

# DEEP GENERATIVE MODELS

---

DANILO COMMINIELLO

GENERATIVE DEEP LEARNING 2021/2022

June 7, 2022



**SAPIENZA**  
UNIVERSITÀ DI ROMA

- Deep representation learning is one of the key concepts to overcome the generative modeling challenges.
- Merging *generative modeling* and *deep learning* leads to deep generative models.
- Deep generative models are built by stacking layers, as any deep neural network.

# Table of contents

## ① DEEP LEARNING FOR GENERATIVE MODELING

## ② TAXONOMY OF DEEP GENERATIVE MODELS

## ① DEEP LEARNING FOR GENERATIVE MODELING

---

Deep Learning Solves Generative Modeling Challenges

Deep Neural Networks to Parameterize Generative Models

From Dense to Convolutional Layers

Batch Normalization

Latent Variable Models and Hidden States

Recurrent Neural Networks

# Deep learning solves generative modeling challenges

A generative model, in order to be successful and impressive, must **generate examples** appearing to belong to  $p_{\text{data}}$  without showing existing overlaps with original data.

In presence of a **big amount of data**, it is not easy for a model cope with the high degree of conditional dependence between features.

The larger the input data space, the more difficult it is to produce an output that satisfies the generation constraints.

**Deep learning** is the key to solving both of these challenges due to its ability to form its own features in a lower-dimensional space.

# Representation learning

**Representation learning** refers to the ability to describe each observation in the training set using some **low-dimensional latent space**.





Then, a function is learned to **map** a point in the latent space into the original domain.

**Deep learning** provides the ability to learn the often highly complex mapping function in a variety of ways.

So each point in the latent space is the *representation* of some high-dimensional image.

# Structured and unstructured data

STRUCTURED DATA				
id	age	gender	height (cm)	location
0001	54	M	186	London
0002	35	F	166	New York
0003	62	F	170	Amsterdam
0004	23	M	164	London
0005	25	M	180	Cairo
0006	29	F	181	Beijing
0007	46	M	172	Chicago

UNSTRUCTURED DATA		
		This service is terrible!
		Your website is great!
images	audio	text

**Figure 1:** There is spatial structure to an image, temporal structure to a recording, and both spatial and temporal structure to video data, but since the data does not arrive in columns of features, it is considered *unstructured*. When our data is unstructured, individual pixels, frequencies, or characters are almost entirely uninformative [1].

# Deep learning for unstructured data

Deep learning is a class of machine learning algorithm that uses multiple stacked layers of processing units to learn high-level representations from unstructured data.

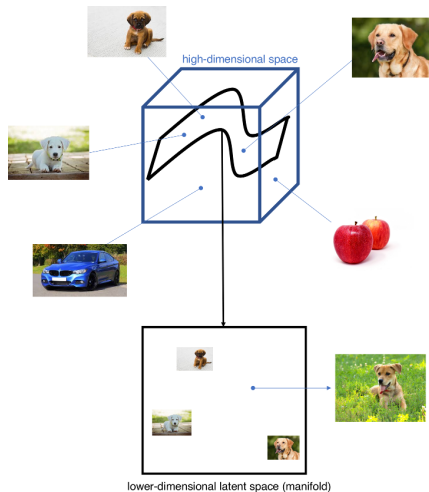
The real power of deep learning, especially with regard to generative modeling, comes from its ability to work with unstructured data.

Most often, we want to generate unstructured data such as new images or original strings of text.

This is the reason why deep learning has had such a profound impact on the field of generative modeling.



# Low-dimensional latent space



**Figure 2:** Representation learning actually learns which features are most important to describe the given observations. The cube represents the extremely high-dimensional space of all images; representation learning tries to find the lower-dimensional latent subspace or *manifold* on which particular kinds of image lie (for example, the dog manifold) [1].

# Learning representation by deep networks

The way human brain learns is inherently hierarchical, thus being able to provide a *deeper* representation.

