented. The idea is that a change in a single underlying factor of variation $\mathbf{z}_i$ should lead to a change in a single factor in the learned representation $r(\mathbf{x})$. For

variational autoencoders (VAE) some of the metrics available are: the *BetaVAE*, the *FactorVAE*, the *Mutual Information Gap*, the *Modularity*, the *DIC Disentanglement* and the *SAP Score* [9]. Some of these metrics will be analyzed later in the work.

In [10], Kolesnikov et al. studied the performance of self-supervised representation learning techniques applied to visual tasks, such as image recognition. They used four different self-supervision models to learn high-quality features and then evaluated them by measuring their performance in some supervised tasks. The results in terms of downstream task accuracy are reported in fig. 2.5. Four different self-supervision training techniques are used. One interesting thing they proved is that "increasing the number of filters in a CNN model and, consequently, the size of the representation significantly and consistently increases the quality of the learned visual representations" [10].

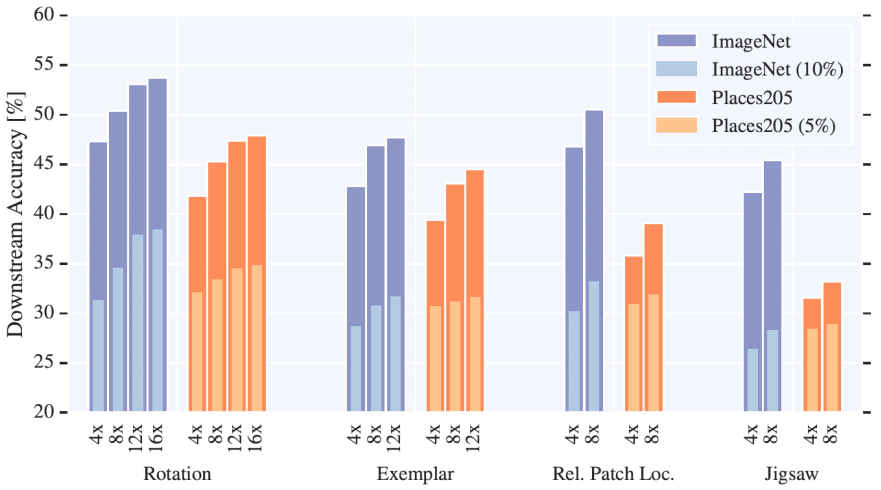


Figure 2.5. Downstream task accuracy on ImageNet and Places205, for four different semi-supervision techniques. In general, increasing the size of the features improves the visual representation quality [10].

2.2 Representation learning methods

2.2.1 Principal Component Analysis

One simple example of an unsupervised representation learning algorithm is principal component analysis (PCA), which is a form of manifold learning that supposes a linear manifold. Given a dataset of points in an d dimensional space, PCA finds a representation of the points in terms of m linearly uncorrelated variables called principal components. Other than finding the components, the method orders them in terms of descending variance. Since PCA is usually used for dimensionality reduction, this information allows us to pick the percentage of cumulative variance that we want the new components to retain. The algorithm can keep the components with the greatest variance under the constraint that each component is orthogonal to the other ones. An example of a PCA transformation on a 2-dimensional dataset is shown in fig. 2.6.

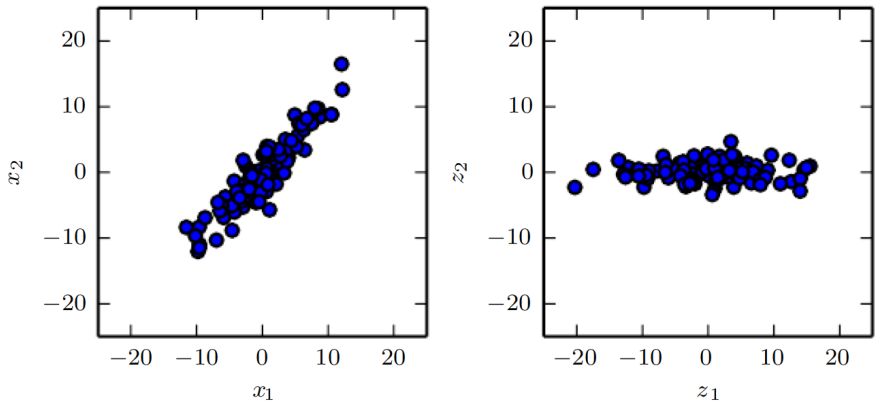


Figure 2.6. In the image above, the dataset was originally expressed in terms of *x1* and *x2*. After PCA, which can be seen as a rigid rotation, the new space is identified by *z1* and *z2*. The maximum variance of the data is across *z1*, therefore removing *z2* would not make us lose much information [4].

One way of obtaining PCA is trough eigen-decomposition of the covariance matrix. This matrix can be obtained by computing: $\mathbf{S} = \mathbf{X}\mathbf{X}^T$, where $\mathbf{X}$ is a d by n design matrix where each row represents a feature and each column represents a sample. Each eigenvector will represent a component in the new space, and its eigenvalue will be a quantity proportional to its variance. We can obtain the transformation matrix ($\mathbf{B}$) as the matrix where each column is an eigenvector of the components that we want to keep. By applying transformation $\mathbf{B}$ to a data vector ($\mathbf{x}$) we will obtain its projection in the subspace spanned by the columns of $\mathbf{B}$, known as the code ($\mathbf{c}$). By applying the inverse transformation to the code, we will obtain the reconstruction of the original vector in the original space ($\bar{\mathbf{x}}$). Since B is an orthogonal matrix (the inverse is equal to the transpose) we can write:

$$\bar{\mathbf{x}} = \mathbf{B}\mathbf{B}^T\mathbf{x} = \mathbf{B}\mathbf{x} \text{ where } \mathbf{x}, \bar{\mathbf{x}} \in \mathbb{R}^d \mathbf{c} \in \mathbb{R}^m$$

In this case, we suppose to keep m components (with $m < d$), so the code will be

m-dimensional. The accuracy of this approximation depends on how many components we choose to keep. If $m = d$ than the reconstruction vector will be equal to the original data vector. PCA is a useful tool used in compression, denoising, data visualization and feature engineering. Furthermore, the new representation can be obtained with a closed form solution. On the other hand, its limit is the sensibility to only linear dependencies, which makes it hard to disentangle more complex relationships.

2.2.2 Word embedding

In the field of natural language processing (NLP), many models dealing with words use a representation called word embedding, which is a way of representing words as vectors in an N-dimensional space. While the simplest representation for words is probably one-hot encoding, this is extremely inefficient since it forces to work in a huge dimensional space (the size of the vocabulary) where points are very sparse. One can instead design a lower-dimensional space where to map the words and try to position them by looking at their semantic and syntactic meaning. This is the space created by a word embedding algorithm and usually goes from 100 to 1000 dimensions. An example of words placed according to semantic and syntactic meaning is shown in fig. 2.7. Word embeddings are usually obtained with unsupervised methods, where the network is trained on identifying neighbor words in sentences. If we would use a supervised training, as the classification of positive vs negative reviews, we will get a representation that is biased towards separating the words associated with the two different classes. A very popular algorithm for obtaining word embedding is called Word2vec [11].

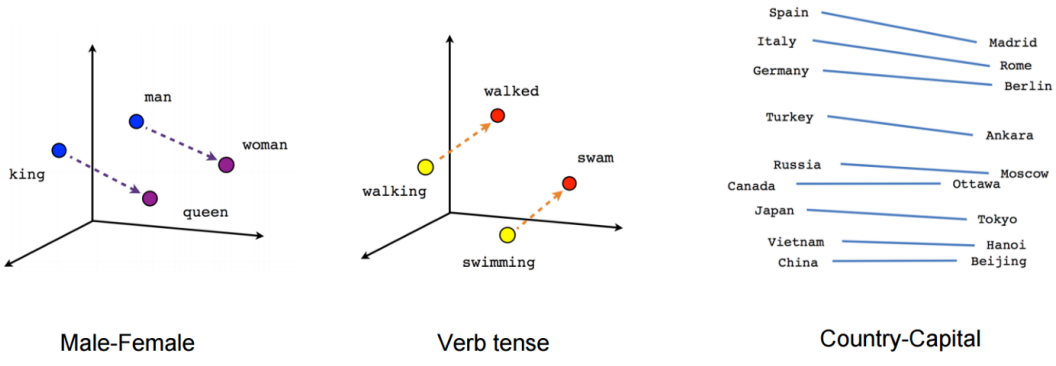


Figure 2.7. A visual example of word embedding. The position of word vectors in the embedding space encodes semantic information.

2.2.3 Autoencoders

The most typical example of a representation learning algorithm is the autoencoder, whose idea has been around since the 80s. Autoencoders were traditionally used for dimensionality reduction or feature learning, while nowadays they are also being employed for generative modeling. An autoencoder can be seen as a particular

case of feed-forward network composed of a combination of an encoder function that converts input data into a new representation, and a decoder function that converts the latent representation back into the original form. We will denote the encoder function as $f_\theta()$ and say that it is parameterized in a closed form by parameters in vector θ. The learned representation or features extracted or code will be denoted as vector $h = f_\theta(x)$ for each input vector x. The decoder function can than be defined as $g_\theta(h)$ and, given the code, it produces a reconstruction of the original input vector: $r = g_\theta(h)$. Autoencoders are trained to preserve as much information as possible, more formally to minimize the reconstruction error $L(x, r)$ which is a measure of the discrepancy between the two inputs (like L2 distance). Furthermore, if the network generalizes well, we want this error to be low on test data while having higher values for other inputs. This means that we do not want an autoencoder that learns to set $g_\theta(f_\theta(x))$ everywhere (identity function), since that would be useless. For this reason, autoencoders are designed to be unable to copy perfectly, by forcing a restriction on the intermediate representation h. The most basic constraint is that the dimensionality of h must be lower than that of x. This way, the network is forced to throw away some information, while still trying to keep $L()$ low; therefore, it learns to extract the most important features. Other than learning to preserve as much information as possible, autoencoders are trained to make the latent representation have specific properties. For this reason, there are various kinds of autoencoders, each trying to obtain different properties. The cost function minimized when trained an autoencoder can be defined as:

$$J(\theta) = \sum_t L(x^{(t)}, g_\theta(f_\theta(x^{(t)}))) + \gamma R()$$

where the sum goes for all $x^{(t)}$ training examples. $R()$ is an optional regularization function that can be used to express further conditions and $\gamma \in \mathbb{R}$ is an hyper-parameter to control the regularization strength. The architecture of an autoencoder network where h has a smaller dimension than x is shown in fig. 2.8. This configuration is sometimes called under-complete autoencoder.

An under-complete autoencoder, where the decoder is a linear operator and the loss function is the mean squared error, corresponds to learning the same subspace as PCA. When both the encoder and the decoder functions are non-linear, the result is a much more robust feature learning algorithm, even if this higher capacity must deal with the overfitting problem. To deal with this obstacle, we use an appropriate regularizer function ($R()$) that encourages properties such as sparsity of the representation, smallness of the derivative of the representation, and robustness to noise or missing inputs [4, Chap. 14]. For many applications, autoencoders are made with only one encoder and one decoder layers, but deeper architectures offer many advantages. The universal approximator theorem ensures that "with at least one hidden layer a feed-forward network can represent an approximation of any function to an arbitrary degree of accuracy, provided that it has enough hidden units" [4, Chap. 14]. A deeper autoencoder can also reduce the computational cost of learning some complex functions and the amount of training data needed.

As we already mentioned, the hypothesis of the manifold in machine learning assumes that data are concentrated near regions that have a much smaller dimensionality than the full data space. Furthermore, the points in this region have

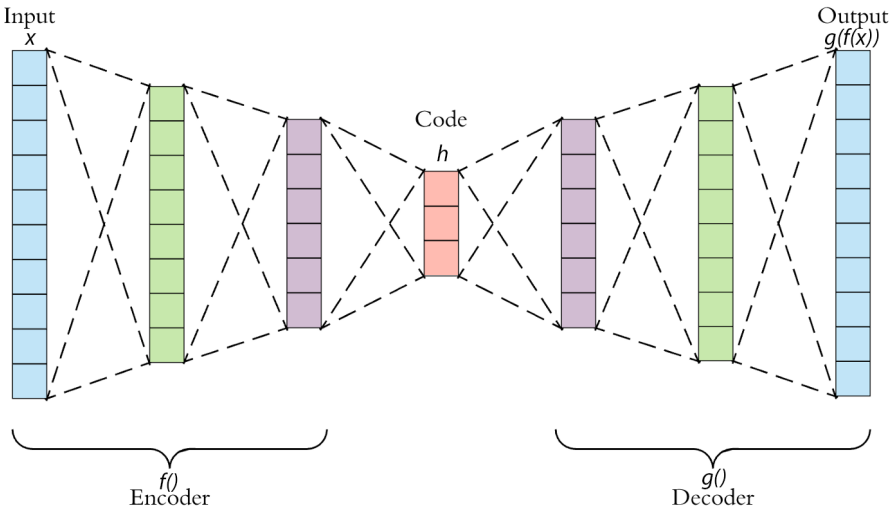


Figure 2.8. The architecture of an under-complete autoencoder. The code h has a smaller dimensionality than the input vector x, therefore the model will try to keep the "most important" information in the learned representation.

nice abstract properties. Autoencoders also use this idea and even try to learn the structure of the manifold. The training procedure tries to balance between the goal of decreasing reconstruction error and the goal of satisfying the constraint of regularization. The result obtained by applying this remarkable recipe is that "autoencoders can afford to represent only the variations that are needed to reconstruct training example" [4, Chap. 14]. This simple sentence summarizes the idea behind regularized autoencoders completely. Now, if the manifold hypothesis holds true on the data distribution we are interested in, according to the considerations above, the learned representation will implicitly capture a local coordinate system for the manifold. This is because only the variations that make the input value x remain in the manifold need to cause variations to the code h. The autoencoder needs to reconstruct well only points belonging to the data-generating distribution. The concept of learning a representation that allows moving in the local coordinates of the manifold is illustrated in fig. 2.9.

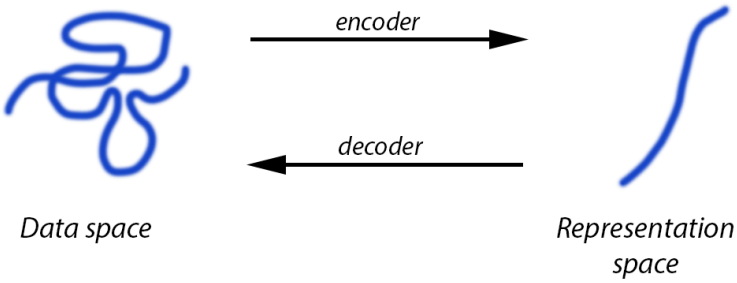


Figure 2.9. In the data space the manifold of the data might have a very complex shape, like a "spaghetti". With an autoencoder it is possible to flatten and disentangle the data manifold, like in this example where in the learned representation space the manifold takes a much simpler shape.

From another point of view, we can imagine having the manifold occupying a

very complex shape in the original data shape. The equations to move inside the manifold here would be very complex and comprehend many variables. On the other hand, the encoder can make inference by interpreting the data at an abstract level, where the manifold can be described in a much lower dimensionality. In the learned representation, the manifold gets flatten out and disentangled. What many algorithms learn is a representation of the data points on (or near) the manifold, sometimes also called embedding. Instead than learning the embedding directly for each point, an autoencoder learns to do the mapping between the two spaces.

One possible benefit of this is the ability to perform operations with a high-level concept of the data in a straightforward way. Indeed it is possible to generate points in the manifold, by generating them on the learned representation and projecting them back with the decoder. The hope in doing so is that some of the dimensions in the latent-space representation correspond to meaningful attributes. For example, for images of faces, those might be gender or age, like in the example in fig. 2.10. Since the space has been "flattened" the distribution in the high-level representation is occupying a convex set. If we take two points in a convex set and interpolate between them, those new points are still part of the set. This nice property can be used to generate samples with specific properties by performing trivial operations in the abstract space. It is also possible that points belonging to two different classes are more easily separable in this new representation.

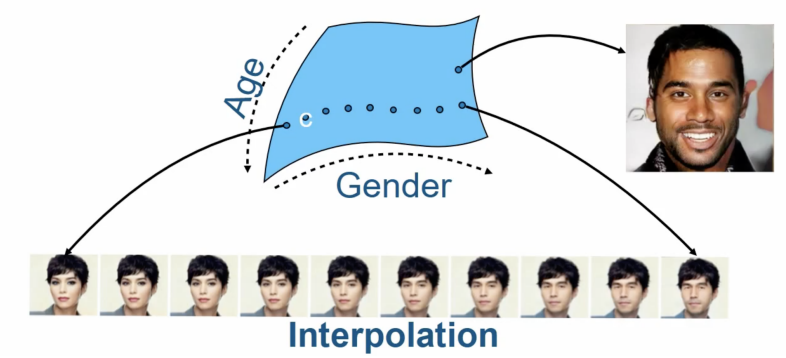


Figure 2.10. An example of image interpolation on the data manifold projected into the learned high-level space. With faces, different directions might correspond to attributes like age or gender. Interpolation is possible since the representation is occupying a convex set.

Autoencoders are arguably the most powerful tool in representation learning, and they have also been extended to the generation of samples, with variational autoencoders, that will be discussed later in this thesis.

2.2.4 Sharing knowledge

The ability to share knowledge between different settings is a crucial skill for an intelligent machine. Imagine if a child that learns how to draw with pencils would need to learn everything from scratch when drawing with markers. Intelligence means also understanding how to solve a problem reusing the experience we already have on similar problems. As humans, we do this by building analogies between concepts and eventually correcting our knowledge basing on a trial and error process that provides us feedback. As a matter of fact deep learning models reach human-level performances, but they always need a significant amount of data. The idea of sharing knowledge aims at reducing this weakness. The concept has been formalized into three different problems: transfer learning, domain adaptation, and multi-task learning. In both transfer learning and domain adaptation, the goal is to exploit what has been learned in one setting to improve generalization in another setting. More formally if we learn to describe a data distribution P_1 or we are able to perform inference on it, we want to be able to use that knowledge to tackle also the same problems on distribution P_2.

While in a typical scenario, train and test data should come from the same distribution, with transfer learning, they come from different distributions. The goal is to reduce the amount of annotated data needed for a new task. What we want is to have a representation of the data that can be shared between tasks and eventually fine-tuned for each one. If we consider image classification, for example, many visual categories share low-level concepts like edges, corners, shapes and shadows. It is, therefore, reasonable to start with a representation learned in the task of recognizing cats to recognize other kinds of animals. In practice, for deep learning, the simplest way to do transfer learning is to use pre-training: the parameters of a model trained on task A are reused as a starting point to learn task B, on data that looks similar. Pre-training is a standard procedure in computer vision today. Pre-training can be supervised or unsupervised. In the former case, the representation will be more biased towards a specific task, while in the second one, it will have more generic properties. Sometimes after choosing the initial weights pre-training, the first layers of the network are frozen, and the optimization focuses only on the last layers. This is because the first layers discover features that are more generic and valid for many datasets. Another possibility is to use both source and target samples to train the model but to weight them differently. In terms of features, we can also find a common space where source and target distributions are close, and project the data in this new space.

More complex approaches to transfer learning have also been proposed, like the multi-modal knowledge transfer by Tommasi et al. [12]. The authors propose a method to transfer the knowledge learned previously in source categories, to newly target object with only a few examples available. The algorithm provides guarantees on the learning performance by selecting the best source distribution from where to transfer and how much. The choice is based on solving a convex optimization problem which ensures minimal error on the available training set. In practice, part of the knowledge needed to discriminate a new and rare (for which a few training examples are available) object is transferred from the one acquired with many similar objects.

In the related task of domain adaptation (DA), we try to learn a predictive model in the presence of a shift between training and test distributions. So the task remains the same, but the input distribution is slightly different. As an example, say we have a labeled dataset of animal pictures (source domain) and an unlabeled dataset of drawings of animals (target domain). We want to be able to train a classifier on the source dataset and make it perform well also on the target dataset. Many DA methods try to build a mapping between the source domain, used at training time, and the target domain, used at test time. This mapping is then exploited when learning to extract the features for classification. This mapping can also be learned if the target domain data are fully unlabeled (unsupervised DA), or if there are a few labeled examples (semi-supervised DA).

The popular DA method proposed by Ganin and Lempitsky [13] (known as DANN or RevGrad) tries to embed domain adaptation into the feature learning process by using adversarial training. In the end, these features will be both discriminative and invariant to the change of domains. To implement this, the method uses a deep neural network (CNN for images), where the first layers are used to extract generic features. This output is fed into two branches on the network; the first is trained as a label predictor and the second one as a domain classifier. The parameters of the label predictor are optimized in order to minimize training set error. Instead, the parameters of the feature extractor are optimized to minimize the loss of the label classifier and maximize the one of the domain classifier. The first part of the optimization is necessary to learn discriminative features while the second part makes them domain-invariant. The architecture of DANN is shown in figure 2.11. The maximization of the domain classifier loss in the feature extractor is achieved by using a gradient reversal layer, that inverts the sign of the gradient of the loss with respect to the parameters. During the forward pass this layer is transparent.