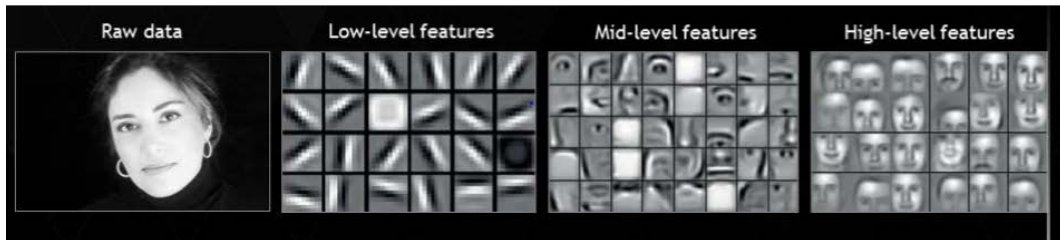


Figure 3: Deep learning implies learning features of features of features...

Image source: [Analytics Vidhya](#).

# Deep neural networks to parameterize generative models

Neural networks are flexible and powerful and, therefore, they are widely used to parameterize generative models.

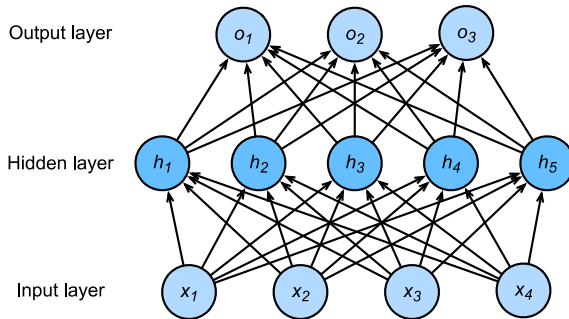
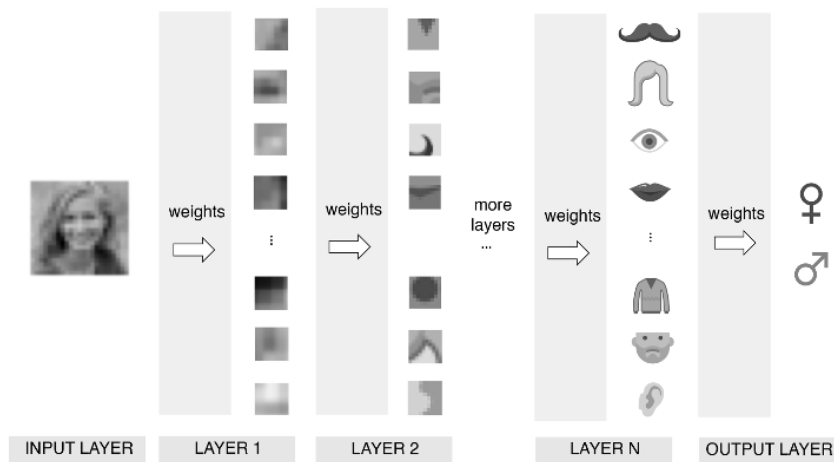


Figure 4: A **multilayer perceptron** (MLP) network can be obtained by stacking one or more **hidden layers**. This example contains a hidden layer with 5 hidden units in it. Note that layers are both fully connected [2].

# Conceptual diagram of deep neural networks



**Figure 5:** In this example, layer 1 consists of units that activate more strongly when they detect particular basic properties of the input image, such as edges. The output is then passed to the units of layer 2, which are able to detect slightly more complex features. The final network outputs are a set of numbers that can be converted into probabilities, to represent the chance that the original input belongs to one of  $n$  categories [1].

# Stacking of layers

Let us consider the MLP of Fig. 4 with a **minibatch** of inputs  $\mathbf{X}$ .

The calculations to produce outputs from an MLP with two hidden layers can thus be expressed:

$$\mathbf{H}_1 = \sigma(\mathbf{W}_1\mathbf{X} + \mathbf{b}_1),$$

$$\mathbf{H}_2 = \sigma(\mathbf{W}_2\mathbf{H}_1 + \mathbf{b}_2),$$

$$\mathbf{O} = \text{softmax}(\mathbf{W}_3\mathbf{H}_2 + \mathbf{b}_3).$$

The `softmax` nonlinearity on the output layer can be used to choose one of the output classes.

The nonlinearities  $\sigma$  are **activation functions**, enabling a *meaningful* transforms stacking.

# Activation functions

An **activation function**  $\sigma$  decides whether a neuron should be *activated or not* by calculating the weighted sum and further adding bias with it.

They are **differentiable operators** to transform input signals to outputs, while most of them add **nonlinearity**.

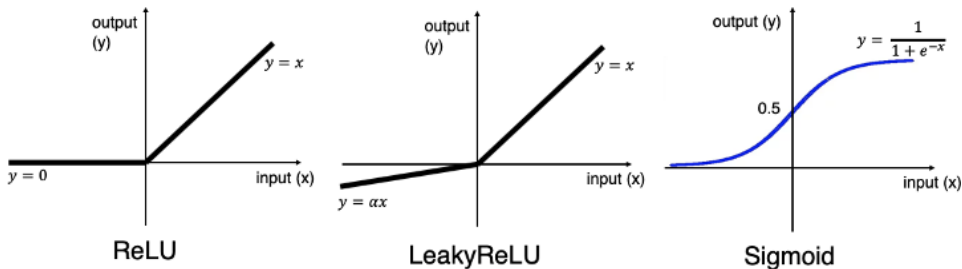


Figure 6: The ReLU, LeakyReLU, and sigmoid activation functions [1].

# Forward and backward propagation

**Forward propagation** refers to the calculation and storage of intermediate variables (including outputs) for the neural network, from the input layer to the output layer.

**Backpropagation** refers to the method of calculating the gradient of neural network parameters.

The **order of calculations** are reversed relative to those performed in forward propagation, since we need to start with the outcome of the compute graph and work our way towards the parameters.

When training networks, forward and backward propagations **depend on each other**.

# Model complexity vs underfitting and overfitting

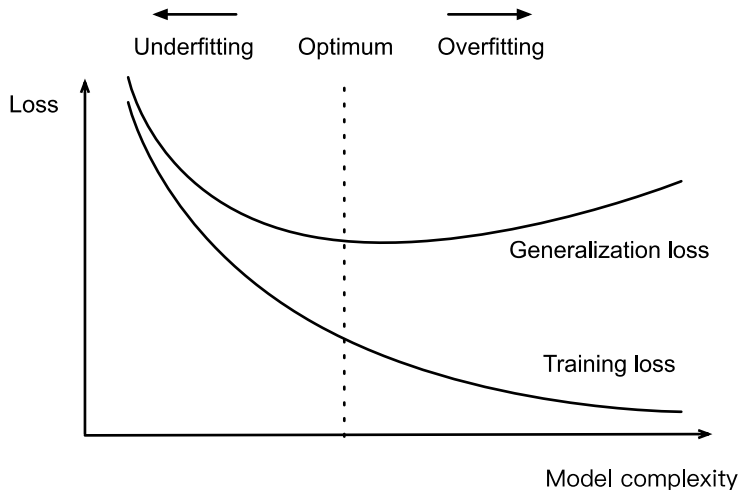


Figure 7: Relation between model complexity and underfitting/overfitting phenomena [2].



# Regularization to reduce overfitting

Now that we have characterized the problem of overfitting, we can introduce some standard techniques for **regularizing models**.

Recall that we can always mitigate overfitting by going out and **collecting more training data**, that can be costly, time consuming, or entirely out of our control, making it impossible in the short run.

When it is not possible we only can focus on **regularization techniques**.

As said, we could **limit our model's capacity** to avoid overfitting.

However, simply tossing aside features can be too blunt a hammer for the job. Thus we often need a more fine-grained tool for adjusting function complexity.