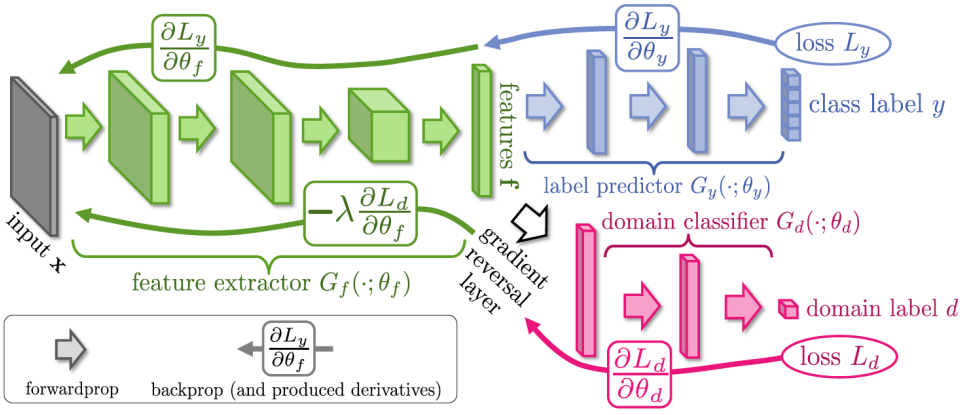


Figure 2.11. The architecture of the RevGrad domain adaptation method. With this scheme the learned representation will be both discriminative and feature invariant [13].

The third scenario that involves sharing knowledge between tasks is multi-task learning. The goal is to have a single model that tries to solve multiple related tasks at the same time. An example where this is used is in self-driving cars; in this case, we need a system that based on images is able to recognize pedestrians, other cars,

road signs and maybe perform segmentation. While we can try to address all those tasks with different models, it is clear that some low-level features might be shared, since they are common to all tasks. Given a certain amount of data for each task, the low-level features will benefit from having more data, and will also experience a regularization effect. It has been proven that overfitting is significantly reduced with multi-task learning; in particular, the risk of overfitting is $O(N)$ where N is the number of tasks, while the task-specific parameters experience a greater risk [14].

In practice, for deep learning, two possible network architectures that exploit multi-task learning are shown in figure 2.12. The first option, called hard parameter sharing, is to have the lower layers of the network shared between all the tasks. For each task that there will be a following branch of the network that generates more specialized and high-level features. The second option, soft parameter sharing, is to have one separate network for each task and to try to have the parameters of lower layers take on similar values. This can be done by minimizing a distance metric between the parameters, which will be an additional regularization constraint. In the end, multi-task learning encourages the model to prefer a representation that also benefits other tasks, which can also be seen as a continuous bi-directional transfer learning.

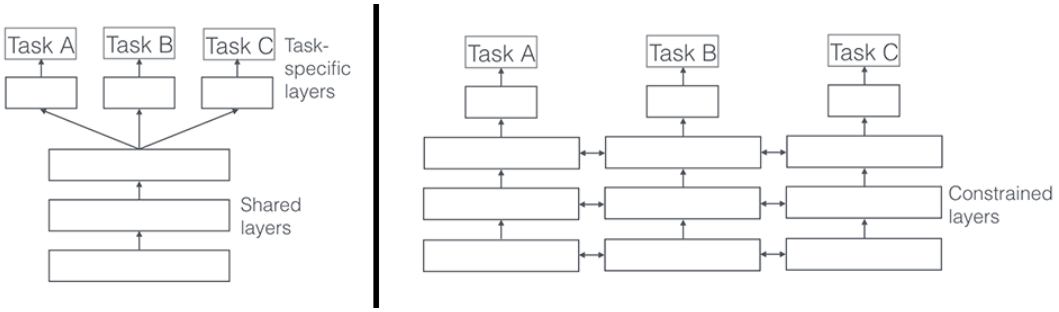


Figure 2.12. Two possible multi-task learning network architectures: hard parameters sharing (left) and soft parameter sharing (right) [15].

To summarize, the methods that allow the sharing of knowledge are all based on representation learning. They suppose that there exist features and underlying factors that are present in more than one learning problem and try to extract them.

2.2.5 New approaches

Some more recent approaches related to representation learning are here illustrated. In the method named MINE [16] by Belghazi et al. the authors argue that "the estimation of mutual information between high dimensional continuous random variables can be achieved by gradient descent over neural networks". Mutual information is a way to quantify the dependence between two random variables, and instead of capturing only linear dependencies like correlation, this is a true measure of the overall dependence. More formally, given random variables X and Z it can be defined as:

$$I(X, Z) = \mathbb{E}_{\mathbb{P}_{X,Z}} \left[log \left(\frac{p(x, z)}{p(x)p(z)} \right) \right]$$

The authors found an estimator for mutual information that is trainable via back-propagation and proved that it is linearly scalable in dimensionality as well as in sample size. This estimation can be used both to minimize or to maximize mutual information. In terms of applications, they have been able to reduce the mode-dropping problem in generative adversarial networks (GAN) (fig. 2.13) and improved inference and reconstructions in other adversarially-learned inference methods. Being able to quantify mutual information in deep learning models can allow building regularizers based on this value, which can lead to more disentangled representations.

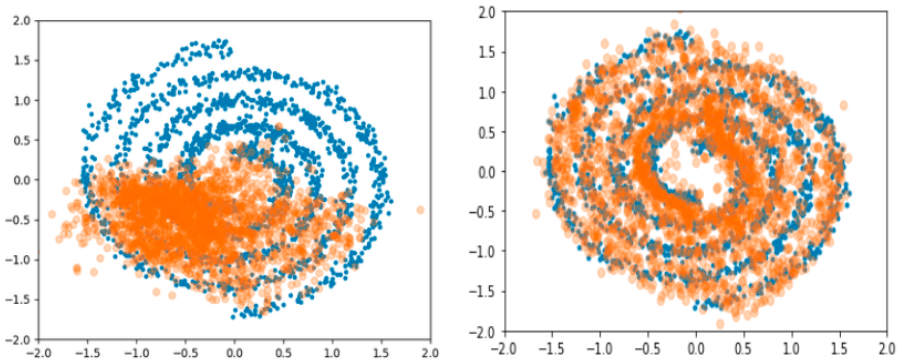


Figure 2.13. The phenomenon of mode dropping in GANs: blue points indicate the distribution of the original dataset, the orange ones the generated points. Usually (left) the generated data tend to cover only a small portion of the dataset, while this phenomenon is greatly alleviated adding the MINE regularizer (right) [16].

In the Deep InfoMax method proposed by [17], the authors (Hjelm et al.) investigate unsupervised representation learning by estimating and maximizing mutual information. The maximization of mutual information is intended between the inputs and the outputs of a deep encoder network, and its estimation is based on the previously described method. The important finding is that MI estimation is often insufficient for learning useful features, depending on the task. The new idea is to leverage the local structure of data points to create an objective that can improve the value of the representation. This objective is based on maximizing the average

MI between the extracted features and local regions of the input (like patches of an image). Globally, MI maximization performs better for reconstruction tasks, while local maximization is more suited for discriminative tasks. Furthermore, to include characteristics like independence, the matching of a prior representation is introduced. Two new measures of representation quality are also presented. Experimental results show that Deep InfoMax outperforms many unsupervised learning methods and performs comparably to some standard fully-supervised classifiers.

Chapter 3

Generative Approaches

3.1 Generative and discriminative approaches

A decision problem can be solved in various ways; more specifically, we will highlight two strategies. The first involves solving an inference problem, where training data is used to learn a model for the posteriors $p(C_k|x)$, where C_k is one of the possible classes and x is a vector in the input space. The following decision phase will use the learned posteriors to give an optimal result, based on a manually crafted decision function. A second possibility would be to solve both problems together, by learning directly a function that maps an input x to a class C_k. Within the first strategy, two ways are possible. The first is to model the class-conditional densities $p(x|C_k)$ for each class C_k and the prior class probabilities $p(C_k)$. In the end, assuming to know the data densities $p(x)$ it is straightforward to obtain the posteriors $p(C_k|x)$ using Bayes' theorem:

$$p(C_k|x) = \frac{p(x|C_k)p(C_k)}{p(x)} \quad (2.0)$$

where $p(x) = \sum_k p(x|C_k)p(C_k)$ (2.1).

Approaches that model the distribution of inputs as well as outputs are known as **generative models**. Having learned such a model, it is possible to sample from them to obtain synthetic data points in the input space. Generative models are very onerous since they involve finding the joint distribution over both x and C_k. When training data has high dimensionality, we would need an extensive training set to be able to determine the class-conditional densities with a good accuracy. An advantage of this approach is that it also allows to quickly compute the marginal density of data p(x) with (2.1). This can be useful to detect new data points that have low probability under the model, for which the prediction may be of low accuracy (outlier detection) [18, chap 1].

On the other hand, it is possible first to solve the inference problem of determining the posterior class probabilities $p(C_k|x)$ and then use decision theory to assign a new data point to one of the classes. Approaches based on this paradigm are known as **discriminative models** since they model the posterior probabilities

directly. This is the most straightforward way when we only care to make classification decisions, even though recent studies highlighted ulterior benefits in using generative or hybrid approaches.

As a guideline, generative models fit very well when handling cases of missing variables, and they offer better diagnostic on the results. By choosing the prior distribution, it is also easy to add prior knowledge about the data and therefore ease the task of learning when not enough labeled data are available. In this case, though, one should be confident in the prior distribution that is using, because it can also introduce errors. On the other hand, discriminative models offer great performances when the underlying distribution of data is really complicated (e.g. texts, images, graphs).

In their 2002 work [19] Ng and Jordan make an extensive comparison between discriminative and generative classifiers, both in a theoretical and in an empirical way, comparing the performance of logistic regression and naive Bayes classification algorithms. The authors challenged the common belief that discriminative classifiers are almost always to be preferred. Several reasons push the usage of discriminative models, like the argument by V. Vapnik saying that "one should solve the problem directly and never solve a more general problem as an intermediate step", referred to directly modeling the conditional $p(C_k|x)$. From this point of view, the benefits of generative models seem to be related only to computational issues or handling missing data.

The authors considered the naive Bayes model (for both discrete and continuous inputs) and what can be considered its discriminative counterpart: logistic regression for linear classification. The study proved that the generative model does have a higher asymptotic error (as the number of training examples becomes large) than the discriminative one. The exciting finding is that a generative model may approach its asymptotic error much faster than the discriminative counterpart, possibly with a number of training examples that is only logarithmic, rather than linear, in the number of parameters. This behavior is showed in the experiments in figure 3.1.

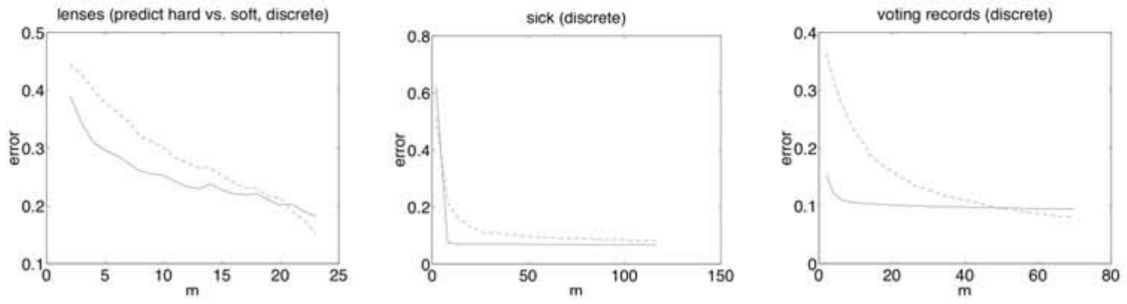


Figure 3.1. Results from three experiments from different datasets from the UCI Machine Learning repository. The plots show classification error vs the number of training samples (m) for naive Bayes (solid line) and logistic regression (dash line). Results are averaged over 1000 random train/test splits. The faster convergence of the naive Bayes to is clearly depicted here [19].

In [20] Bishop and Lasserre also discuss the differences between these two approaches and how to get the best of both worlds. The authors first underlay the

advantage of using generative methods, since they can also exploit unlabelled data, which are much more readily available. While it is possible to improve generalization performance of generative models by training them discriminatively, they would no longer be able to leverage unlabelled data. In an attempt to gain benefits from both approaches, the authors propose a heuristic procedure which interpolates between these two extremes. The interpolation is done by taking a convex combination of the generative and discriminative objective functions.

In the work, it is argued that generative and discriminative models correspond to specific choices for the prior over parameters and that "a discriminatively trained generative model is fundamentally a new model". The proposed approach is a new and generic way of interpolating between generative and discriminative extremes through alternative choices of prior. This new framework is shown using both synthetic data and a practical example in the domain of multi-class object recognition. Finally, the authors propose a method to automatically find the best trade-off between the two approaches. The reported experiments confirm that when labeled training data are limited, the maximum performance is reached with a balance between the purely generative and the purely discriminative models.

3.2 Generative models

Other than classification, generative models can be used directly for the task of generating new samples from the same distribution of some given training data. More formally, given a training data distribution $p_{data}(x)$, the goal is to learn a distribution $p_{model}(x)$ as similar as possible to the first distribution. The task is a particular case of density estimation, which is a core problem in unsupervised learning. There are mainly two ways of solving these problems. It is possible to do explicit density estimation, where we explicitly define and solve for $p_{model}(x)$, or one can do implicit density estimation where the goal would be to learn a model that can sample from $p_{model}(x)$, without explicitly defining it. Generally speaking, the training of a generative model would optimize a score that measures the difference between the generated distribution and the true data distribution. For example, the KL-Divergence can be used to measure the dissimilarity of two probability distributions. The applications of generative models also include tasks like super-resolution, where the goal is to artificially enlarge the size of an image or colorization of black and white images. If applied to time series, generative models can be used for simulation and planning of the future.

Various kind of generative models are available, all based on the principles explained above. The three most popular approaches are:

- **Generative Adversarial Network (GAN)** [21]: composed of two neural networks which are trained with adversarially against one another. A generative network tries to generate data similar to the input data distribution. In this way, it will try to fool the discriminative network, whose job is to distinguish between original and generated data. As training goes on, hopefully, both networks will improve, with the result of having a generator that exactly reproduces the true data distribution. For images, GANs currently generates the best quality samples, but they are difficult to optimize, due to problems of training instability.

- **Variational Autoencoder (VAE)**: is a neural network model based on an autoencoder with sampling in the latent space, which explicitly models the probability distribution we are trying to obtain. It can be considered a probabilistic graphical model with the goal of maximizing a lower bound on the log likelihood of the data. When working with images, generated samples tend to be blurry.

- **Autoregressive models**: they use a neural network that models the conditional distribution in terms of a spatial relationship between sub-portions of an input data point. These approaches model explicitly a distribution governed by a prior imposed by the model structure. An example of this category is the PixelRNN [22] algorithm, which works with images and models the conditional distribution of every individual pixel given nearby pixels. PixelRNN have a very stable training process and provides good quality data, however the sampling process can be inefficient.

A map that tries to classify the various generative models available today is shown in fig. 3.2. Between those models, VAE and the ones that explicitly model a parametric representation of a probability distribution function are the ones the were developed first. These models can be trained by simply maximizing the log likelihood of the distribution. Since complex data distributions of real data have intractable likelihood functions they require various approximations. As will be illustrated later, VAE treats an approximate density through the optimization of a lower bound. Another very successful algorithm in this category is Deep Boltzmann Machine [23]. These difficulties motivated the development of generative models whose goal is directly the one of generating samples from a certain distribution, without explicitly representing the likelihood. The most famous in this category are GANs and Markov Chain based models like Generative Stochastic Networks (GSN) [24]. Those models can be trained with standard backpropagation and without making approximations. GANs extend the GSN idea but remove the Markov chains used there.

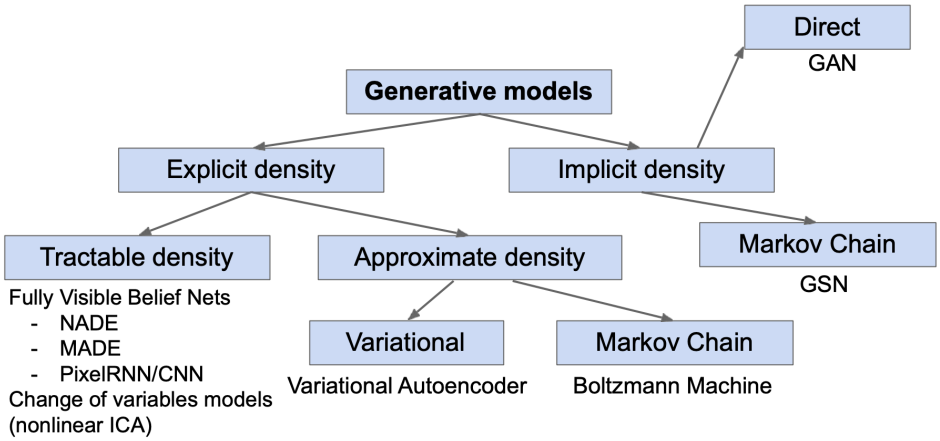


Figure 3.2. Deep generative models map [25]. These approaches can be differentiated in how they model the likelihood: on the left branch the ones that explicitly express it and try to maximize it and on the right the ones who interact with it indirectly. For example, for GANs, the training algorithm considers only the quality of the generated samples. Between the explicit ones, it is possible to distinguish the one that work with a tractable density, and the ones that work with an intractable one and therefore have to approximate it.

3.2.1 Boltzmann machines

Boltzmann machines have been introduced to learn generic probability distribution functions over vectors. They are tailored to work with binary vectors, in which each component can take only a value between two discrete ones. Training a Boltzmann machine involves adjusting its parameters such that the represented probability distribution fits the training data as well as possible. It is usually considered a generative model, so it allows sampling from the learned distribution, more specifically to a marginal distribution of interest. For example, one can fix the units corresponding to a partial observation and sample the remaining ones to complete the input piece. That being said, the model can also be used to classify new samples by training it to learn the joint probability distribution of the inputs and the corresponding labels, fed together in the input units.

Given a training set of n-dimensional points, the joint probability distribution over the observed variables is modeled as follow:

$$P(\boldsymbol{x})\frac{exp(-E(\boldsymbol{x}))}{Z} = \frac{exp(\boldsymbol{x}^T\boldsymbol{U}\boldsymbol{x} + \boldsymbol{b}^T\boldsymbol{x})}{Z}$$

Where $\boldsymbol{x} \in \{0,1\}^d$, Z is a normalization function that ensures that $\sum_x P(\boldsymbol{x}) = 1$ and $E(\boldsymbol{x})$ is known as an *energy function*. The trainable model parameters are spread between the weight matrix $\boldsymbol{U}$ and the bias vector $\boldsymbol{b}$. With the energy function written above, the Boltzmann machine can express a probability density as a linear relationship of the input vectors. By adding some latent variables, the model becomes similar to a multi-layer neural network and will be able to approximate any probability mass functions over discrete variables. This joint probability is known in Physics as the Boltzmann distribution and it is used in quantum mechanics to compute the likelihood that a particle can be observed in the state with energy E. From this it derives the name of the model.

Maximum likelihood estimation is usually employed to train Boltzmann machines, but since they have an intractable partition function (Z), the gradients must be approximated. A more common variant of this model is the Restricted Boltzmann Machine (RBM), invented by P. Smolensky in 1986, which is an undirected model containing a single layer of latent variables. No connections between units of the same layer are permitted. Usually, each unit of the input layer is connected to every unit of the hidden layers, but it is also possible to have a sparse structure. If there is more than one hidden layer, it is called a Deep Boltzmann Machine (DBM). RBMs are almost straightforward to train, compared to other deep models, since it is possible to compute $P(\boldsymbol{h}|\boldsymbol{v})$ in closed form exactly. Training DBM instead is much more difficult since both the partition function and the inference are intractable.

While this model is not frequently used today, in 2007, Salakhutdinov et al. [26] scored a state-of-the-art performance on the Netflix collaborative filtering prize using a two-layer RBM. Given a training set of per-user movie reviews, the goal is to predict a new movie's rating that a specific user has not yet rated. For example, considering movies, one can think that each movie can be explained in terms of some latent factors, that can ideally be associated with the genre, the actors who play in

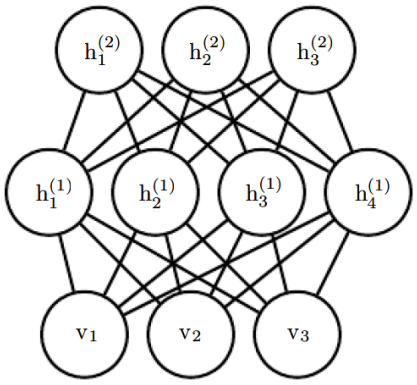


Figure 3.3. A deep Boltzmann machine, an undirected graphical model with two hidden layers $(h^{(1)}, h^{(2)})$. The connections are only between units of different layers, in this case each unit is connected to every units of the following layer. The structure of a this model looks like a feed-forward neural network, but the output layer has a different function, since predictions are made in a different way [4, Chap. 20].

it, or the year it was released in. In the same way, users might be identified by the movies they prefer or watch more frequently. One can then craft a model that tries to associate users to movies based on these underlying factors. A model like this can be trained using data about user's movie preferences and the corresponding collaborative movie tastes of all users. Since RMBs are designed for binary inputs, one can input user preferences as simply like/dislike. Passing this data to the input layer, the model can discover latent factors in the data that can explain the movie choices. In particular, each hidden layer will represent a latent factor. In [26], the authors trained the model with over 100 million user/movie ratings. The results outperformed the best-tuned Singular Value Decomposition (SVD) models, and when combined, the two models achieved a record error rate for the time.

3.2.2 Pixel Recurrent Neural Networks

Among the generative models applied to images that explicitly deal with a probability density function, we find the Pixel Recurrent Neural Networks or PixelRNN, introduced by van den Oord et al. [22] from Google DeepMind in 2016. When dealing with natural images, we have the advantage of having a practically endless amount of data, which can be leverage as-is by unsupervised models. On the other hand, images are a very high-dimensional and structured kind of data, therefore building a good estimator for their distribution is a very challenging task.

While approaches like VAE work with an intractable density function and must make approximations, the idea behind autoregressive models like PixelRNN is to factorize the objective distribution into a product of conditional distributions. In the case of images, we are talking about obtaining the joint distributions of the pixels by casting it as a product of the conditional distribution of each pixel, given all the previous ones. In this way, the problem is converted into a sequence problem that can be solved with a sequence model like the famous Recurrent Neural Network (RNN) architecture. RNNs offer a compact and shared parametrization for a series of conditional distributions, as a sequence of image pixels. Both the PixelRNN and the related PixelCNN do not introduce independence assumptions like in latent variable models. They instead capture the dependencies between pixels and RGB colors within each pixel.

Compared to VAE and GAN, autoregressive models have the advantage of having a very simple and stable training process, while for example, GANs are usually challenging to optimize due to unstable training dynamics since they are trying to find a Nash equilibrium with a trial and error approach. They also currently give the best log likelihood between the three, and offer a tractable likelihood that can also be used for additional tasks, like such as compression and probabilistic planning and exploration. On the other hand, they are relatively inefficient during sampling, for reasons that will be clarified later. In terms of image quality though, GANs have the best perceived quality. Another note is on the type of data that can be used: autoregressive models are more flexible, while it would be difficult to generate discrete data with a GAN. Efforts are being made to incorporate both classes' advantages in a single model, but it is still an open research area.

The PixelRNN model is based a two-dimensional RNN, composed of up to twelve Long Short-Term Memory (LSTM) layers. With PixelRNN the image is treated as a sequence of pixels going row by row, or according to other patterns described later. Each pixel is dependent on the previously generated pixels, as expressed by the following equation:

$$p(\boldsymbol{x}) = \prod_{i=1}^{n^2} p(x_i | x_1, ..., x_{i-1})$$

where $p(\boldsymbol{x})$ is the probability of the pixel values of an image $\boldsymbol{x}$ represented as an n by n vector (row by row for example).

The principle behind an RNN is that each unit should depend not only on the current input but also on previous input values. This is implemented by introducing a loop on the network that propagates forward a signal from previous time steps.

This design allows modeling ordered sequences of data, where the result would depend on the trend of the entire sequence. Such a network structure is ideal for treating things like stock prices, speech, and text, and it has been widely used in tasks like speech recognition, language modeling, translation, and image captioning. An illustration of a basic RNN architecture is shown in figure 3.4. The simplest training algorithm for an RNN is called backpropagation through time (BPTT), and it includes a further step compared to normal backpropagation used to train feed-forward neural networks. The error signal obtained from the loss computed in the output for each pass must also be propagated backwards to each time step unit. This is because the parameters are shared between time steps, and the gradient at each output also depends on the previous steps. The loss function is computed by accumulating the losses of each time step. This algorithm also has its weaknesses, like the vanishing gradient problem, for which variations have been proposed.

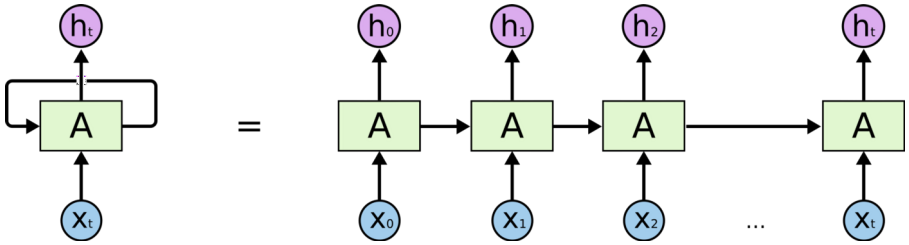


Figure 3.4. This scheme represents the architecture of a Recurrent Neural Network. The loop inside the network allows information to persist (left image). On the right, the same network with the loop unrolled. The node (A) can be a neural network neuron or an entire layer. Input x_t is the data point at time t and h_t denotes the corresponding output. At each time step, the input is dependent on the current input point and on the state computed at the previous time step. Therefore, the output sequence is computed sequentially and cannot be parallelized.

LSTM networks are a special kind of recurrent neural network which have achieved most of the exciting results in the sequence modeling field. It can be considered an improved version of RNN since it performs better almost every time. When modeling sequence of data it might be useful to pick both short term and long term dependencies, like a long sentence where the final word's meaning is related to the entire phrase. Unfortunately, RNNs have problems picking up long-term dependencies, even if theoretically they are allowed to. This problem has been studied and explained by Hochreiter [27] in 1991 and Bengio et al. [28] 1994. In the latter, the authors affirm that the gradient descent training method "becomes increasingly inefficient when the temporal span of the dependencies increase". The LSTM architecture has been proposed exactly to address this limitation by Hochreiter and Schmidhuber in 1997 [29]. An LSTM cell takes the place of the RNN cell inside the network, but inside it has a different structure: there are four layers connected in a specific way. This structure is illustrated in details in figure 3.5.

Going back to PixelRNN, it uses LSTM units and convolution to compute at once all of the states along one spatial dimension of the image. More specifically, the Row LSTM layer applies the convolution row by row, while the Diagonal BiLSTM applies the convolution along the diagonals of the image. Some residual connections

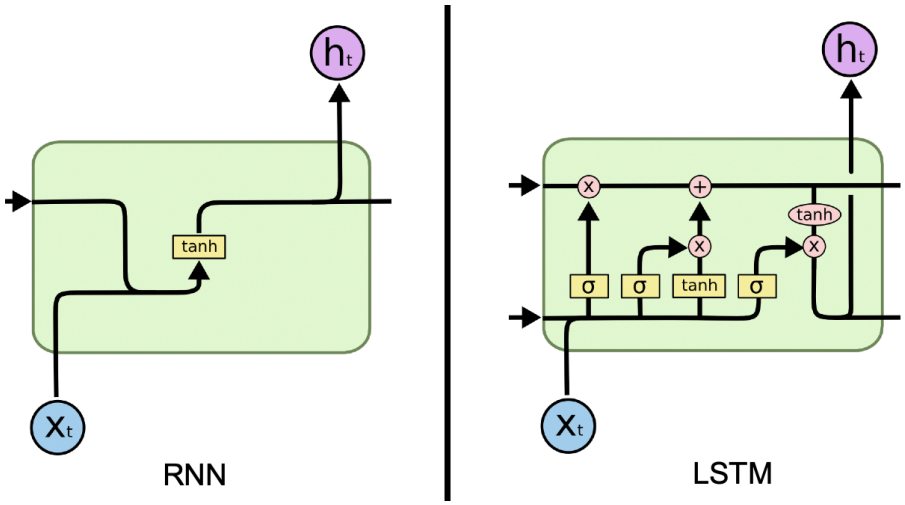


Figure 3.5. A comparison between a simple RNN cell and the classic LSTM cell. While the first can contain only a simple *tanh*() unit, four layers are used inside an LSTM cell. Furthermore the LSTM cell has two loop signals instead of one.

are also employed since they benefit the training procedure of the relatively deep architecture (12 layers). Residual connections are a mechanism to improve the training of very deep architectures developed by He et al. [30], within the so-called Residual Networks (ResNet in short). The authors also developed a simplified architecture called PixelCNN, which is a 15-layers network. CNNs are used as a sequence model with a fixed dependency range by applying masked convolutions. This model preserves the resolution of the input mage going throughout the layers and outputs a conditional distribution at each location [22]. Finally, a multi-scale version of PixelRNN has been proposed, composed by a regular-size and a few smaller versions of the model. The smaller versions should capture global features better, and the regular one captures more local features. Feature vectors are added together in the first part of the network and then fed to the remaining pipeline. In all architectures, pixels are modeled as discrete values using a multinomial distribution, which has shown better performances.

Figure 3.6 shows the sequential process of image generation with the PixelRNN model, which happens row-by-row in the left figure. In this case, each pixel is conditioned on all the previously generated pixels above and to the left of it. Both long and short term dependencies are carried out thanks to LSTM cells. In the middle figure, the multi-scale context pixel dependencies are depicted. Each pixel can also depend on the other subsampled pixels in all directions. On the right, the dependencies between colour channels are shown. On the first layer, for example, the green channel of one pixel is dependent both on the previous channel (red here) and to the context of all previous pixels. In subsequent layers, channels are also connected to their corresponding channel on the previous layer.

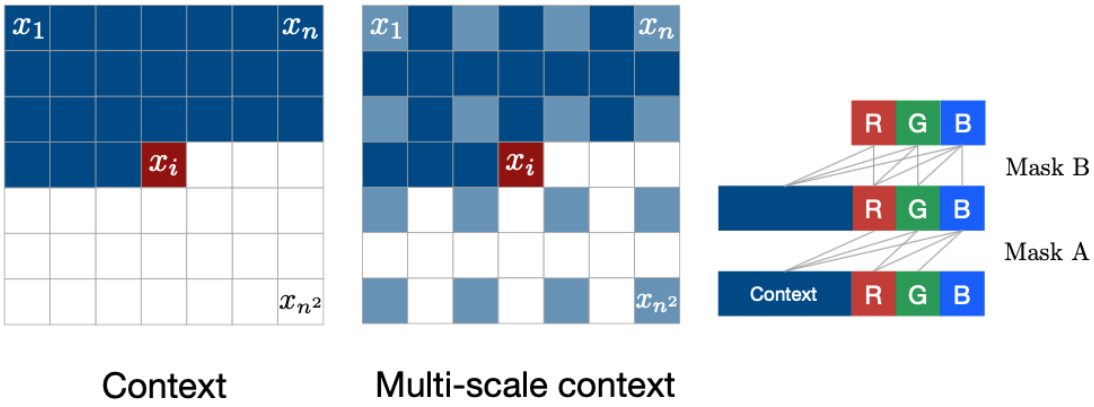


Figure 3.6. Diagrams of PixelRNN intra-pixel dependencies. On the left, the row-by-row case is shown, where each pixel depends only on the previously generated ones. In the center, the multiscale case is shown: each pixel depends both on pixels of the current image and on the subsampled version. On the right, the connections between color channels are shown [22].

PixelRNN has been extensively tested on benchmark datasets like MNIST, CIFAR-10 and the more complex ImageNet, which contains a huge variety of natural images. Excellent results have been obtained in terms of negative log-likelihood and visual quality. The model has also been tested on the task of image completion (fig. 3.7) achieving results that demonstrate its ability to model both spatially local and long-range dependencies.

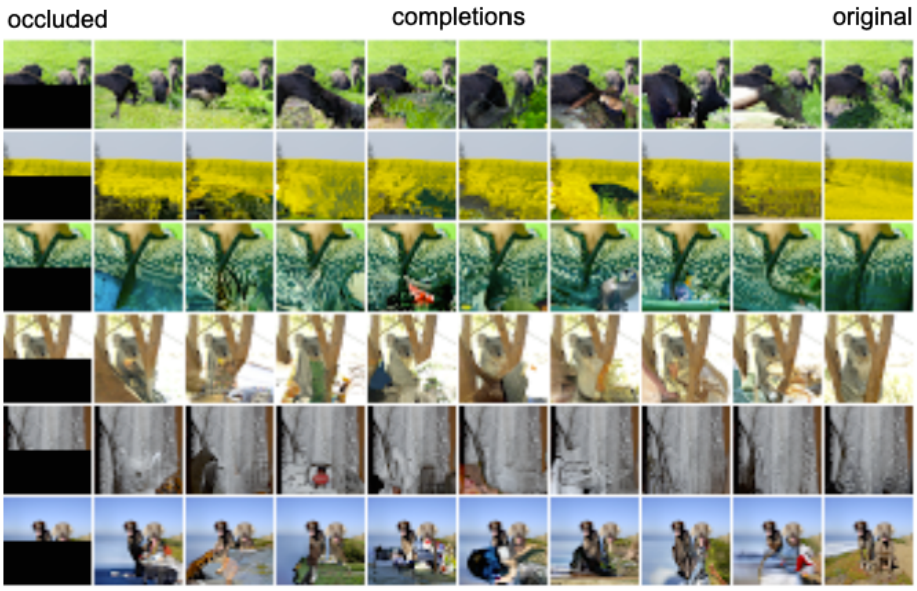


Figure 3.7. Example results of PixelRNN on the task of image completion on ImageNet [22].

3.2.3 Variational Autoencoders

We have already introduced the concept of autoencoder; an unsupervised approach for learning a lower-dimensional feature representation of a given data distribution. In an under-complete autoencoder, the hidden layer act as a bottleneck promoting the learning of a compressed representation. The reconstruction loss forces the latent representation to capture the most relevant information about the data as possible.

Once an autoencoder has been trained, the expanding part of the network can theoretically be used to generate new samples belonging to the distribution. In practice, this is very difficult since the latent space extracted is not regular in the sense that it depends a lot on its dimension and the encoder architecture. Without an additional mechanism, it is difficult to ensure that the latent space is compatible with a working generative process. By randomly sampling points on the latent space, the decoder will likely generate meaningless output samples. This behavior has to be expected since the model has been designed solely to encode and decode samples with as few loss as possible, no matter how the latent space is organized.

A variational autoencoder [31] can be defined as being an autoencoder whose training is regularized to avoid overfitting and ensure that the latent space has good properties that enable generative process. The model uses a learned approximate inference and can also be trained with standard gradient-based methods. The new model is now probabilistic, allowing to generate new samples that belong to the training distribution, and eventually control some meaningful features.

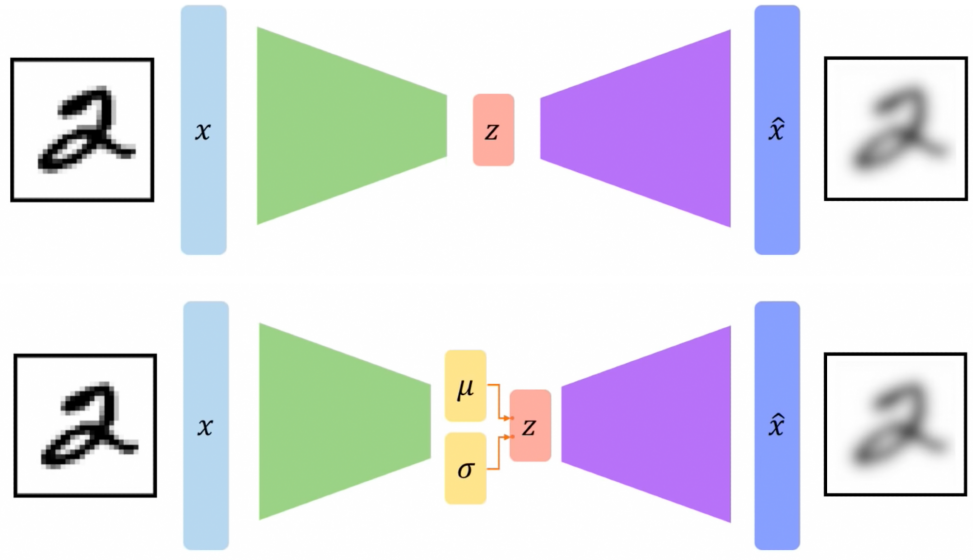


Figure 3.8. Schemes of the classic autoencoder network (above) and the corresponding variational autoencoder (bottom). In both cases an input, like an image of a handwritten digit, is fed into the encoder and the decoder reconstructs a sample that belongs to the same distribution. In the case of VAE the network learns the parameters of a latent distribution that allows the encoder to sample from it. (Source: MIT course 6.S191 Introduction to Deep Learning.)

The deterministic inner layer ($\boldsymbol{z}$) is here replaced with a stochastic sampling operation. Instead of learning the latent variables directly, for each variable we learn a mean ($\boldsymbol{\mu}$) and a standard deviation ($\boldsymbol{\sigma}$), associated to a reasonable prior distribution (fig. 3.8). What the encoder network computes is now the probability distribution $p_\phi(\boldsymbol{z}|\boldsymbol{x})$, while the decoder computes $q_\theta(\boldsymbol{x}|\boldsymbol{z})$, both parameterized by learnable parameters. The loss $L(\boldsymbol{\phi})$, reported below, includes a term that measures the difference between the generated output and the original one and a regularizer that tries to keep the distributions of the latent variable close to a prior $p(\boldsymbol{z})$. A common choice for this prior is a normal Gaussian. The encoder is therefore encouraged to center the latent variables evenly in their space. If this was not the case, the network could "cheat" by memorizing training data, which corresponds to clustering points in specific regions of the latent space.

$$L(\boldsymbol{\phi}, \boldsymbol{\theta}, \boldsymbol{x}) = ||\boldsymbol{x} - \hat{\boldsymbol{x}}|| + D(p_\phi(\boldsymbol{z}|\boldsymbol{x}) \,||\, p(\boldsymbol{z}))$$

The regularization term D is usually defined as the KL-divergence between two distributions.

An issue that arises with this architecture is the impossibility to back-propagate gradients through the sampling layers, since they are now a stochastic layers. As a matter of fact, the backpropagation algorithm requires to compute gradients at each layer and apply the derivative's chain rule. The key idea to solve this issue has been named **"reparametrization trick"** and consists in re-parametrizing the sampling layer such that the network can be trained with backpropagation end-to-end. More specifically we can consider the sampled latent vector $\boldsymbol{z}$ as a sum of a fixed mean vector ($\boldsymbol{\mu}$) and a standard deviation vector ($\boldsymbol{\sigma}$). Both will than be scaled by a random constant drawn from the prior distribution (ε) [32]. So, instead of having $\boldsymbol{z} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\sigma})$ we will have $\boldsymbol{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \varepsilon$, where $\varepsilon \sim \mathcal{N}(0, 1)$. The computational graph of the sampling layer is shown in figure 3.9.

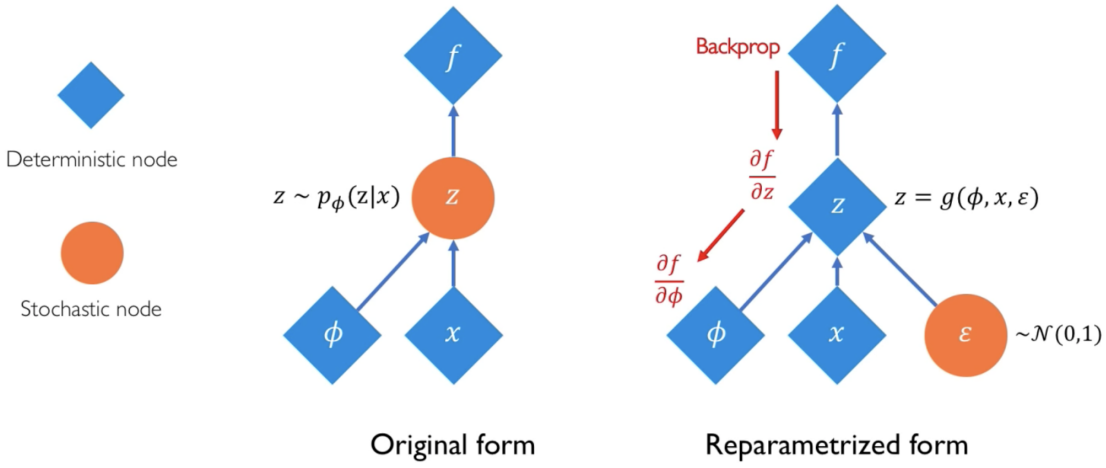


Figure 3.9. Computational graph of the sampling layer in a variational autoencoder. In particular the reparameterized form allows computing gradients and train the network with backpropagation. (Source: MIT course 6.S191 Introduction to Deep Learning).