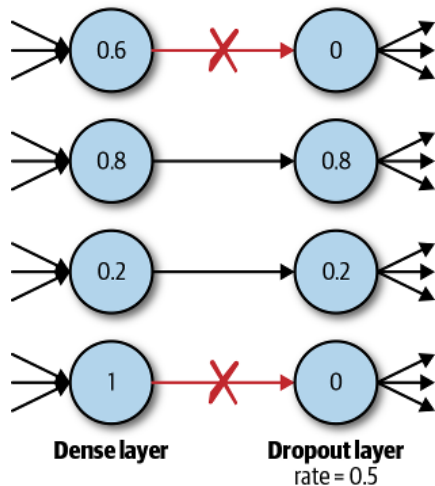
# Dropout

Due to the presence of **very large datasets** (more examples than features), deep learning methods aim at **learning interactions** among groups of features to avoid overfitting.

In that sense, the **dropout** method involves **injecting noise into each layer** during forward propagation.

Throughout training, on each iteration, standard dropout consists of **zeroing out some fraction of nodes in each layer** (typically 50%) before calculating the subsequent layer [3].

# Dropout in practice



**Figure 8:** A dropout layer. During training, each dropout layer chooses a random set of units from the preceding layer and sets their output to zero [1].

# Leveraging high-dimensional dependence between features

Until now, we have dealt with 2D images simply flattening each image into a **1D vector**, and fed it into a **fully-connected network**.

This network is **invariant** to the order of their inputs.

However, if we consider the processing of **high-quality images**, learning the parameters of this network may turn out to be impossible.

Ideally, we would leverage the prior knowledge that **nearby pixels are more related to each other**.

**Convolutional neural networks (CNNs)** are a powerful family of neural networks that were designed precisely for this purpose.

# Exploiting spatial structure

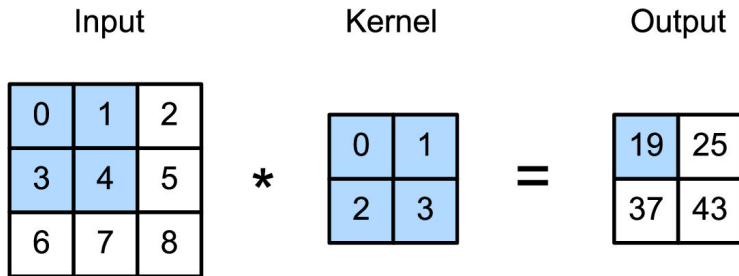
We can derive some properties for a **good object detector method** using the **spatial arrangement** of pixels on an image:

- **Translation invariance**: the same object can appear equivalently on any part of the image.
- **Locality**: an object should depend only on a *local region* of the overall image, without regard for what else is happening in the image at greater distances.

These assumptions bring forth immediately the idea of **filters** and **image convolutions**.

# The cross-correlation operator

In a convolutional layer, an *input array* and a *correlation kernel array*, also called *kernel* or *filter* are combined to produce an output array through a **cross-correlation operation**.



**Figure 9:** The input is a 3 × 3 2D array, and the kernel array is 2 × 2. The shape of the kernel window (also known as the *convolution window*) is given precisely by the height and width of the kernel. It is possible to notice the correspondence between cross-correlation and convolution. The latter can be easily obtained by simply flipping the kernel from the bottom left to the top right [2].