From a theoretical point of view, the VAE loss presented here is an approximation of the quantity that we want to maximize, which is the log likelihood of the original data $p_\theta(\boldsymbol{x})$. This likelihood can be expressed as:

$$\log p_\theta(x) = \mathbb{E}_{z\sim q(z|x)}[\log p_\theta(x)] =$$

$$= \mathbb{E}_z\left[\log \frac{p_\theta(x|z)p_\theta(z)}{p_\theta(z|x)}\right] = \mathbb{E}_z\left[\log \frac{p_\theta(x|z)p_\theta(z)}{p_\theta(z|x)}\frac{q_\phi(z|x)}{q_\phi(z|x)}\right] =$$

$$= \mathbb{E}_z[\log p_\theta(x|z)] - \mathbb{E}_z\left[\log \frac{q_\phi(z|x)}{p_\theta(z)}\right] + \mathbb{E}_z\left[\log \frac{q_\phi(z|x)}{p_\theta(z|x)}\right] =$$

$$= \mathbb{E}_z[\log p_\theta(x|z)] - D_{KL}(q_\theta(z|x)||p_\theta(z))) + D_{KL}(q_\theta(z|x)||p_\theta(z|x))$$

On the expression above the last term is intractable, since it is not possible to compute $p_\theta(z|x)$ for all values of z. On the other hand, since it is a distance we know it is always greater or equal to zero. With this observation in mind, we decide to optimize the following **variational lower bound** instead, which is differentiable:

$$\mathcal{L}(\phi, \theta, x) = \mathbb{E}_z[\log p_\theta(x|z)] - D_{KL}(q_\theta(z|x)||p_\theta(z)))$$

$$\text{where } \mathcal{L}(\phi, \theta, x) \leq \log p_\theta(x)$$

therefore the gradient-based training algorithm will find the optimal parameters by solving:

$$\theta^*, \phi^* = arg\max_{\theta,\phi} \sum_{i=1}^{N} \mathcal{L}(\phi, \theta, x^{(i)}).$$

The variational autoencoder applied on images produces excellent results in terms of image quality and it also is a powerful manifold learning tool. The architecture of VAE, having both an encoder and a generator trained together naturally promotes to learn a predictable coordinate system. Results of a linear interpolation in a two coordinate system can be seen in figure 3.10. Even if the VAE model is somewhat elegant and between the state of the art solutions in terms of results, it suffers from a problem: the produced output images look a bit blurry. The precise cause of this issue remains unknown, but it has also been observed in other generative models that optimize a log-likelihood or the KL-Divergence between the model distribution and the training data one. Furthermore, VAEs tend to use only a small portion of the latent space dimensionality [4, Chap. 20].

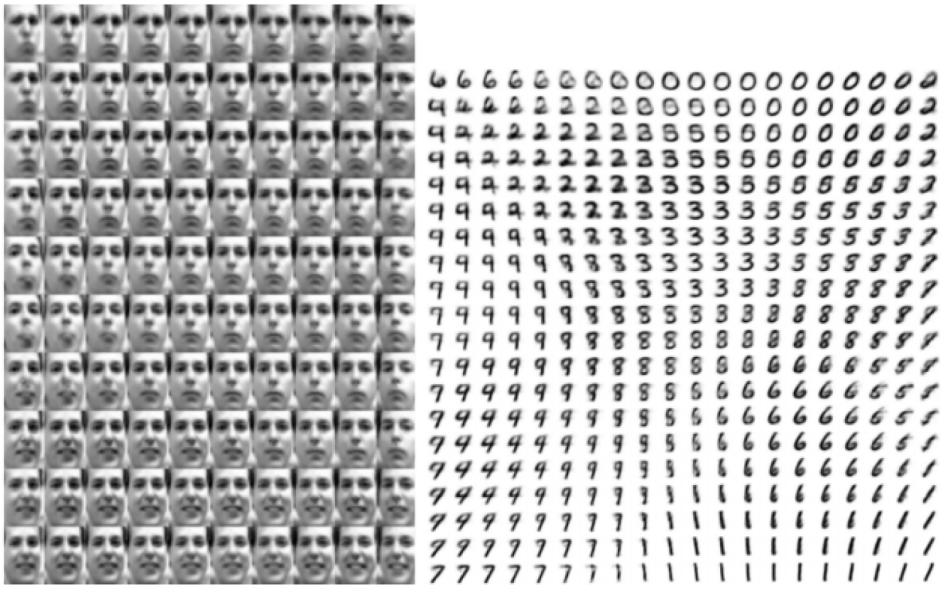


Figure 3.10. Visualization of the learned manifold, by interpolating two latent codes linearly and mapping the produced images on a 2D plane. On the left the Frey Face dataset, in which the model have learned to encode head rotation and face expression. On the right the MNIST digits dataset [31].

VAEs have many applications in the field of representation learning and are a great tool to build a meaningful latent space. One great example is the 2018 work by Gómez-Bombarelli et al. [33] where they used a VAE to model some chemical compounds. In particular, the network is trained on thousands of molecules of existing drugs and a continuous representation in the latent space is obtained. In the new continuous representation, it is possible to perform exploration near known molecules, interpolation between two of them or efficient searching using gradient-based optimization (fig. 3.11).

A popular variant of the presented model is the so-called **Beta Variational Autoencoder** (β-VAE) introduced by Higgins et al. [34], which focuses on the independence of the latent factor. The model tries to extract a disentangled representation, where a change in a single latent variable corresponds to a single change in the generative factors. As a matter of fact, the goal is to decompose the generative process into independent factors. This yields to a more interpretable and useful representation, which can also benefit downstream tasks. In practice the difference with respect to the original VAE is the introduction of a multiplier $\beta > 1$ in the KL-Divergence term of the loss function:

$$\mathcal{L}(\phi, \theta, x) = \mathbb{E}_z[\log p_\theta(x|z)] - \beta \cdot D_{KL}(q_\theta(z|x)||p_\theta(z)))$$

This factor penalizes the KL-divergence in the optimization, favoring the prior and encouraging the model to learn the most efficient representation of the data. Assuming that the real data has some conditionally independent underlying factors, a higher value of β encourages learning such representation. A too high value of the hyperparameter, on the contrary, will lead to a lower reconstruction fidelity.

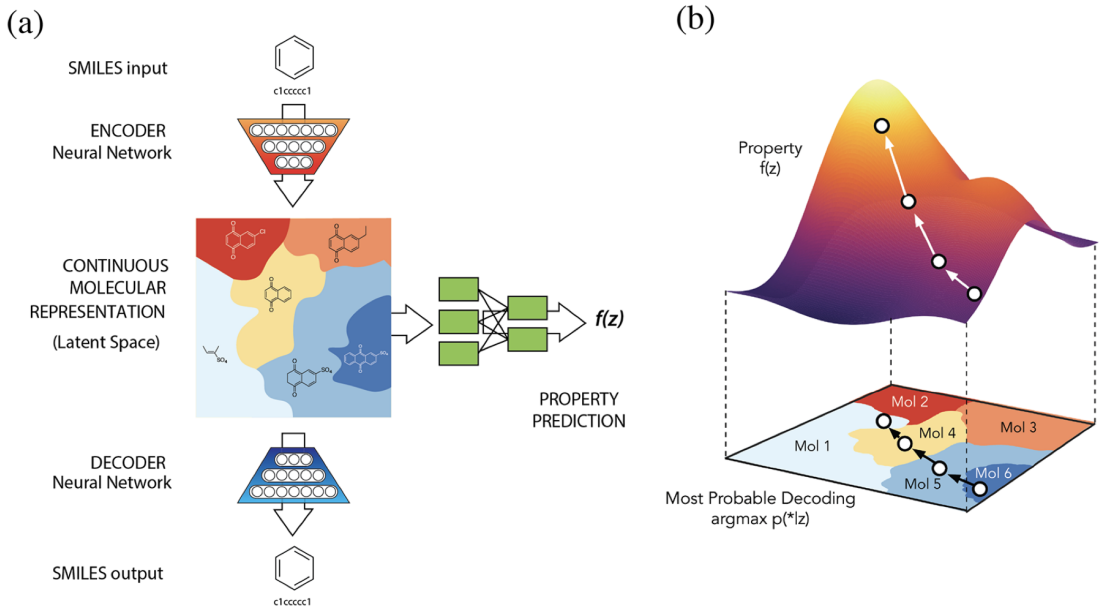


Figure 3.11. Applying VAE to chemical molecules. The original discrete space in which molecules are represented (SMILES notation) can be converted to and from a continuous space with a VAE. From the latent space representation an additional network tries to predict properties (fig. a). In the continuous space it is also possible to model a certain property and find new molecules with a specific characteristic thanks to gradient based methods (fig. b) [33].

3.2.4 Generative Adversarial Networks

A Generative Adversarial Network (GAN) is a deep generative model proposed in 2014 by Goodfellow et al. [21]. The method is based on two CNN networks trained together with an adversarial procedure; a generative model captures the data distribution, and a discriminative model estimates the probability that a sample comes from the real data distribution. During training, the generator and the discriminator roles are equal and opposite, in what is called a zero-sum game. GANs do not explicitly model the probability density of the training distribution; instead, they use a trainable model to directly evaluate the quality of the generated samples. The intuition behind GANs is that the two models can improve together by sharing feedback; this is also the reason why training needs to be balanced between these two parts, as we will discuss later. The training procedure will ideally stop when generated samples are indistinguishable from the real ones. In the proposed framework, the two multilayer perceptrons working together are called adversarial networks.

Compared to VAEs, they also pair a differentiable generator network with a second neural network. But differently from GANs that use a discriminative network, VAEs network perform approximate inference. Furthermore, since GANs requires differentiation through visible units, they cannot model discrete data. In contrast, VAEs requires differentiation in the hidden units, so it cannot have discrete latent variables.

Let us denote the generated samples as $\boldsymbol{x} = g(\boldsymbol{z}, \boldsymbol{\theta}^{(g)})$, where g is the generator network transformation, parameterized by the matrix $\boldsymbol{\theta}^{(g)}$. This network takes as input a random noise vector ($\boldsymbol{z}$), usually sampled from a normal or uniform distribution. This input gives the desired stochastic characteristic to the produced outputs. The discriminator produces a probability value $p = d(\boldsymbol{x}, \boldsymbol{\theta}^{(d)})$ corresponding to $\boldsymbol{x}$ being drawn from the training examples, which belong the real distribution.

During training each network tries to maximize its own payoff function: $v(\boldsymbol{\theta}^{(g)}, \boldsymbol{\theta}^{(d)})$ for the discriminator and $-v(\boldsymbol{\theta}^{(g)}, \boldsymbol{\theta}^{(d)})$ for the generator. At convergence the optimal generator would be:

$$g^* = arg\min_g \max_d v(g, d)$$

The payoff function is usually defined as:

$$v(\boldsymbol{\theta}^{(g)}, \boldsymbol{\theta}^{(d)})) = \mathbb{E}_{\boldsymbol{x} \sim p_{data}} \log d(\boldsymbol{x}) + \mathbb{E}_{\boldsymbol{x} \sim p_{model}} \log(1 - d(\boldsymbol{x}))$$

At convergence, the generated samples should look like they have been drawn from the training distribution, and the discriminator outputs 1/2 everywhere. The discriminator can be discarded if the purpose is just generation. The GAN paper authors also theoretically proved that the above training criterion for adversarial networks allows recovering the data generating distribution given enough capacity. When implementing the training, it is necessary to use an iterative, numerical approach which will alternate k steps of optimizing the discriminator and one step of optimizing the generator (usually k=1).

This algorithm is stable as long as the two networks learn at a similar rate;

otherwise, no useful feedback is produced and learning stops. This problem has been investigated first by Goodfellow [35] and is due to the non-convex nature of $v(g, d)$ when d and g are represented by neural networks. This issue may prevent the reaching of convergence which results in underfitting, a general problem of simultaneous gradient descent on two actors. The equilibrium point, in this case, is represented by a point that is simultaneously a local minimum for both player's cost functions. The author also introduced a new formulation of the GAN payoff function which performs better in terms of convergence. In this formulation, the generator tries to maximize the log probability that the discriminator makes a mistake. This is motivated by the heuristic that the derivative of the cost function remains large even when the discriminator outputs a strong negative signal. Today, training stabilization in GANs remains a critical point, but various techniques have been introduced to ease it.

The training algorithm of the original GAN (from [21]) is reported below. G and D indicate respectively the generator and discriminator network transformations. This algorithm allows to find the optimal generator g^*.

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**
  for k steps **do**
    • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
    • Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

  end for
  • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  • Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for
The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

The scheme of the GAN architecture is shown in figure 3.12. Real images are taken from the unlabelled training dataset. The binary switch before the discriminator input shows how the network is trained with both real and fake data in different steps.

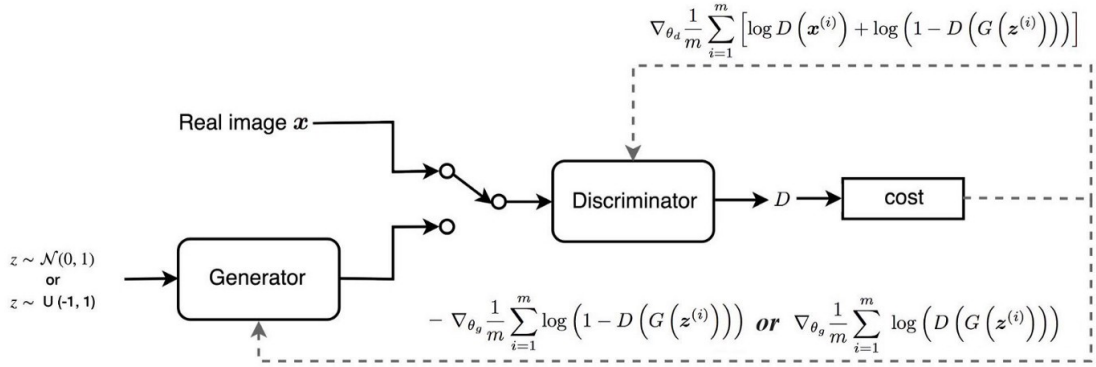


Figure 3.12. Scheme of the original GAN architecture. The generator and discriminator blocks are multi-layer perceptrons. The random noise vector z is drawn from a normal or uniform distribution. The two cost function to minimize during backpropagation are specified for both the generator and the discriminator.

In 2016 Radford et al. [36] introduced an improved version of GAN specifically tailored for images, called **DCGAN**. The authors proposed a set of improvements to the architecture to make training more stable. In addition, the quality of the features learned by the model also gets assessed.

The first improvement is to use strided convolutions on the discriminator and fractional-strided ones on the generator. These should be used instead of the usual deterministic spatial pooling functions that do not allow the model to learn its own spatial downsampling or upsampling. The second improvement regards removing fully-connected layers in deeper architectures, which provides improved training stability. The third suggestion is to use Batch Normalization layers [37], as in most of the modern CNN architectures, which also stabilizes learning by normalizing the input of each unit to zero mean and unit variance. This technique mitigates poor initialization of weights and helps the gradient flow through deep networks. Batch Normalization applied to every layer can cause instabilities, so the generator output layer and the discriminator input layer do not have it in DCGAN. In addition, DCGAN uses bounded activation functions like ReLU (Rectified Linear Unit) in the generator (except for the output layer which uses Tanh) and LeakyReLU in the discriminator. Such family of functions allows the model to learn more quickly to saturate and cover the entire color space of the training distribution.

To asses the semantic quality of the features extracted by the generator, the images corresponding to an interpolation of the input vector have been analyzed. Some of these are shown figure 3.13, where all images look plausible and there is a smooth transition between them.

A problem that can happen during generation with GANs is the so-called mode collapsing. A mode in a distribution is roughly speaking an area with a high concentration of samples - for example, a normal distribution as a single-mode

Figure 3.13. Images generated by the DCGAN model with a linear interpolation of the input noise vector, from left to right. In the third row the concept of "size of a window" seems to be expressed [36].

corresponding to its mean value. In a hyperspace defined by meaningful features of the data generating distribution, data will likely be clustered around dense areas. For example, consider the handwritten digits in a feature space defined by two features that correspond to the two coordinates of the points belonging to each digit. The numbers will be clustered around areas corresponding to digits with similar features, and we will likely have a 10-mode distribution. Mode collapse happens when the generator learns to "fool" the discriminator by producing examples from a single class or mode from the whole training dataset. This is unfortunate because, while we want the generator to produce samples that look as real as possible, we also care about covering all the possible modes in the original distributions.

In the original formulation of GANs, the Binary Cross-Entropy (BCE) loss is used. In the **Wasserstein GAN** paper [38] (WGAN) the authors found that this loss is prone to mode collapsing and vanishing gradient problems. This is due to the BCE loss outputting values between 0 and 1 and, as the discriminator learns, it will by pushing the values towards those limits. The authors of WGAN propose to use the Earth Mover's distance, which measures the difference between two distributions. The latter is not limited, so the cost will continue to grow regardless of how far the two distributions are. As a result, the gradient of the cost function will not approach zero, making the GAN less prone to vanishing gradient problems and mode collapse.

With the Wasserstein Loss (or W-loss) that approximates the Earth Mover's distance, training is defined as follows:

$$\min_g \max_c \mathbb{E}\, c(\boldsymbol{x}) - \mathbb{E}\, c(g(\boldsymbol{x}))$$

where c indicates the discriminator network function, here called critic since it no longer outputs values between 0 and 1. Since it is not bounded the critic is

allowed to improve without degrading its feedback to the the generator. In addition, for the W-loss to work well the critic network should respect the 1-Lipschitz continuity condition. A function meet this condition if the norm of the gradient is at most one in every point. This means that its slope cannot be too steep at any point, because the function cannot grow more than linearly. 1-Lipschitz continuity ensures that the W-loss is not only continuous and differentiable, but also that it does not grow too much and maintain some stability during training.

Two common ways of ensuring 1-Lipschitz continuity are weight clipping and gradient penalty. With the first technique, the weights of the critic network are forced to take values between a fixed interval. The clipping is done after updating the weights during gradient descent. The downside of weight clipping is that it could limit the critic's ability to learn, therefore slowing down learning or even avoiding it to find the optimal point. Furthermore, the clipping range corresponds to two more hyper-parameters to tune.

The gradient penalty method, instead, is based on a soft constraint that is implemented by adding a regularization term to the loss function. This term is crafted in a way that penalizes the critic when its gradient norm is higher than one. There is one hyper-parameter involved to control how much this constraint weight on the total loss. In practice gradient penalty has proven to work better than gradient clipping in most cases.

Classical GAN's generator produces a completely random sample every time, according to the random noise vector fed as input. Mirza and Osindero proposed an approach named **Conditional GAN** (or CGAN) [39] with the goal of allowing some kind of control on the generated images. In particular, the conditioning is done over a class label, therefore introducing supervision (or semi-supervision) of the training set. This provides a new degree of flexibility for GANs since they can be used to generate specific instances of a certain kind, an ability that is necessary in certain applications. The results of CGAN on the simple MNIST digits dataset are shown in figure 3.14.

In the CGAN architecture, the generator network receives as input a random noise vector and the class label as a one-hot encoded vector. The random noise is still necessary to allow some variance in the produced output. The discriminator network also needs to receive the class label since it will be able to perform better knowing this information. To the discriminator, which receives as input a matrix containing the image, the label is concatenated as one-hot encoded matrix. This matrix has the width and height of the input images and a number of layers equal to the number of classes. The full architecture is shown in figure 3.15. Other approaches for controllable generation are available, but they all deal with the problem of disentanglement of the internal features. The InfoGAN approach discussed in the next chapter is specifically crafted to deal with this issue.

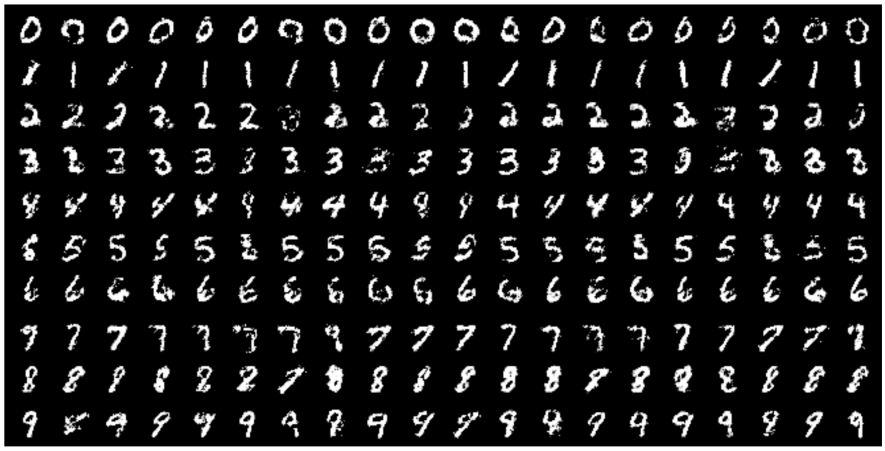


Figure 3.14. The output produced by CGAN trained on the supervised MNIST digits dataset. Each row is generated using a different labels, and the results are exactly as expected [39].

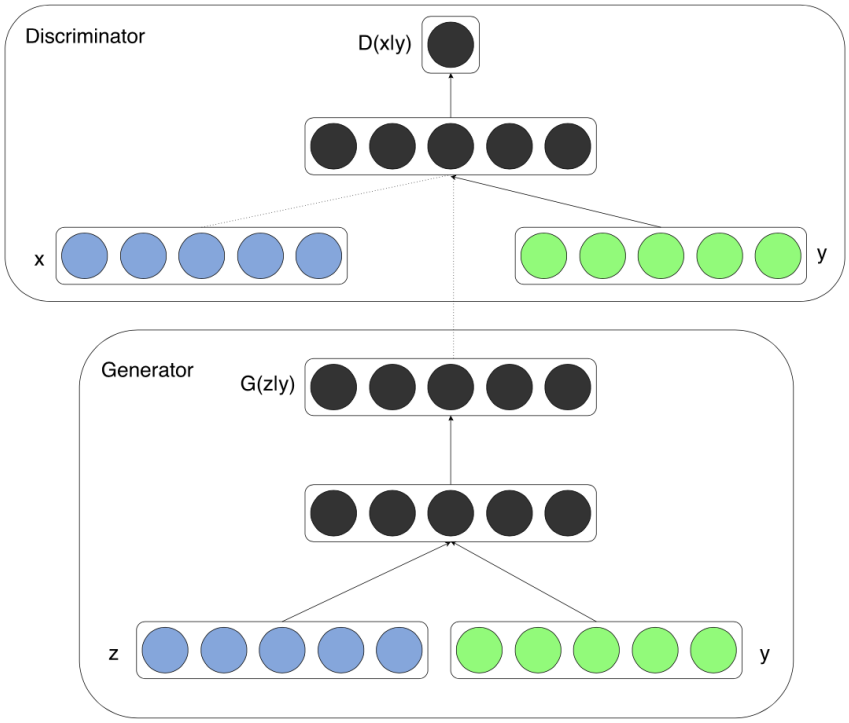


Figure 3.15. The CGAN model architecture. Both the discriminator and the generator receive an additional input volume that encodes the label with one-hot encoding (green block) [39].