# Image convolution with a kernel

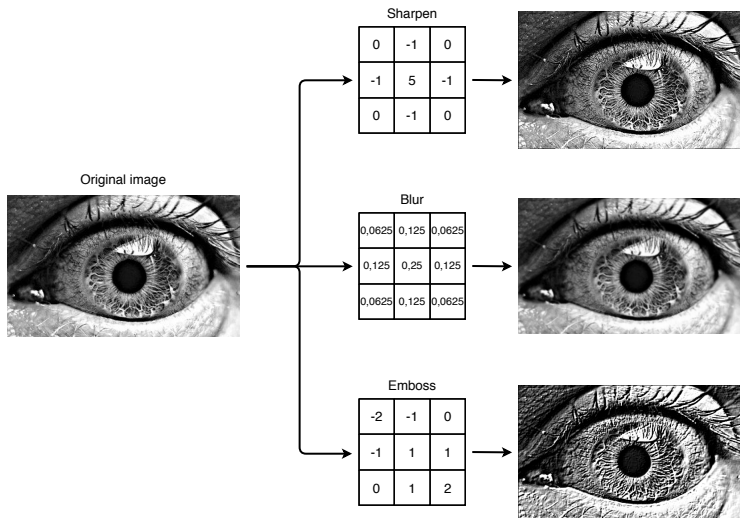


Figure 10: For an interactive visualization, see <http://setosa.io/ev/image-kernels/>.

# Output shape of a convolutional layer

The **output shape** of the convolutional layer is determined by the shapes of input and convolution kernel window, as shown in Fig. 9.

In some cases we might want to **control the size of the output**:

- In general, kernels generally have width and height greater than 1, thus, after applying many successive convolutions, the output results much smaller than the input. **Padding** handles this issue.
- In some cases, we want to reduce the resolution drastically if we find an original input resolution to be unwieldy. **Strides** can help in these instances.



# Pooling operators

Like convolutional layers, **pooling operators** consist of a fixed-shape window that is slid over all regions in the input according to its *stride*, computing a single output for each location traversed by the fixed-shape window (or *pooling window*).

The pooling layer contains **no parameters** (there is no *filter*).

Instead, pooling operators are *deterministic*, typically calculating either the *maximum* or the *average* value of the elements in the pooling window.

These operations are called **maximum pooling** (or also **max pooling**) and **average pooling**, respectively.

# Example of a convolutional neural networks

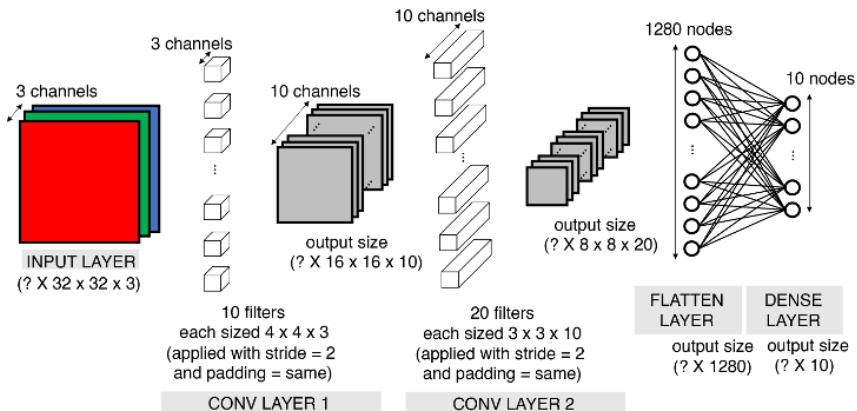


Figure 11: A diagram of a convolutional neural network [1].

# Batch normalization

An effective technique that consistently accelerates the convergence of deep nets is represented by the **batch normalization** (BN) [4].

BN is applied to individual layers as follows:

- first **compute its activations** as usual,
- **normalize the activations** of each node by subtracting its mean and dividing by its standard deviation for the current *minibatch*.

It is precisely due to this *normalization* based on *batch* statistics that **batch normalization** derives its name.

BN has many beneficial side effects, primarily that of *regularization*.

# Formulation of batch normalization

Formally, BN transforms the activations at a given layer  $\mathbf{x}$  according to the following expression:

$$\text{BN}(\mathbf{x}) = \gamma \odot \frac{\mathbf{x} - \hat{\mu}}{\hat{\sigma}} + \beta$$

where  $\hat{\mu}$  is the minibatch sample mean and  $\hat{\sigma}$  is the minibatch sample variance.

We commonly include coordinate-wise scaling coefficients  $\gamma$  and offsets  $\beta$ .