Chapter 4

Experimental Study

4.1 InfoGAN

In a classic GAN the input to the generator is a random noise vector, of which each dimension does not correspond to a specific factor in particular. This does not offer any controllability to the produced output. The InfoGAN approach, proposed by Chen et al. in 2016 [40], tries to address this problem by promoting disentanglement in the generator's input representation. As discussed many times before, this conceptually corresponds to disentangling the data generating factors, therefore changing one dimension in the input vector would change a single perceived variation in the produced output.

The idea behind InfoGAN is to split the generator's input vector into two parts: a normal random part and a new latent code part. While the first portion is made to give the produced output a level of randomness, the second part is made to contain codes that should have a specific meaning. To make the codes meaningful, the concept of mutual information is exploited. Mutual Information (MI) is a measure of the mutual dependence between two random variables. More specifically, it quantifies the "amount of information" obtained about one random variable through observing the other one. The quantity, measured in shannons, is defined using the Kullback-Leibler divergence as follows:

$$I(X;Y) = D_{KL}(P_{(X,Y)}||P_X \otimes P_Y)$$

where (X, Y) is a pair of random variables, $P_{(X,Y)}$ is their joint distribution and P_X is a marginal distribution. The MI is equal to zero when the joint distribution coincides with the product of the marginal, i.e. the two variables are independent. Equivalently, it can also be expressed in terms of entropy:

$$I(X;Y) = H(X) - H(X|Y) = H(Y) - H(Y|X)$$

which intuitively means that $I(X;Y)$ is the reduction of uncertainty in X when Y is observed. The code portion of the input vector ($\boldsymbol{c}$), is made meaningful by maximizing the MI between each code and the generator output. This corresponds to an additional term on the GAN objective function that can be seen as a regularizer:

$$\min_g \max_d [V(g,d) - \lambda I(\boldsymbol{c}, g(\boldsymbol{z}, \boldsymbol{c}))]$$

where λ is a hyperparameter that controls the contribution of the regularization, but is usually set to 1.

Computing the mutual information term $I(\boldsymbol{c}, g(\boldsymbol{z}, \boldsymbol{c})$ is difficult since, by definition it requires computing the posterior $P(c|x)$. Therefore the authors of InfoGAN propose to optimize a lower bound, obtained by defining an auxiliary distribution:

$$I(c, g(z,c) = H(c) - H(c|g(z,c)) =$$

$$= \mathbb{E}_{x\sim g(z,c)}[\mathbb{E}_{c'\sim P(c|x)}[\log P(c'|x)]] + H(c) =$$

$$= \mathbb{E}_{x\sim g(z,c)}[D_{KL}(P(\cdot|x)||Q(\cdot|x)) + \mathbb{E}_{c'\sim P(c|x)}[\log Q(c'|x)]] + H(c)$$

$$\geq \mathbb{E}_{x\sim g(z,c)}[\mathbb{E}_{c'\sim P(c|x)}[\log Q(c'|x)]] + H(c)$$

where $Q(c|x)$ is the auxiliary distribution that approximates $P(c|x)$. The latter relationship is due to the fact that the KL-Divergence is always ≥ 0. The entropy of the code can also be optimized, but in InfoGAN is treated as a constant term. Thanks to this lower bound is not necessary to compute the posterior $P(c|x)$ explicitly, but we still need a way to sample from $Q(c'|x)$. Thanks to a lemma, under certain regularity conditions, it is possible to equivalently express the lower bound as:

$$L_1(G,Q) = \mathbb{E}_{c\sim P(c),x\sim g(z,c)}[\log Q(c|x)] + H(c)$$

A reparametrization trick is used to allow sampling from a user-specified prior $P(c)$ (i.e., uniform distribution) instead of the unknown posterior. The auxiliary distribution will be modeled directly by a neural network. This definition of the lower bound allows to optimize it directly by adding it to the GAN objective without changing the GAN training procedure, that for InfoGAN becomes:

$$\min_{g,q} \max_d V_1(g,d,q) = V(g,d) - \lambda L_1(G,Q).$$

The InfoGAN architecture, shown in fig. 4.1, therefore includes the latent code as a second input to the generator. The neural network which models the auxiliary distribution $Q()$ is just a fully-connected layer that is trying to predict what the code is. This part will be used only during the iterations that feed fake inputs to the discriminator, since the code is not known for real images. For this reason, the added computational cost is negligible, and in practice, InfoGAN usually converges faster than the original GAN.

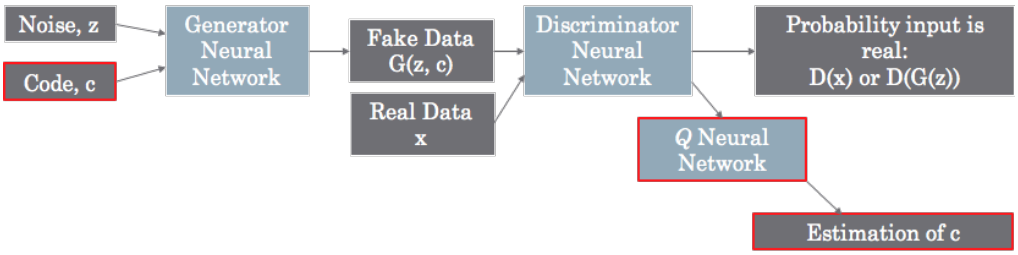


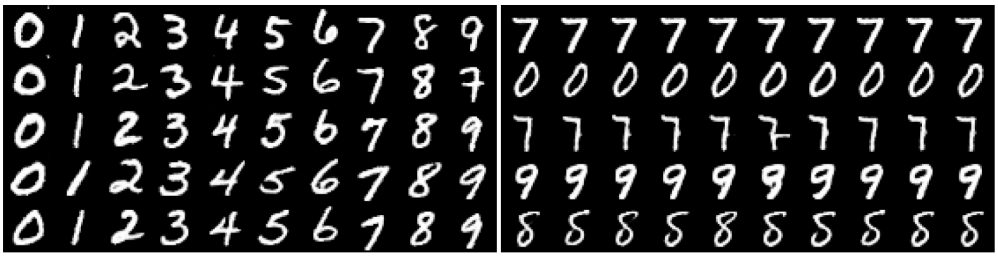
Figure 4.1. The InfoGAN architecture. Differently from GAN, it has a new input vector c, concatenated to the noise vector z. Another fully connected layer, the Q neural networks estimates the code statistic needed to compute the regularization term.

The improvements proposed in DCGAN, illustrated in the previous chapter, have also been included in InfoGAN to improve training stabilization. With the experiments, the author's goals were to prove that mutual information can be maximized efficiently and to check whether the learned representation presents characteristics of disentanglement and interpretability. To asses the latter, one latent factor at a time is varied to check if this corresponds to a single type of semantic variation in the produced image.

Experiments show how the lower bound allows to efficiently maximize mutual information and that in a regular GAN it is not guaranteed that the generator uses all latent codes. In terms of disentanglement, the authors qualitatively proved great performances on various image datasets. For the latent code vector $\boldsymbol{c}$ it is possible to constraint a single coordinate to take on continuous or discrete values and the range of those values. In particular, with discrete codes is possible to capture discontinuous factors, like classes in a dataset.

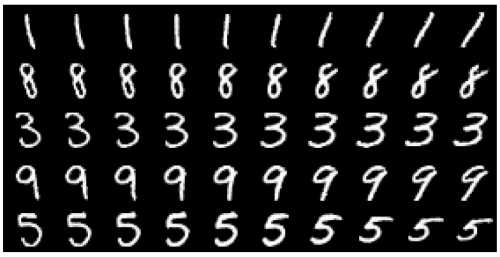
For example, in the MNIST hand-written digit datasets (fig. 4.2) a single discrete code with 10 values modeled the 10 possible digits. In practice, it learned to classify the digits with no supervision, with a 5% error rate. With continuous codes instead, it learned to represent the rotation and the width of the digits. These features are high-level concepts that prove the power of the model. By varying one code only a single one of those aspects changes, which means factors have been learned in a disentangled way. It should be noticed that the learned representation is generalizable since if it has been trained with codes varying in a certain range, during generation it is possible to expand the range of values with coherent results.

InfoGAN has also been tested on some 3D-image datasets, like chairs and faces. On chairs (fig. 4.3) it learned a continuous codes that represent the rotation and the width of the furniture. On the faces of the CelebA dataset (fig. 4.4), which contains colored images, InfoGAN recovered head azimuth, presence of glasses, hairstyle and facial expressions.

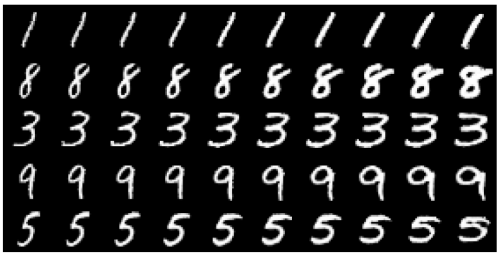


(a) Varying c_1 on InfoGAN (Digit type)

(b) Varying c_1 on regular GAN (No clear meaning)



(c) Varying c_2 from -2 to 2 on InfoGAN (Rotation)



(d) Varying c_3 from -2 to 2 on InfoGAN (Width)

Figure 4.2. InfoGAN on the MNIST handwritten digits. A discrete code learned to distinguish between digits. Two discrete codes learned rotation and with of the digit. Such a behavior has not been observed in original GAN [40].

(a) Rotation

(b) Width

Figure 4.3. InfoGAN on a dataset of 3D CAD images of chairs. The model learned to represent factor like rotation and with of the furniture in a disentangled way [40].

(c) Hair style

(d) Emotion

Figure 4.4. On the celebrity faces of the CelebA dataset, InfoGAN learned to extrapolate complex high-level concepts like hair style and emotion in a completely unsupervised way [40].

4.2 AI Explainability

The field of Explainable AI has been gaining popularity in the latest years due to the excellent results obtained by more complex machine learning models and their diffusion in a large variety of settings. Classical ML algorithms, like linear regression, SVM, K-Nearest Neighbors, clustering and decision trees are by construction more interpretable than deep learning methods. Neural networks are very powerful and are basically the only option to tackle more complex problems, but differently from the previous methods, they are not easily interpretable. These are often called black-box models since they perform great, but looking at the thousands of parameters of the network does not provide any insight into their functioning. The lack of interpretability causes a problem of trust towards these methods that can slow down or inhibit their adoption in critical fields like finance, medical and military. Besides, getting more information on the model's functioning can help find their shortcomings, like negative biases, avoiding negative consequences. To tackle this issue, researchers have proposed a variety of approaches for the different learning algorithms. We will report here some of the methods related to our experimental study.

The 2018 paper by Fong and Vedaldi [41] explains how controlled perturbations can be used to explain computer vision classifiers. While a lot of previous methods that produce image saliency maps that indicate where a neural network focuses its attention are effective, they are limited by their architectural constraints. The authors of this paper propose a method that is model-agnostic, or that works with any image classification model. The algorithm consists of producing explicit and interpretable image perturbations, feeding the modified images to a model and observing its response.

In the 2019 work by Uzunova et al. [42] the authors use a generative model (VAE) to improve the meaningful perturbation generation process. In particular, they worked with medical image classifiers that are trained to differentiate between different types of pathologies and healthy tissues. They pursued the idea of image deletion to remove some areas of the image and observe the classifier response. They defined the process of deletion for medical images as substituting a part of the picture with a healthy equivalent produced with a VAE.

The 2021 paper by Schutte et al. [43] proposes to use a state-of-the-art GAN architecture (StyleGAN) to generate meaningful image perturbations. The method is applied to histology and radiology images. In addition, they propose a technique to find the generative code associated with a new image by training a separate encoder model. More specifically, the method finds the optimal perturbation direction in the latent space to create a change in the model prediction. Synthetic images with changed predictions can be produced and analyzed. The last part of our experimental study borrows the idea of searching the direction of greatest output change in an input representation space.

4.3 Experimental setup

The following experimental part has been developed to test the capabilities of the InfoGAN model and to what extent they generalize the results reported on the original paper. After this, we decided to focus our attention on AI explainability and how an unsupervised controllable generation method like InfoGAN could help with that. The goal here is to get more insight into a well-functioning image classification model. Therefore the experimental study has been divided into three parts. The first part focus is the evaluation of the capabilities of InfoGAN and to what extent they generalize. The second part is concerned with the usage of InfoGAN output images to build an explainability system on a good-performing classifier. Finally, the third part is focused on extending this explainability system with the usage of InfoVAE.

The goals of **the first part of the analysis** are:

1. Evaluating InfoGAN's image quality and diversity both visually and with an objective metric.
2. Evaluating the model's performance on more complex datasets, in terms of interpretability and disentanglement of the produced representation.
3. Evaluating the effect of the Dropout technique [44] on InfoGAN training stability.

Image quality evaluation is done both visually in a subjective way and through the FID score [45], a well-known metric for GAN evaluation. The important point to consider is that GANs do not provide an objective function to compare the performance of different models. In short, the Frechet Inception Distance (FID) score measures the distance between feature vectors of real and generated images. Since the goal is to evaluate the performance of the model as a whole, this measure considers the statistic of a group of images and not a single specific one. The distance is measured on a latent space of extracted feature with a particular model, the Inception v3. A lower score indicates that the group of images are more similar, so it is desirable.

The FID score is an improved version of the previous Inception score, proposed by Salimans et al. in 2016 [45]. The authors proposed it as an automatic method to evaluate samples and compared it to human evaluation, using a crowd-source platform (Turing test). The idea is to use a pre-trained classifier network to classify the generated images. In particular, the authors propose to use the highly successful Inception v3 architecture (Szegedy et al. [46]), from which the name derives. A batch of generated images from the GAN generator is fed to the Inception classifier, and the probabilities of each image belonging to each class are computed. These are then summarized into a single score that considers the two fundamental aspects of generative models:

- **Image quality**: the resemblance of generated images to real images.
- **Image diversity**: how wide is the spectrum of images that can be produced.

The **Inception score** ranges from 1.0 to the number of classes on which the classifier has been trained on, 1,000 in the case of the ILSVRC 2012 dataset used in the Inception v3 model. For is nature, the technique involves using a labeled dataset to perform the evaluation. In more specific terms, by applying a pre-trained model on the generated data, we obtain the conditional distribution of the label $p(y|\boldsymbol{x})$. For the produced images that are meaningful, the conditional should have a low entropy, which means they clearly belong to a given class. At the same time, given that we want varied images, we desire the marginal $\int p(y|\boldsymbol{x} = G(z))dz$ to have high entropy. By combining these two conditions, we obtain the Inception score formulation:

$$I(\boldsymbol{x}, y) = exp[\mathbb{E}_{\boldsymbol{x}}\, D_{KL}(p(y|\boldsymbol{x})||p(y))].$$

Since the score also measures diversity, the authors suggest computing it on a large number of samples ($\sim 50k$).

Another interesting aspect discussed in [45] is the idea of feature matching, later exploited in the FID score. The authors used it as a way to improve GAN convergence; instead of maximizing the output of the discriminator, the distance in feature space, measured in an intermediate layer of the discriminator, is minimized. The role of this branch of the network, therefore, becomes to shape the latent space in such a way that it extracts features we are interested in. This technique has helped to reduce over-training of the discriminator, which slows down or stops convergence to the Nash equilibrium.

The authors also discuss the usage of human evaluation for GAN performance. The simplest idea, which is the one we also used in our experiments, is to ask people to distinguish between real and fake images, which is essentially the Turing test. This can be done, for example, by using a website with a specific purpose user interface. The downside of human evaluation lies in the fact that it is subjective and therefore affected by factors like the setup used and the motivation of the person. For example, telling people about their mistakes in evaluation makes them smarter. On the other hand, even a good score is useless if humans can easily spot the mistakes, so we believe the two evaluation techniques should be used in conjunction.

The Inception score has mainly three drawbacks: first, it can easily be fooled in the evaluation of diversity if a few (or even one) images of each class are produced. Second, it only looks at the generated images and does not compare them to the real image dataset. This might create a statistic that is a bit off and idealistic because it depends on the classifier's abilities. Lastly, since it is computed with a network pre-trained on a benchmark dataset (usually ImageNet [47]) it can miss useful features of the dataset it is evaluating. This last point is also a weakness of the FID score.

These shortcomings motivated the invention of the improved **FID score** [45], where the Inception network is used to extract features from an intermediate layer. The distribution for these features is modeled through a multivariate Gaussian distribution of mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$. The score is then measured between generated ($\boldsymbol{g}$) and real images ($\boldsymbol{x}$) of the same distribution as follows:

$$\text{FID}(\boldsymbol{x}, \boldsymbol{g}) = ||\boldsymbol{\mu}_x - \boldsymbol{\mu}_g||^2 + Tr(\boldsymbol{\Sigma}_x + \boldsymbol{\Sigma}_g - 2(\boldsymbol{\Sigma}_x \boldsymbol{\Sigma}_g)^{\frac{1}{2}}).$$

As shown in figure 4.5, the FID score is consistent with the human judgment of image quality since it increases by applying various disturbances to them. The score shows that it is sensible to various image degradation effects and captures the disturbance level very well. On the negative side, since it is based solely on feature extraction, like Inception, it might not consider the spatial relationship between objects as much as their presence or absence.

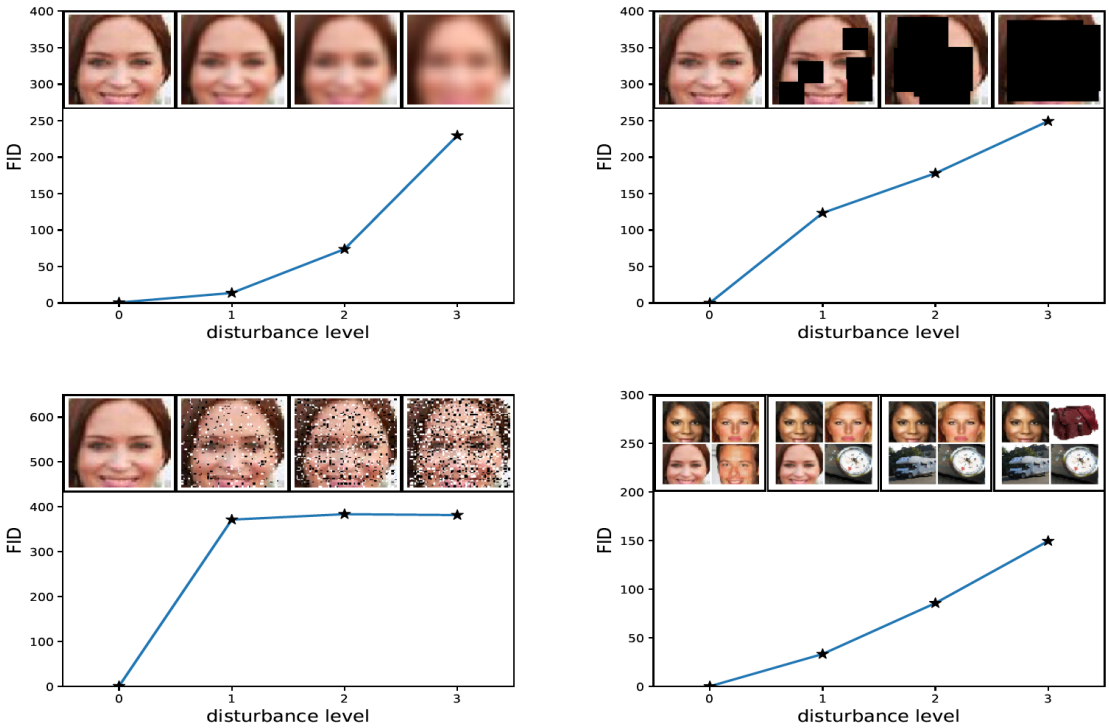


Figure 4.5. Visual evaluation of FID score performance. The score is sensible to various kind of modifications on images which lower perceived quality. In particular, going clockwise, Gaussian blur, implanted black rectangles, salt and pepper noise and a contamination of dataset [45].

It should be underlined that the FID score suffers from a relatively high bias problem. The variance instead is pretty low. By computing the score between different batches of the same dataset, a zero value should ideally be produced; instead, values like the following appears:

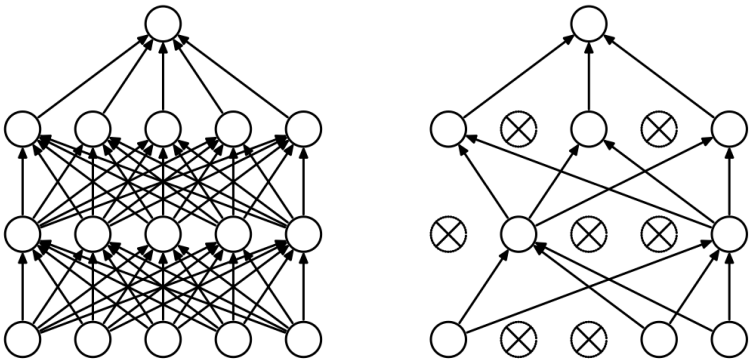
DATA SET	AVG. FID	DEV. FID
CELEBA	2.27	0.02
CIFAR10	5.19	0.02
FASHION-MNIST	2.60	0.03
MNIST	1.25	0.02

Figure 4.6. Bias and variance of the FID score applied on various datasets [48].

That being said, the FID score still remains one of the most accepted ways of evaluating a GAN objectively, and that is also why we decided to use it in our experiments.

Overfitting is often a serious problem with big neural networks with high capacity, and GANs are no exception. **Dropout** is a simple technique for neural network regularization introduced by Srivastava et al. [44]. The idea is to randomly drop some units and their connections at training time. The equivalent architecture is shown in fig. 4.7. In this way, it is like in each layer there are fewer units which has a regularization effect. This prevents units from co-adapting too much. The intuition is that a neuron cannot rely too much on one input feature, so each neuron is encouraged to provide a useful output.

More specifically to each layer is assigned a probability of dropping a neuron in a training iteration. This technique has the effect of shrinking the norm of the weights, similar to what L2-regularization does. At test time Dropout has to be disabled. This method has been proved to reduce overfitting and performs better than other regularization methods. Dropout improves deep neural networks performance and is used as a standard tool in many applications. Therefore we decided to investigate its usage to improve the challenging convergence of InfoGAN.



(a) Standard Neural Net (b) After applying dropout.

Figure 4.7. The architecture of a neural network without and with Dropout [44].

In the second part of our experiments, we tried to exploit the unsupervised controllable generation of InfoGAN to generate classification maps, that we called InfoGAN Explainability Maps. We decided to work with the FashionMNIST dataset since it is the one where we obtained the best results in terms of code interpretability and disentanglement. We started by creating a classifier for the dataset and training it until it had good performances (accuracy $> 90\%$). For this purpose, we used a simple CNN architecture, whose code is reported in figure 4.8. The FashionMNIST contains 70 000 images, of which 60k were used for training and the remaining ones for testing the accuracy of the classifier. The dataset is labeled with 10 classes.

With the goal of having more insights on how this classifier works and makes its decisions, we decided to feed it some generated input images by interpolating on meaningful codes and seeing how the model reacted to these changes. This approach is inspired by the previously discussed works on meaningful perturbations for explainability [41] [42]. For each image, we recorded the classifier's predicted class and the maximum Softmax output as a metric of decision confidence. By varying the input according to some high-level concepts of an image, we expect to get additional insights into the classifier and possibly discover some of its shortcomings. This method, intended as a tool for explainability has the advantage of being model-agnostic. If proven useful, these grids might be attached to a model when it is given for production.

```
Net(
  (conv1): Conv2d(1, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(8, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (fc1): Linear(in_features=784, out_features=75, bias=True)
  (dropout): Dropout(p=0.2, inplace=False)
  (fc2): Linear(in_features=75, out_features=10, bias=True)
)
```

Figure 4.8. Simple CNN model to classify FashionMNIST images at 28x28 resolution. The goal of the second experimental part was to explain the decisions of this classifier.

Inspired by the interesting results produced by InfoGAN, **in the third part of our experiments**, we decided to investigate more precisely what kind of factors influence a model's decision. The question we asked our self was: what is the minimum perturbation of an image necessary to change a model prediction? To achieve this, we needed a way to map an input image in a latent space where we can apply perturbations. Unfortunately, this is a shortcoming of GANs since they do not allow mapping between the data space and the input space. Therefore, we decided to use the Variational Autoencoder model, in particular the InfoVAE architecture [49], which focuses on image quality and the usage of all latent features by the decoder. The procedure we used, was inspired by the work of Schutte et al. [43], previously described, and can be summarized by the following steps:

1. Compute latent representation of the image to explain using the encoder network.
2. Pass the latent representation to the decoder network and obtain a generated image.

3. Pass the generated image to the classifier and compute the predicted class.
4. Compute a vector $\boldsymbol{\alpha}$ in the latent space indicating the direction producing the greatest classifier output variation.
5. Search for the minimum variation to the latent space in the $\boldsymbol{\alpha}$ direction that changes the classifier predicted class.
6. Compute the saliency map by subtracting the original image from the one obtained by passing the newly computed latent vector.

With these steps we obtain a color-coded map, that we call InfoVAE Decision Map, which should highlight the areas of the image that influence the decision. The implementation of the three experimental parts is described in the following section.

4.4 Implementation

The experimental setup implementation is made using PyTorch, a popular machine learning library written in Python. The project implementation is composed of five parts: the InfoGAN code, the FID score code, a Python notebook [50] to run and manage the benchmark on InfoGAN, a second notebook that implements the FashionMNIST classifier and a third one to compute the InfoVAE Decision Maps. The InfoGAN Wxplainability Grids are obtained by combining the saved output of InfoGAN and the output of the FashionMNIST classifier. The InfoGAN code is based on an open-source implementation [51], which has been extended with the support of additional datasets and a slightly modified architecture, like adding optional Dropout layers. The FID score evaluation is also based on code available on an open-source repository [52]. The experiments are controlled by a Python script that can run on Google Colab computational resources or any other Linux machine, like the remote AWS one we employed.

The InfoGAN implementation is a Python project structured in many source-code files, reported in figure 4.9.

```
├── train.py # train script
├── data
│   ├── mnist # mnist data (not included in this repo)
│   ├── ...
│   └── fashion-mnist # fashion-mnist data (not included in this repo)
│
├── config.py # hyperparameters for training
├── utils.py # utils
├── dataloader.py # dataloader
├── mytorchsummary.py # create a summary of the network architecture
│
├── mnist_generate.py # to generate images with the trained model
├── ...
├── chestxray_generate.py
│
├── models # infoGAN networks for different datasets
│   ├── mnist_model.py
│   ├── svhn_model.py
│   ├── celeba_model.py
│   └── chestxray_model.py
│
└── results # generation results to be saved here
```

Figure 4.9. InfoGAN source code project structure.

The *train.py* script contains the code that implements the network training loop according to stochastic gradient descent, more specifically the Adam algorithm. Training iterations are alternated between generator and discriminator in the adversarial manner already described for GANs. In terms of training criteria, three different losses are used. For the discriminator output, a Binary Cross-Entropy loss is used. The output of the *QHead* module employs a Cross-Entropy loss for discrete codes and a Normal Negative-Log Likelihood loss for continuous codes.

The script also allows creating a checkpoint to resume training at a later time or to perform model selection. The checkpoint also saves, every x epochs, a generated

image at that point to allow a visual quality check. PyTorch provides components optimized to run on GPUs, hardware accelerators specifically designed to perform operations with multiple data in parallel. Extensive logging of the training progress is also printed in the console and saved on file for future inspection. In addition, at the end of the training, a chart of the loss function trend is also produced.

The configuration file *config.py* allows setting parameters that affect training, of which the names are already explanatory. It is possible to set the learning rate for the generator and discriminator separately and use or not the Dropout layers. Since the Adam optimizer is used, it is possible to control the Momentum and RMSprop terms' coefficients. The file *utils.py* contains utility functions for things like weight initialization, normalization and getting noise samples from a distribution. The number and shape of InfoGAN input noise and codes can be configured inside the training script, in the section shown in figure 4.10.

```
# Set appropriate hyperparameters depending on the dataset used.
# The values given in the InfoGAN paper are used.
# num_z : dimension of incompressible noise.
# num_dis_c : number of discrete latent code used.
# dis_c_dim : dimension of discrete latent code.
# num_con_c : number of continuous latent code used.
if(params['dataset'] == 'MNIST'):
    params['num_z'] = 62
    params['num_dis_c'] = 1
    params['dis_c_dim'] = 10
    params['num_con_c'] = 2
elif(params['dataset'] == 'SVHN'):
    params['num_z'] = 124
    params['num_dis_c'] = 4
    params['dis_c_dim'] = 10
    params['num_con_c'] = 4
```

Figure 4.10. InfoGAN input vector configuration. It is possible to set the size of the random vector z, the number of discrete codes, the conditionality of those and the number of continuous codes.

In terms of input datasets, the model has been trained with five of them: MNIST handwritten digits (B&W) [53], FashionMNIST (fashion items from Zalando, B&W [54]), ChestXRay (thoracic X-Rays, B&W [55]), Furniture (from Kaggle, B&W [56]) and Aloi (Amsterdam Library of Object Images, varying viewing direction, B&W [57]). The last three datasets must be downloaded in the data folder before running training, while the formers can be downloaded automatically with PyTorch. In additional the original model supports also SVHN (Google's Street View House Numbers [58]) and CelebA [59], both in RGB colors. A data-loader component, defined in *dataloader.py*, provides batches of shuffled data to the training process, according to a specified batch size in the configuration file (*config.py*). In addition, the data loader performs some pre-processing on the data, like resizing, cropping and applying normalization.

The *mytorchsummary.py* script contains code to print a human-readable description of the network architecture and parameters in use. In particular, it computes for each layer the input and output dimensions, which is useful for inspection and debugging.

The various *dataset_generate.py* scripts import a saved checkpoint of the generator weights and produce images by interpolating two input codes at a time. In this way, the produced image is a matrix containing multiple images (by default 10 by 10), where moving from left to right is varying one code, and top to bottom another one. The random noise vector and the other codes are kept static. In this way, it is possible to see if a code corresponds to some high-level feature, as it is desired. The produced images are saved under the *results* folder.

The *dataset_model.py* scripts contain the model definition for the various datasets. The distinctions are mainly justified by the different sizes and depth of these datasets, which determine the network layers' shape. The models for MNIST (also used for FashionMNIST), SVHN and CelebA are defined according to the original InfoGAN paper description. We have designed the ChestXRay and Aloi architectures to fit the dataset's unique characteristics. For the Furniture dataset we used the ChestXRay model. Optionally this last model can use Dropout layers in the first half of training epochs, which has been observed to improve convergence.

In each model implementation's file, four different components are defined: the discriminator (fig. 4.11), the generator (fig. 4.13), *DHead* and *QHead*. *DHead* contains a convolutional layer, whose result is passed to a sigmoid function. The output of the discriminator, still a vector, is passed to this last layer which transforms it into a probability (fig. 4.12). *QHead* contains a two-layer network with three different outputs: the first layer is common, and the last one is different for each output. These layers are crafted to compute the estimation of the c codes in terms of logits for the discrete codes and mean and variances for the continuous ones (fig. 4.14).

The Chest X-Ray dataset contains over 5,000 black and white images divided into two categories "normal" and "pneumonia". We did not use the labels in our experiments, and the images are first resized to 256 by 256 pixels. The Chest X-Ray model generator comprises seven transpose-convolutional layers that gradually increase the output size in terms of width and height while reducing the depth up to 1. Each of the first four layers is also followed by a batch normalization layer similar to what is done in the CelebA model. The activation functions used in each layer are ReLUs, except for the last one where a *tanh* is used. The *QHead* module contains one standard convolutional layer, followed by a batch normalization and a Leaky ReLU activation. Then the discrete logits and the mean for continuous codes are obtained by going through another convolutional layer. The variance for continuous codes is obtained by going through two additional convolutional layers and an exponential function.

The discriminator is composed of six convolutional layers, and to each one, with the exception of the first, is applied batch normalization. According to the configuration, the first four layers can be followed by a Dropout layer, with a probability of an input being zeroed equal to 0.25. If the parameter *use_dropout* is equal to true, the Dropout layers will be active for the first half of the training epochs (*num_epochs*). The activation function used here is the Leaky ReLU. The *DHead* module simply contains a convolutional layer whose output is fed into a sigmoid function.

The first Python notebook [50] made for the Google Colab platform contains

the essential code for executing the experiments of the first part (see previous section). The notebook is composed of cells containing code; each one can be executed independently. At the beginning of the "InfoGAN" section, the GitHub repo of InfoGAN is cloned, which means downloading the latest version of the code-base on the machine. Supposing to work on the Google platform, the datasets can be imported from Google Drive, a cloud storage service. The outputs of the training can also be saved in the same place for persistence. This is necessary since the Linux-machine instance that Colab provides is a clean one every time you log in. Therefore, the following lines are to copy and extract the various datasets supported, most of which cannot be download directly using PyTorch.

Next, there is the cell that starts training simply by executing *train.py*. It is possible to pass an option to reset training by removing the previous output (*-ro*) and to resume training using the *-load_path* option with the path of a checkpoint. The following cell allows to generate an image with perturbations interpolating the codes, according to the used script. For example, for the ChestXRay dataset, it is defined into *chestxray_generate.py*. The *-load_path* option allows specifying the checkpoint to use for generation. After generation, is it possible to execute the next cell to save the output on Google Drive.

The second section of the notebook, titled "FID Score" starts by using one of the *_generate.py* scripts with the *-s* option to generate and save 1000 images from computing the FID score. The following cell load and extract the original dataset to compute the score. There is also a small script to sample a number of images from a folder randomly. This is done in case we use a labeled dataset divided into separate folders. In this case, we must pick images from each of the classes in order to represent the entire distribution. In the end, in the last cell, by calling *fid_score.py* with the generated and the original data, the score is finally computed and printed.

```
class Discriminator(nn.Module):
    def __init__(self):
        super().__init__()
        # 1 x 64 x 64
        self.conv1 = nn.Conv2d(1, 32, 4, 2, 1, bias=False)
        self.drop1 = nn.Dropout(0.25)
        # 64 x 32 x 32
        self.conv2 = nn.Conv2d(32, 128, 4, 2, 1, bias=False)
        self.bn2 = nn.BatchNorm2d(128)
        self.drop2 = nn.Dropout(0.25)
        # 128 x 16 x 16
        self.conv3 = nn.Conv2d(128, 256, 4, 2, 1, bias=False)
        self.bn3 = nn.BatchNorm2d(256)
        # 256 x 8 x 8
        self.conv4 = nn.Conv2d(256, 1024, 4, 2, 1, bias=False)
        self.bn4 = nn.BatchNorm2d(1024)
        # 1024 x 4 x 4

    def forward(self, x):
        if config.currentEpoch < (config.params['num_epochs']/2) and config.params['use_dropout'] == True:
            x = self.drop1(F.leaky_relu(self.conv1(x), 0.1, inplace=True))
            x = self.drop2(F.leaky_relu(self.bn2(self.conv2(x)), 0.1, inplace=True))
        else:
            x = F.leaky_relu(self.conv1(x), 0.1, inplace=True)
            x = F.leaky_relu(self.bn2(self.conv2(x)), 0.1, inplace=True)

        x = F.leaky_relu(self.bn3(self.conv3(x)), 0.1, inplace=True)
        x = F.leaky_relu(self.bn4(self.conv4(x)), 0.1, inplace=True)

        return x
```

Figure 4.11. InfoGAN discriminator architecture code, for the Aloi dataset.

```
class DHead(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv = nn.Conv2d(1024, 1, 4)
        # 1 x 1 x 1

    def forward(self, x):
        output = torch.sigmoid(self.conv(x))
        return output
```

Figure 4.12. InfoGAN architecture code of the *DHead* module, for the Aloi dataset.

```python
class Generator(nn.Module):
    def __init__(self):
        super().__init__()
        # 112 x 1 x 1
        self.tconv1 = nn.ConvTranspose2d(112, 1024, 4, bias=False)
        self.bn1 = nn.BatchNorm2d(1024)
        # 1024 x 4 x 4
        self.tconv2 = nn.ConvTranspose2d(1024, 256, kernel_size=4, stride=2, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(256)
        # 256 x 8 x 8
        self.tconv3 = nn.ConvTranspose2d(256, 128, 4, 2, 1, bias=False)
        self.bn3 = nn.BatchNorm2d(128)
        # 128 x 16 x 16
        self.tconv4 = nn.ConvTranspose2d(128, 64, 4, 2, 1, bias=False)
        # 64 x 32 x 32
        self.tconv5 = nn.ConvTranspose2d(64, 1, 4, 2, 1, bias=False)
        # 1 x 64 x 64

    def forward(self, x):
        x = F.relu(self.bn1(self.tconv1(x)))
        x = F.relu(self.bn2(self.tconv2(x)))
        x = F.relu(self.bn3(self.tconv3(x)))
        x = F.relu(self.tconv4(x))
        img = torch.sigmoid(self.tconv5(x))

        return img
```

Figure 4.13. InfoGAN generator architecture code, for the Aloi dataset.

```python
class QHead(nn.Module):
    def __init__(self):
        super().__init__()

        self.conv1 = nn.Conv2d(1024, 128, 4, bias=False)
        self.bn1 = nn.BatchNorm2d(128)

        self.conv_disc = nn.Conv2d(128, 10, 1)
        self.conv_mu = nn.Conv2d(128, 2, 1)
        self.conv_var = nn.Conv2d(128, 2, 1)

    def forward(self, x):
        x = F.leaky_relu(self.bn1(self.conv1(x)), 0.1, inplace=True)

        disc_logits = self.conv_disc(x).squeeze()

        mu = self.conv_mu(x).squeeze()
        var = torch.exp(self.conv_var(x).squeeze())

        return disc_logits, mu, var
        # [10], [2], [2]
```

Figure 4.14. InfoGAN architecture code of the *QHead* module, for the Aloi dataset.

To implement the second part of the experimental study, regarding the composition of Explainability Grids, we simply used the image grids produced by InfoGAN. In particular, since we trained the model for FashionMNIST with the same architecture as MNIST, we used the *mnist_generate.py* script to produce the images. We added a launch option to the said script (*-ss*), that other than producing the grids with 100 images, it saves those singularly in a folder, numbered from 0 to 99. In this way, we are able to feed them to the FashionMNIST classifier that we previously trained in order to get the desired output. In particular, we printed the predicted class and the corresponding Softmax score as an index of confidence. We then added those pieces of information on the bottom of each image to compose the explainability grids.

To implement the third and last part, regarding the InfoVAE Decision Maps, we used an open-source implementation of the said model and trained it on the FashionMNIST dataset at 28 by 28 pixels. In particular, we used size of 4 for the latent space. The code that implements the steps described in the previous section, for finding the class-changing perturbation, is reported in figures 4.15 and 4.16.

```
index_expl = 1001 # index of the instance to explain

prediction = clf(Xtest[index_expl]) # instance classification
sm = torch.nn.Softmax(dim=1)

latent = VAE.encoder(Xtest[index_expl]) # input code of the selected instance, using the encoder
original_prediction = sm(clf(VAE.decoder(latent)))
original_image = VAE.decoder(latent) # reconstruction of the selected instance (decoder)
pred_index = original_prediction.argmax().detach().numpy() # predicted class
alpha = np.zeros(latent.shape[1])
j = 0
delta_latent = 0.1 # delta for finite differences: (f(x+dh)-f(x))/dh

for i in range(alpha.shape[0]):
    # for each latent code compute the following derivative:
    # dalpha = ((predicted probability of the perturbed code)-(original predicted probability)/(code perturbation))
    perturbed_latent = torch.clone(latent)
    perturbed_latent[0,i] = perturbed_latent[0,i] + delta_latent
    out = sm(clf(VAE.decoder(perturbed_latent)))
    alpha[i] = (out[0,pred_index]-original_prediction[0,pred_index])/delta_latent
```

Figure 4.15. InfoVAE decision maps code. Here the direction in the latent space that causes maximum variation of the classifier output is computed (vector *alpha*).

```
prediction = pred_index
lambda_coeff = 0
while (prediction==pred_index) & (lambda_coeff <10.):
    # once the vector alpha has been computed search for the minimum perturbation
    # in the alpha direction necessary to change the classifier prediction
    # new_image = decoder(latent + lambda * alpha)
    perturbed_latent = torch.clone(latent)
    perturbed_latent = perturbed_latent + lambda_coeff * torch.from_numpy(alpha)
    out = sm(clf(VAE.decoder(perturbed_latent.float())))
    out_image = VAE.decoder(perturbed_latent.float())
    prediction = out.argmax().detach().numpy()
    lambda_coeff += 0.1

# saliency map to use as explenation (perturbed image with different class - original image)
explanation = (out_image-original_image)[0,0,:,:].detach().numpy()
```

Figure 4.16. InfoVAE decision maps code. Here shown the search of the point when the predicted class changes, in the direction of the vector *alpha*.

4.5 Results

The first part of the experimental study focuses on the performance of InfoGAN. We will first report the experiments and results relative to this part. We have executed about 50 experiments in total between the five different datasets. Other than the data, in each run, we changed the values of some hyper-parameters or tweaked the network architecture. Between the hyper-parameters, we tuned we have the batch size, the learning rate for both generator and discriminator, the size of the z input vector and the number and shape of the continuous and discrete codes. Also, when we experienced training collapse, we tried to restart the experiment with the Dropout layers activated on the discriminator. When used, Dropout was active for the first half of the training epochs.