Consequently, the activation magnitudes for intermediate layers cannot diverge during training because BN actively centers and rescales them back to a given mean and size (via  $\mu$  and  $\sigma$ ).

# Batch normalization layers

Batch normalization implementations for fully-connected layers and convolutional layers are slightly different.

For fully-connected layers, we usually insert BN after the affine transformation  $f_{\theta}(\cdot)$  and before the nonlinear activation function  $\phi(\cdot)$ :

$$\mathbf{h} = \phi(\text{BN}_{\beta, \gamma}(f_{\theta}(\mathbf{x})))$$

For convolutional layers, we typically apply BN after the convolution and before the nonlinear activation function.

When the convolution has multiple output channels, BN is carried out for *each* of the outputs of these channels.

# Latent variable models

In sequence data, the **conditional probability** of a sample  $x_t$  at position  $t$  only depends on the  $n - 1$  previous samples.

To check the possible effect of earlier samples than  $t - (n - 1)$ , we need to **increase  $n$** .

However, the **number of model parameters** would also increase exponentially with it, as we need to store  $|V|^n$  numbers for a vocabulary  $V$ .

Hence, rather than modeling  $p(x_t \mid x_{t-1}, \dots, x_{t-n+1})$  we use a **latent variable model**:

$$p(x_t \mid x_{t-1}, \dots, x_1) \approx p(x_t \mid x_{t-1}, h_t)$$

where  $h_t$  is a **latent variable** that stores the sequence information.

# Hidden state variables

A latent variable  $h_t$  is also called as *hidden variable* or *hidden state*.

The hidden state at time  $t$  could be computed based on both  $x_t$  and  $h_{t-1}$ :

$$h_t = f(x_t, h_{t-1}).$$

After all,  $h_t$  could simply store *all the data it observed so far*. But it could potentially makes both *computation and storage expensive*.

Note that we also use  $h$  to denote by the number of hidden units of a hidden layer, but they are *not to be confused*!

*Recurrent neural networks* are neural networks with hidden states.

# Neural network with hidden states: the recurrent neural network

Now consider a **neural network with hidden states** with a minibatch  $\mathbf{X}_t \in \mathbb{R}^{n \times d}$  and a hidden variable  $\mathbf{H}_t \in \mathbb{R}^{n \times h}$  at time step  $t$ .

Unlike the MLP, here we save  $\mathbf{H}_{t-1}$  from the previous time step and introduce a **new weight parameter**  $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$  to describe how to use  $\mathbf{H}_{t-1}$  in the current time step:

$$\mathbf{H}_t = \phi(\mathbf{X}_t \mathbf{W}_{xh} + \mathbf{H}_{t-1} \mathbf{W}_{hh} + \mathbf{b}_h). \quad (1)$$

The neural network described by (1) and containing the additional term  $\mathbf{H}_{t-1} \mathbf{W}_{hh}$  denoting the hidden state, is called **recurrent neural network** (RNN).

The **output of the RNN** is defined like an MLP without hidden states:  $\mathbf{O} = \mathbf{H} \mathbf{W}_{hq} + \mathbf{b}_q$ .

It is worth noting that the **number of RNN model parameters** does not grow as the number of timesteps increases.



## 2 TAXONOMY OF DEEP GENERATIVE MODELS

---

How to Formulate Deep Generative Models

Autoregressive Models

Flow-Based Models

Latent Variable Models

Energy-Based Models

Summary of Deep Generative Models

# How to formulate deep generative models

After highlighting the importance deep learning for generative modeling, we need to **formulate deep generative models**. This means we want to know how to express  $p(\mathbf{x})$ .

A possible taxonomy of deep generative models is shown in the picture.