For each experiment, we recorded the FID score, the lower bound of the mutual information for both discrete and continuous codes, the final value of the loss and its trend for both the generator and the discriminator. Additionally, we evaluated subjectively the visual quality, visual domain coverage and interpretability of the codes. FID score was computed only for MNIST and FashionMNIST instances, and as said in the previous chapter, it is a slightly different version than the theoretical score. Still, it is a useful metric to compare our experiments. Some of most interesting results with the FashionMNIST dataset are reported in tables 4.1 and 4.2.

ID	**z size**	**c codes**	**epochs**	**batch size**	**L.R.**	**Dropout**
111	62	1 disc(10)+2 cont	50	128	2E-4	No
112	62	1 disc(10)+2 cont	50	256	2E-4	No
113	62	1 disc(10)+2 cont	100	128	2E-4	No
114	62	1 disc(10)+2 cont	100	128	2E-4	No
115	100	1 disc(10)+2 cont	100	128	2E-4	No
116	30	1 disc(10)+2 cont	100	128	2E-4	No
117	30	1 disc(10)+2 cont	200	128	2E-4	No
118	0	1 disc(10)+2 cont	100	128	2E-4	No
119	62	1 disc(10)+3 cont	100	128	2E-4	No
120	62	0 disc + 2 cont	100	256	2E-4	No

Table 4.1. Experimental setting for 10 different experiments on the FashionMNIST dataset. Each experiment is identified by the ID number. For the c codes 1 disc(10)+2 cont means there is 1 discrete code with 10 possible values and 2 continuous ones. L.R. stands for learning rate, which is the same for both the generator and the discriminator networks.

Results of these experiments show that InfoGAN is actually able to reproduce an excellent image quality on simpler datasets like the FashionMNIST. To a human eye images are indistinguishable from the training set and they cover pretty well the entire distribution. This visual evaluation is also confirmed by the FID score, which follows a pattern that is coherent to the human evaluation. We did not need to use Dropout with the simpler MNIST and FashionMNIST datasets since it is rare that training collapses with those.

ID	FID score	Visual Quality [1-5]	Interpretability of Cs [1-5]	M.I. lower bound (disc)	M.I. lower bound (cont)
111	68.23	4	4	0.0358	-0.0615
112	74.44	4	4	0.0313	-0.0826
113	68.62	4	4	0.0175	-0.0626
114	66.86	4	5	0.0035	-0.0497
115	64.82	5	5	0.0484	-0.0325
116	70.89	5	5	0.0077	-0.1019
117	73.69	4	5	0.0014	-0.1027
118	121.45	3	4	0.0016	-0.0937
119	70.13	5	5	0.0400	-0.1530
120	76.22	4	5	0.0000	-0.1481

Table 4.2. Measured parameters for the 10 different experiments on the FashionMNIST dataset reported in the previous table. A lower FID score actually corresponds to a better perceived visual quality.

Dropout layers on the discriminator were instead beneficial with the three more complex datasets we tried. Frequently training did not manage to improve at all in the first epochs because of an unbalance between discriminator and generator capabilities. The problem of stuck training was more frequent in the first epochs of training. By using Dropout, we managed to overcome this obstacle in all datasets. An example of this situation is reported in figure 4.17.

With the ChestXRay dataset, after some tuning, we also managed to obtain great results in terms of visual quality. In our best instance, at epochs 50, images were already sharp, and at epoch 150 they were indistinguishable from the original ones (fig. 4.18). It should be noted that we worked with this dataset at a high resolution (256 by 256 pixels) because we wanted to pick up details in the X-ray. This is necessary if images need to be used in a medical setting for diagnosis. Increasing the resolution of the images required a bigger model with more trainable parameters. This higher capacity might be one of the reasons for the difficulties we found in obtaining meaningful input codes for this dataset.

In terms of visual quality also the Aloi 3D object datasets distribution was reproduced with great results (fig. 4.23). Instead, on the Furniture images, we obtained a good quality and sharpness, but the objects are often not realistic in terms of shape (fig. 4.24). The Furniture dataset is the most visually complex of our setup, and it is where the model starts to shake.

Interpretability of the input codes is the main goal of InfoGAN but, while it does that better than GANs, we have seen that it struggles on more complex datasets. On the MNIST digits we basically obtained the same results of the original work by finding digit thickness, inclination and categories. On FashionMNIST, which is already more articulated, we manage to recover a continuous factor that seems to express brightness (fig. 4.19). A second continuous factor might encompass the length of sleeves in T-shirts, but is not as distinct (fig. 4.20). Probably the most

interesting result is the ability to classify almost perfectly the various categories of articles with a discrete code, shown in figure 4.20. This is particularly remarkable if we consider that the model has been trained completely unsupervised.

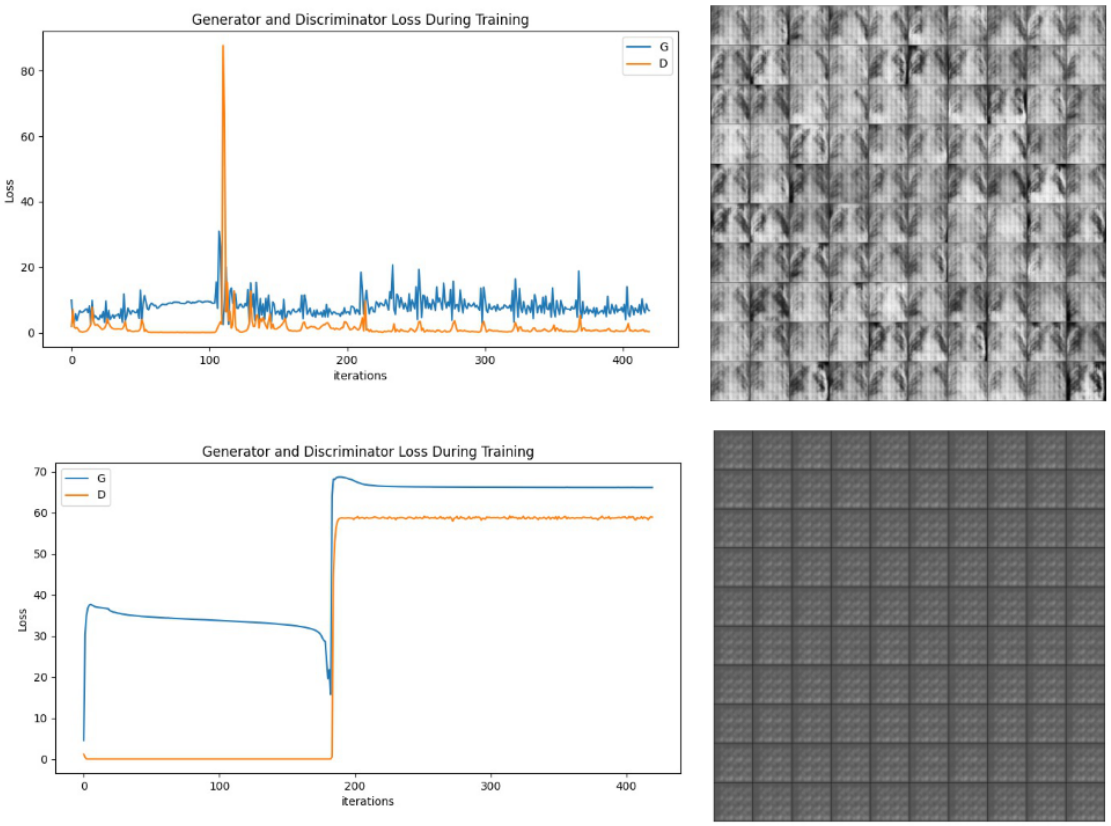


Figure 4.17. InfoVAE: the effect of dropout on the first 20 epochs. Here the ChestXRay dataset is used. Results on multiple experiments show that Dropout avoids training collapse.

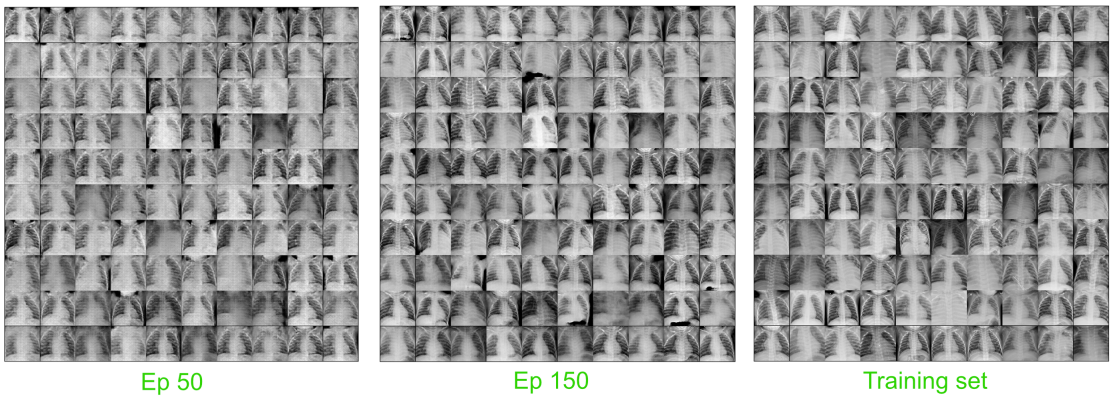


Figure 4.18. InfoGAN generated images trained on the ChestXRay dataset. On the left, the images produced at epoch 50: they look good but are not very detailed. On the centre, results at epoch 150 where images are almost indistinguishable from the training set (right).

Moving on to increasingly complex datasets, with the ChestXRay we performed more than 15 experiments, but we only observed one interpretable continuous code related to brightness (fig. 4.22). In the original dataset, we observed that some images are blurrier than others or that the rib cage's width varies, but none of those factors has been recovered by the model. On the Aloi distribution (fig. 4.23), we did hope to recover the rotation of objects since the data contains the same object at different angles, but we did not manage to do it. Instead, it looks like we recovered the usual brightness factor on a continuous code and the discrete code seems to recover some objects that have similar shapes, which is related to the class. On the Furniture data (fig. 4.24) we did not observe explainable factors, except maybe for a stronger presence of some items in some of the rows, are that generated with the same value of a discrete code. ' Even though InfoGAN struggles with more complex datasets, it remains a valuable approach for certain domains where it is able to learn an interpretable and disentangle representation. Training is not straightforward and requires the tuning of the input codes, therefore modifying the input layer of the architecture, in addition to the usual hyper-parameters. Techniques proposed for improving GAN convergence, like the usage of Dropout, can help, but the power of the model remains in the fact that it does not need supervision. Perhaps an interesting topic could be the usage of a similar approach in a semi-supervised setting, where one has already an idea of the features he would like to extract.

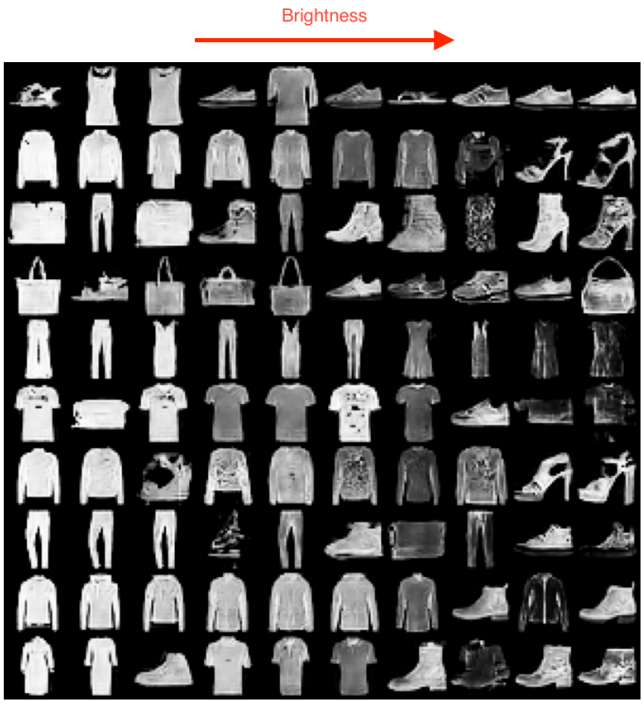


Figure 4.19. InfoGAN generated images on the FashionMNIST dataset; the noise vector z is fixed while a continuous code varies on the horizontal axis and a discrete one on the vertical axis. This representation seems to capture the object's brightness.

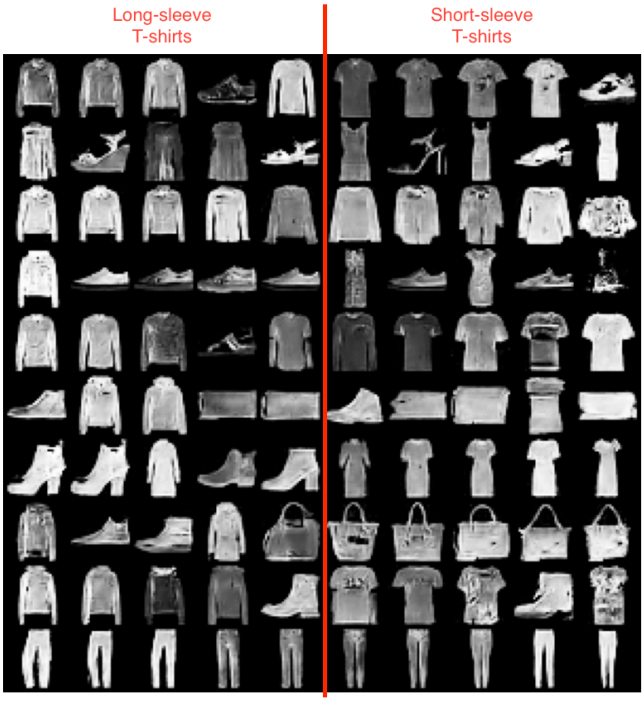


Figure 4.20. InfoGAN - FashionMNIST; continuous code variation on the horizontal axis and discrete on the vertical axis. The continuous code seems to capture the length of sleeves on shirts.

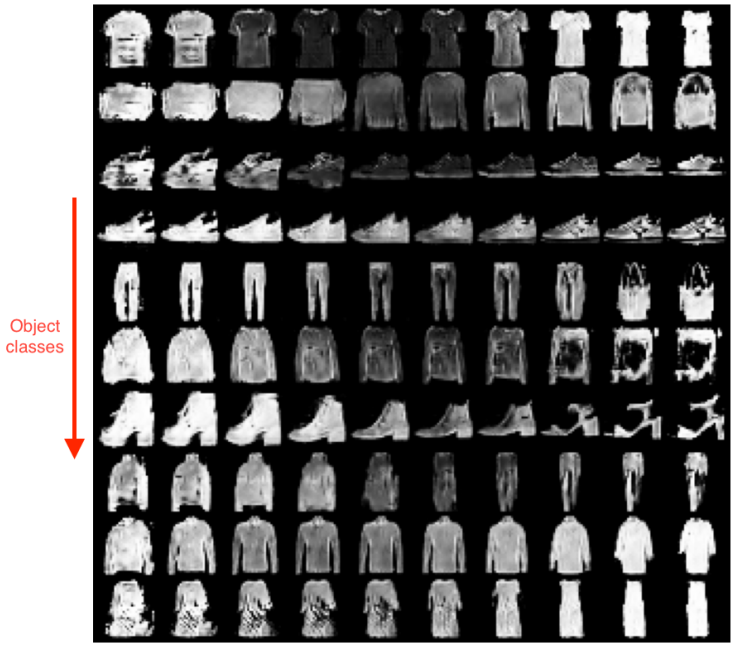


Figure 4.21. InfoGAN - FashionMNIST; continuous code interpolation on the horizontal axis and discrete one on the vertical axis. The discrete code classify almost perfectly the fashion items. This shows the power of this representation, since it can understand such high-level differences without supervision.

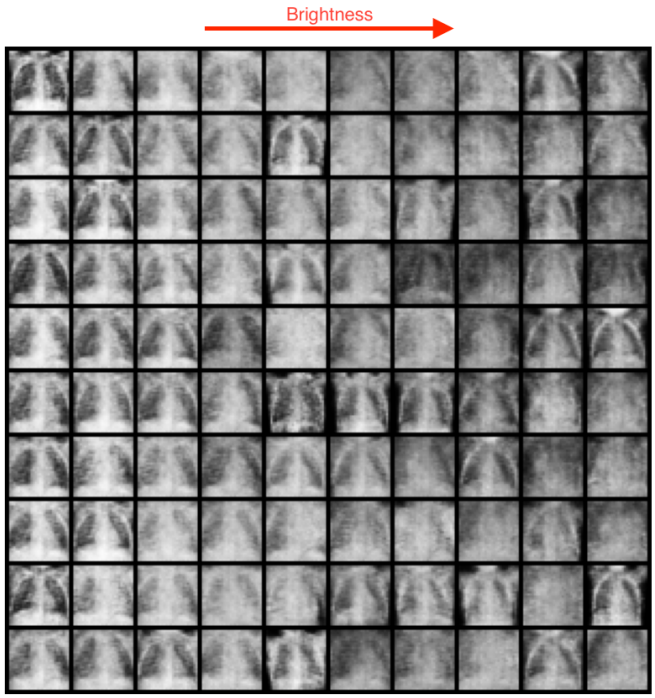


Figure 4.22. InfoGAN generated images on the ChestXray dataset. The only interpretable code we obtained is the continuous brightness variation on the horizontal axis.

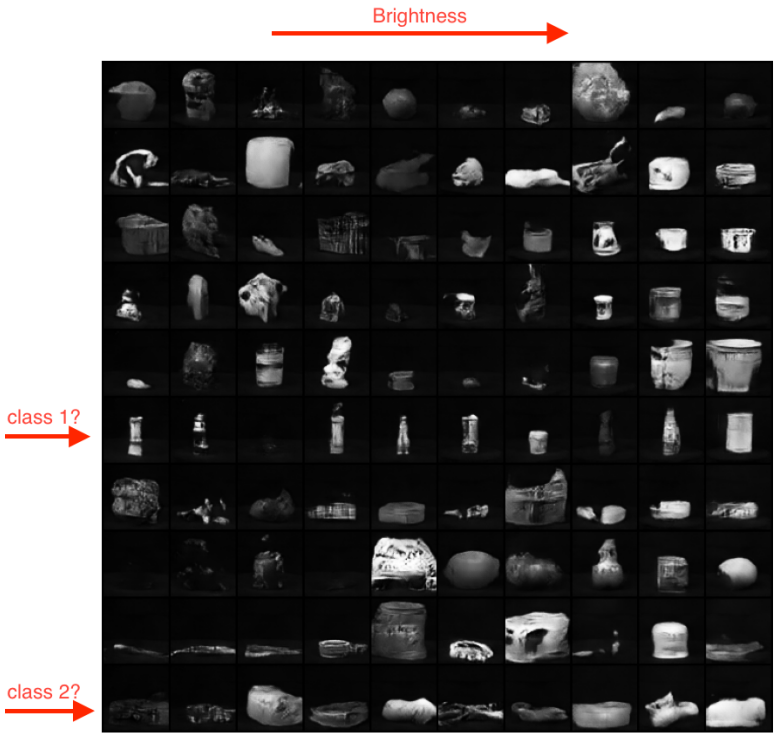


Figure 4.23. InfoGAN generated images on the Aloi dataset. A variation of brightness is present in the horizontal direction of the continuous code. Two different classes of objects seem to appear varying the discrete code.

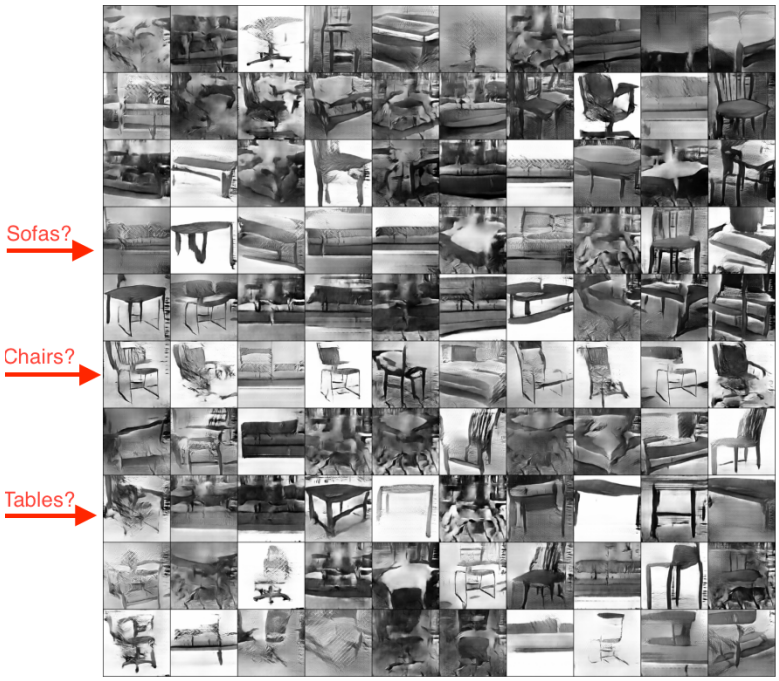


Figure 4.24. InfoGAN generated images on the Furniture dataset. Some of the produced images does not realistic. The variation of the discrete code vertically shows the stronger presence of some objects in some lines.