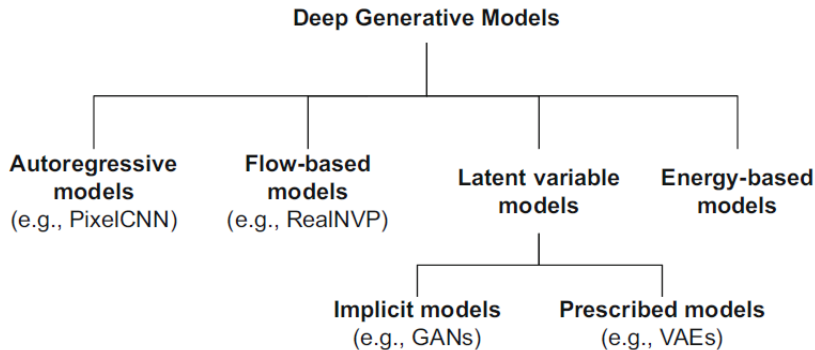


Figure 12: Taxonomy of the most popular generative deep learning models.

# Autoregressive models

The first group of deep generative models utilizes the idea of **autoregressive (AR) modeling**, i.e., the distribution over  $\mathbf{x}$  is represented in an autoregressive manner:

$$p(\mathbf{x}) = p(x_0) \prod_{i=0}^D p(x_i | \mathbf{x}_{<i})$$

where  $\mathbf{x}_{<i}$  denotes all  $\mathbf{x}$ 's up to the index  $i$ .

**Causal convolutions** will solve the computational inefficiency of conditional distributions  $p(x_i | \mathbf{x}_{<i})$ .

# Flow-based models

These models rely on a change of variable to express a density  $p(\mathbf{x})$  with an **invertible transformation**:

$$p(\mathbf{x}) = p(\mathbf{z} = f(\mathbf{x})) |\mathbf{J}_{f(\mathbf{x})}|,$$

where  $\mathbf{J}_{f(\mathbf{x})}$  denotes the Jacobian matrix, and the function  $f(\cdot)$  can be *parameterized* by a deep neural network.

Several attempts have been made in the literature to perform the change of variables.

All generative models that take advantage of the change of variables formula are referred to as **flow-based models**.

# Latent variable models

The idea behind **latent variable models** is to assume a *lower-dimensional latent space* and the following generative process:

$$\mathbf{z} \sim p(\mathbf{z})$$

$$\mathbf{x} \sim p(\mathbf{x}|\mathbf{z})$$

meaning that the latent variables correspond to *hidden factors* in data, and the conditional distribution  $p(\mathbf{x}|\mathbf{z})$  could be treated as a *generator*.

The most widely known latent variable model is the **probabilistic Principal Component Analysis** (pPCA) [5] where  $p(\mathbf{z})$  and  $p(\mathbf{x}|\mathbf{z})$  are Gaussian distributions, and the dependency between  $\mathbf{z}$  and  $\mathbf{x}$  is linear.

A nonlinear extension of the pPCA with arbitrary distributions is the **Variational AutoEncoder** (VAE) framework [6].

# Implicit latent variable models

So far, AR models, flows, pPCA-based models and VAEs are probabilistic models with the objective function being the **log-likelihood function** that is closely related to using the **Kullback-Leibler divergence** between the data distribution and the model distribution.

A different approach utilizes an **adversarial loss** in which a discriminator  $D(\cdot)$  determines a difference between real data and synthetic data provided by a generator in the implicit form, namely:

$$p(\mathbf{x}|\mathbf{z}) = \delta(\mathbf{x} - G(\mathbf{z})),$$

where  $\delta(\cdot)$  is the Dirac function.

This group of models is called **implicit models**, and **Generative Adversarial Networks** (GANs) [7] become one of the first successful deep generative models for synthesizing realistic-looking objects (e.g., images).

Physics provide an interesting perspective on defining a group of generative models through defining an energy function,  $E(\mathbf{x})$ , and, eventually, the **Boltzmann distribution**:

$$p(\mathbf{x}) = \frac{\exp\{-E(\mathbf{x})\}}{Z},$$

where  $Z = \sum_{\mathbf{x}} \exp\{-E(\mathbf{x})\}$  is the **partition function**.