4.5.1 InfoGAN Explainability Grids

With InfoGAN on FashionMNIST we managed to obtain a representation that distinguishes classes with a discrete code and is able to interpolate high-level factors within each class with a continuous code. We, therefore, were able to use the generated image grids to create what we called InfoGAN Explainability Grids. Starting from a well-performing classifier on the dataset, we fed it the artificial images in order and noted the predicted class and the Softmax probability. The result is a grid like the one in figure 4.25, which can be used to get more insight on the classification model.

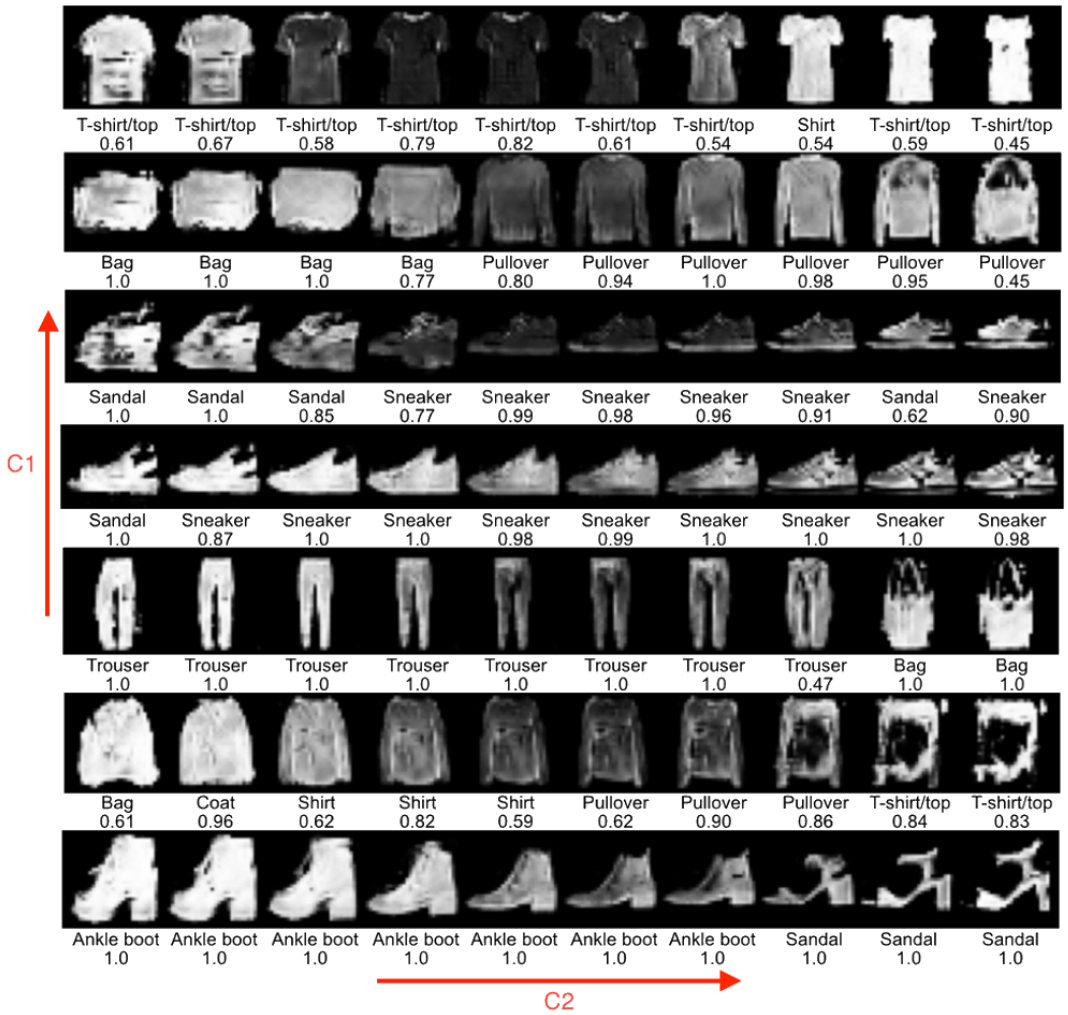


Figure 4.25. Example of a possible InfoGAN explainability grid. The C1 discrete code discriminates well between classes, while the continuous C2 produces interesting variations. Under each image, the predicted class and the Softmax score from the classifier object of study.

Some interesting results in terms of explainability have been reported here individually. The interpolation in figure 4.26 shows what looks like a dress, with a smooth interpolation in its width. This factor of variation can be assimilated to the size of the dress. Therefore we can analyze how the classifier changes its response when this factor varies. In this particular example, the classifier is responding well

since it predicts a Dress in any case, but on average, its confidence seems to be lower in the left part of the grid. Perhaps more training images with items of large size could fix the issue.

A second line we reported (fig. 4.27) shows the variation between some long long-sleeve coat and a short-sleeve dress. It is possible to see how this variation affects the predicted class and where the turning point between and dress and a coat is. In the last example (fig. 4.28) an interpolation between sneakers and sandals highlight the decision turning point. In this case, it seems that the model is deciding based on the open tip, which appears to be a reasonable decision criterion.

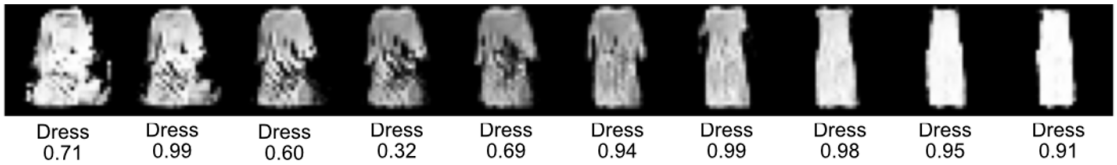


Figure 4.26. A line with a fixed value of the discrete code and an interpolation of a continuous code horizontally. The continuous code seems to capture variation of dress size. Here the response of the classifier is correct but not evenly confident.

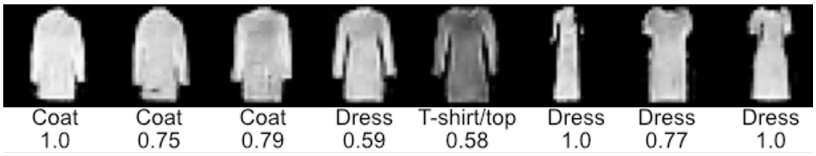


Figure 4.27. A continuous code seems to capture the length of the dress and sleeves. The turning point between Coat and Dress classes of this model can be observed.

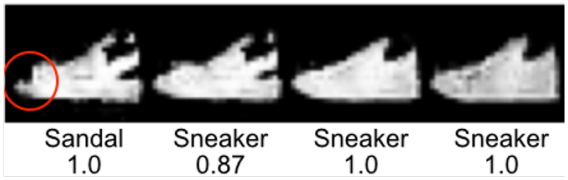


Figure 4.28. The decision turning point between a sneaker and a sandal. The only difference between the two closest images seems to be the open tip (circled in red), which seems to be a reasonable decision criterion.

4.5.2 InfoVAE Decision Maps

Intending to go even further with the explanation of the classifier's decisions, we built heatmaps that show the areas of the image that mostly define the output class. These have been obtained using InfoVAE, as described in section 4.3, by searching the perturbations of the latent space that causes the biggest variation in the classifier's output. The decision maps of some different shoes are shown in figure 4.29. Top decision importance areas correspond to distinctive features of the item, like the tip of the heel-sandal, already reported by InfoGAN Explainability grids. Lastly, figure 4.30 shows some other common fashion items. It is interesting how the last part of the long-sleeve shirt is signaled as an area of interest. The area is, as a matter of fact, important since the dataset's labels distinguish between long and short-sleeve shirts. Illustrations like these can give more insight into how a well-performing model is making its choices.

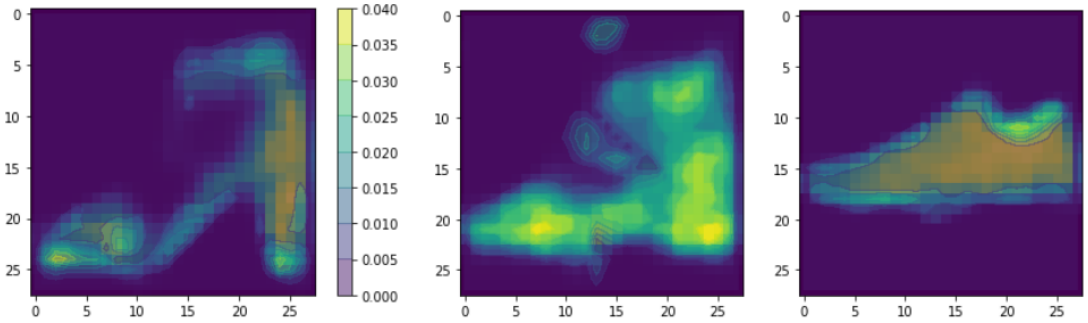


Figure 4.29. InfoVAE decision maps that highlight the areas of the image primarily affecting the classifier's decision. On the left an open sandal with a high heel, where the method highlights the area of the tip and the base of the hell as most important. Similar behavior is shown in the sandal at the center. For the sneaker (left) the most important points seem to be the one on the top border.

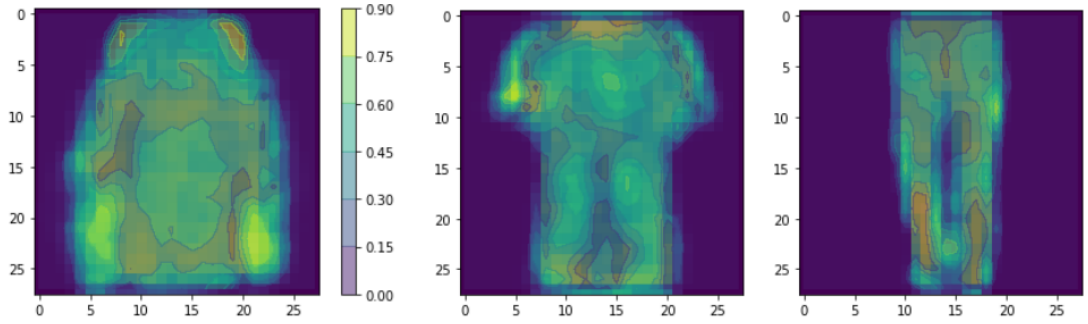


Figure 4.30. InfoVAE decision maps. On the left a long-sleeve coat, where the terminal areas of the sleeve and the shoulders are highlighted. On the center a t-shirt with a highlight on the neck. On the right, trousers where the terminal part of the legs are highlighted.

Chapter 5

Conclusions

In this work we started by examining the powerful concept of representation learning, contextualizing it into generative models and focusing our attention on the InfoGAN approach. We performed experiments with this method and applied the acquired knowledge to contribute to the AI Explainability field.

Our contribution from the first experimental part shows that training InfoGAN is not trivial, as with basic GANs, but the Dropout technique can solve training collapses, that are more frequent during the first epochs. When learning is successful, image quality is very high both in terms of fidelity and diversity. What InfoGAN struggles most with is obtaining meaningful codes, that correspond to a disentangled representation, in more articulated datasets.

With the second experimental part, we managed to produce useful Expandability Grids that show the classifier response, in terms of predicted class and Softmax score, at the variation of meaningful codes. For the FashionMNIST classifier used in our examples, we noticed how it has lower confidence with big-size dresses or how it determines if a shoe is a sandal or a sneaker. Finally, in the third part, by using InfoVAE capabilities and a search algorithm, we managed to produce interesting decision maps that highlight the areas of an image on which the classifier bases its decisions most, like the final part of the shirt's sleeves. Both of these explainability techniques are model-agnostic and do not require additional supervision. We believe that the insights these artifacts provide can be a valuable addition to the validation of a machine learning algorithm.

Future works can explore the possibilities of finding a reliable method for InfoGAN to map a generic image to an input code, such that it would be possible to extend the method we presented using VAE. We expect to see better explainability results with GANs since they produce a better image quality than VAEs. Finally, it would be interesting to improve these methods with semi-supervised learning to obtain some specific output variations, which can be employed to generate more specific explainability grids.

Bibliography

[1] D. Zender, "Mit technology review: We analyzed 16,625 papers to figure out where ai is headed next." https://www.technologyreview.com/2019/01/25/1436/we-analyzed-16625-papers-to-figure-out-where-ai-is-headed-next/, 2019, Accessed: 2020-07-08

[2] Y. Bengio, A. Courville, and P. Vincent, "Representation learning: A review and new perspectives", IEEE transactions on pattern analysis and machine intelligence, vol. 35, no. 8, 2013, pp. 1798–1828

[3] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks", European conference on computer vision, 2014, pp. 818–833

[4] I. Goodfellow, Y. Bengio, and A. Courville, "Deep learning", MIT press, 2016

[5] S. Shalev-Shwartz and S. Ben-David, "Understanding machine learning: From theory to algorithms", Cambridge university press, 2014

[6] E. D. Sontag, "Vc dimension of neural networks", NATO ASI Series F Computer and Systems Sciences, vol. 168, 1998, pp. 69–96

[7] Y. Bengio and O. Delalleau, "On the expressive power of deep architectures", International Conference on Algorithmic Learning Theory, 2011, pp. 18–36

[8] X. Glorot, A. Bordes, and Y. Bengio, "Domain adaptation for large-scale sentiment classification: A deep learning approach", ICML, 2011

[9] F. Locatello, S. Bauer, M. Lucic, G. Raetsch, S. Gelly, B. Schölkopf, and O. Bachem, "Challenging common assumptions in the unsupervised learning of disentangled representations", international conference on machine learning, 2019, pp. 4114–4124

[10] A. Kolesnikov, X. Zhai, and L. Beyer, "Revisiting self-supervised visual representation learning", Proceedings of the IEEE conference on Computer Vision and Pattern Recognition, 2019, pp. 1920–1929

[11] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space", arXiv preprint arXiv:1301.3781, 2013

[12] T. Tommasi, F. Orabona, and B. Caputo, "Learning categories from few examples with multi model knowledge transfer", IEEE transactions on pattern analysis and machine intelligence, vol. 36, no. 5, 2013, pp. 928–941

[13] Y. Ganin and V. Lempitsky, "Unsupervised domain adaptation by backpropagation", International conference on machine learning, 2015, pp. 1180–1189

[14] J. Baxter, "A bayesian/information theoretic model of learning to learn via multiple task sampling", Machine learning, vol. 28, no. 1, 1997, pp. 7–39

[15] S. Ruder, "An overview of multi-task learning in deep neural networks." https://ruder.io/multi-task/, 2017, Accessed: 2020-07-16

[16] M. I. Belghazi, A. Baratin, S. Rajeswar, S. Ozair, Y. Bengio, A. Courville, and R. D. Hjelm, "Mine: mutual information neural estimation", arXiv preprint arXiv:1801.04062, 2018

[17] R. D. Hjelm, A. Fedorov, S. Lavoie-Marchildon, K. Grewal, P. Bachman, A. Trischler, and Y. Bengio, "Learning deep representations by mutual information estimation and maximization", arXiv preprint arXiv:1808.06670, 2018

[18] C. M. Bishop, "Pattern recognition and machine learning", springer, 2006

[19] A. Y. Ng and M. I. Jordan, "On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes", Advances in neural information processing systems, 2002, pp. 841–848

[20] C. Bishop and J. Lasserre, "Generative or discriminative? getting the best of both worlds", Bayesian Statistics, vol. 8, January 2007, p. 3?23

[21] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets", Advances in neural information processing systems, 2014, pp. 2672–2680

[22] A. v. d. Oord, N. Kalchbrenner, and K. Kavukcuoglu, "Pixel recurrent neural networks", arXiv preprint arXiv:1601.06759, 2016

[23] R. Salakhutdinov and G. Hinton, "Deep boltzmann machines", Artificial intelligence and statistics, 2009, pp. 448–455

[24] Y. Bengio, E. Laufer, G. Alain, and J. Yosinski, "Deep generative stochastic networks trainable by backprop", International Conference on Machine Learning, 2014, pp. 226–234

[25] I. Goodfellow, "Nips 2016 tutorial: Generative adversarial networks", arXiv preprint arXiv:1701.00160, 2016

[26] R. Salakhutdinov, A. Mnih, and G. Hinton, "Restricted boltzmann machines for collaborative filtering", Proceedings of the 24th international conference on Machine learning, 2007, pp. 791–798

[27] S. Hochreiter, "Untersuchungen zu dynamischen neuronalen netzen", Diploma, Technische Universität München, vol. 91, no. 1, 1991

[28] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult", IEEE transactions on neural networks, vol. 5, no. 2, 1994, pp. 157–166

[29] S. Hochreiter and J. Schmidhuber, "Long short-term memory", Neural computation, vol. 9, no. 8, 1997, pp. 1735–1780

[30] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition", Proceedings of the IEEE conference on computer vision and pattern recognition, 2016, pp. 770–778

[31] D. P. Kingma and M. Welling, "Auto-encoding variational bayes", arXiv preprint arXiv:1312.6114, 2013

[32] D. P. Kingma and M. Welling, "An introduction to variational autoencoders", arXiv preprint arXiv:1906.02691, 2019

[33] R. Gómez-Bombarelli, J. N. Wei, D. Duvenaud, J. M. Hernández-Lobato, B. Sánchez-Lengeling, D. Sheberla, J. Aguilera-Iparraguirre, T. D. Hirzel, R. P. Adams, and A. Aspuru-Guzik, "Automatic chemical design using a data-driven continuous representation of molecules", ACS central science, vol. 4, no. 2, 2018, pp. 268–276

[34] I. Higgins, L. Matthey, A. Pal, C. Burgess, X. Glorot, M. Botvinick, S. Mohamed, and A. Lerchner, "beta-vae: Learning basic visual concepts with a constrained variational framework", 2016

[35] I. J. Goodfellow, "On distinguishability criteria for estimating generative models", arXiv preprint arXiv:1412.6515, 2014

[36] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks", arXiv preprint arXiv:1511.06434, 2015

[37] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift", arXiv preprint arXiv:1502.03167, 2015

[38] M. Arjovsky, S. Chintala, and L. Bottou, "Wasserstein gan", arXiv preprint arXiv:1701.07875, 2017

[39] M. Mirza and S. Osindero, "Conditional generative adversarial nets", arXiv preprint arXiv:1411.1784, 2014

[40] X. Chen, Y. Duan, R. Houthooft, J. Schulman, I. Sutskever, and P. Abbeel, "Infogan: Interpretable representation learning by information maximizing generative adversarial nets", Advances in neural information processing systems, vol. 29, 2016, pp. 2172–2180

[41] R. C. Fong and A. Vedaldi, "Interpretable explanations of black boxes by meaningful perturbation", Proceedings of the IEEE International Conference on Computer Vision, 2017, pp. 3429–3437

[42] H. Uzunova, J. Ehrhardt, T. Kepp, and H. Handels, "Interpretable explanations of black box classifiers applied on medical images by meaningful perturbations using variational autoencoders", Medical Imaging 2019: Image Processing, 2019, p. 1094911

[43] K. Schutte, O. Moindrot, P. Hérent, J.-B. Schiratti, and S. Jégou, "Using stylegan for visual interpretability of deep learning models on medical images", arXiv preprint arXiv:2101.07563, 2021

[44] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting", The journal of machine learning research, vol. 15, no. 1, 2014, pp. 1929–1958

[45] T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen, "Improved techniques for training gans", arXiv preprint arXiv:1606.03498, 2016

[46] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision", Proceedings of the IEEE conference on computer vision and pattern recognition, 2016, pp. 2818–2826

[47] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database", CVPR09, 2009

[48] M. Lucic, K. Kurach, M. Michalski, S. Gelly, and O. Bousquet, "Are gans created equal? a large-scale study", arXiv preprint arXiv:1711.10337, 2017

[49] S. Zhao, J. Song, and S. Ermon, "Infovae: Information maximizing variational autoencoders", arXiv preprint arXiv:1706.02262, 2017

[50] Fiorino, "infogan-notebook." https://colab.research.google.com/drive/1GMpAhnM76glnSYM-8BIUHfkb_0xdVknL?usp=sharing, 2020

[51] Natsu6767, "Infogan-pytorch." https://github.com/Natsu6767/InfoGAN-PyTorch, 2019

[52] mseitzer, "pytorch-fid." https://github.com/mseitzer/pytorch-fid, 2020
[53] Y. LeCun and C. Cortes, "MNIST handwritten digit database", 2010
[54] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms", arXiv preprint arXiv:1708.07747, 2017
[55] D. Kermany, K. Zhang, M. Goldbaum, *et al.*, "Labeled optical coherence tomography (oct) and chest x-ray images for classification", Mendeley data, vol. 2, no. 2, 2018
[56] N. Akki, "Furniture detector." https://www.kaggle.com/akkithetechie/furniture-detector/metadata, 2018
[57] J.-M. Geusebroek, G. J. Burghouts, and A. W. Smeulders, "The amsterdam library of object images", International Journal of Computer Vision, vol. 61, no. 1, 2005, pp. 103–112
[58] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng, "Reading digits in natural images with unsupervised feature learning", 2011
[59] Z. Liu, P. Luo, X. Wang, and X. Tang, "Deep learning face attributes in the wild", Proceedings of International Conference on Computer Vision (ICCV), December 2015