In other words, the distribution is defined by the **exponentiated energy function** that is further normalized to obtain values between 0 and 1 (i.e., probabilities) [8].

# Summary of deep generative models

Generative models	Training	Likelihood	Sampling	Compression	Representation
Autoregressive models	Stable	Exact	Slow	Lossless	No
Flow-based models	Stable	Exact	Fast/slow	Lossless	Yes
Implicit models	Unstable	No	Fast	No	No
Prescribed models	Stable	Approximate	Fast	Lossy	Yes
Energy-based models	Stable	Unnormalized	Slow	Rather not	Yes

Figure 13: Here is a comparison of all four groups of models, with a distinction between **implicit** latent variable models and **explicit** (or *prescribed*) latent variable models, using arbitrary criteria.



- We will focus on a particular generative approach to estimate the distribution of the data  $p(\mathbf{x})$ .
  - A potential solution is utilizing a single, shared model for the conditional distribution.
- We introduce **generative autoregressive models** for sequential data.
  - We show how to **parameterize autoregressive models** by neural networks in order to build generative autoregressive models.

# References I

- [1] D. Foster, *Generative Deep Learning – Teaching Machines to Paint, Write, Compose and Play*. O'Reilly Media, Inc., Jun. 2019.
- [2] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola, *Dive Into Deep Learning*, 2020.
- [3] 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, pp. 1929–1958, 2014.
- [4] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” *arXiv preprint arXiv:1502.03167*, 2015.
- [5] M. E. Tipping and C. M. Bishop, “Probabilistic principal component analysis,” *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, vol. 61, no. 3, pp. 611–622, 1999.
- [6] D. P. Kingma and M. Welling, “An introduction to variational autoencoders,” *Foundations and Trends in Machine Learning*, vol. 12, no. 4, pp. 307–392, Nov. 2019.
- [7] I. J. Goodfellow, “Generative adversarial networks (GANs) – NIPS 2016 tutorial,” Dec. 2016.

# References II

- [8] Y. LeCun, S. Chopra, R. Hadsell, M. Ranzato, and F. Huang, "A tutorial on energy-based learning," *Predicting structured data*, vol. 1, 2006.
- [9] S. Bai, J. Z. Kolter, and V. Koltun, "An empirical evaluation of generic convolutional and recurrent networks for sequence modeling," *arXiv preprint arXiv:1803.01271*, 2018.
- [10] T. Karras, S. Laine, and T. Aila, "A style-based generator architecture for generative adversarial networks," *arXiv preprint arXiv:1812.04948v1*, Dec. 2018.
- [11] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," *preprint*, 2019.
- [12] E. Schönfeld, B. Schiele, and A. Khoreva, "A U-Net based discriminator for generative adversarial networks," in *IEEE/CVF Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2020, pp. 8207–8216.
- [13] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Boston, MA, Jun. 2015, pp. 1–9.

# References III

- [14] A. Vahdat and J. Kautz, "NVAE: A deep hierarchical variational autoencoder," in *Neural Information Processing Systems (NeurIPS)*, 2020.
- [15] D. Zhang, S. Mishra, E. Brynjolfsson, J. Etchemendy, D. Ganguli, B. Grosz, T. Lyons, J. Manyika, J. C. Niebles, M. Sellitto, Y. Shoham, J. Clark, and R. Perrault, "The AI index 2021 annual report," *arXiv preprint arXiv:2103.06312*, Mar. 2021.
- [16] J. M. Tomczak, *Deep Generative Modeling*. Springer Nature Switzerland AG, 2022.

# DEEP GENERATIVE MODELS

GENERATIVE DEEP LEARNING  
2021/2022

**DANILO COMMINIELLO**

PhD Course in Information and Communication Technology (ICT)

<http://danilocomminiello.site.uniroma1.it>

[danilo.comminiello@uniroma1.it](mailto:danilo.comminiello@uniroma1.it)