BSC AI Factory

Generative and Self-Supervised ML

Autoregressive Models

Javier Béjar - UPC

16/1/2026

BSC AI Factory Partners

Affiliated entities

Motivation

- We are interested in models able to learn a probability distribution $p(x)$ from samples of a dataset
- With a probability distribution we can
  - Generate new samples: Images, video, speech, text...
  - Obtain representations able to compress the data (codes)
  - Compute the probability of new data for applications (e.g.: anomaly detection)
- Likelihood-based models allow estimating $p(x)$ from samples $x_1, \ldots, x_n$ given a parametrized distribution function

- For real world applications we want
  - To be able to model complex and high dimensional data (e.g.: images, video, text)
  - To be efficient in terms of model training and representation
  - To be able to represent a large variety of problems (expressive), and to be able to generalize from examples
  - To be able to generate good samples efficiently
  - To have a good compression ratio of the data (representation size/amount of data)

Parameterized Distributions

Parameterized functions

- Our goal is to approximate $p_{data}(x)$ using a parameterized function $p_\theta(x)$
- We need to learn the parameters θ so $p_\theta(x) \approx p_{data}(x)$
- Basically we need to solve an optimization problem over the parameters that:

$$\arg\min_\theta loss(\theta, x_1, \ldots, x_n)$$

- There are different approaches that we can use, but we have to take into account
  - We need to work with large datasets
  - We need to obtain a function that gets as close as possible to the data distribution
  - We need the function to generalize (we have a limited sample, high dimensionality)

- Maximum likelihood parameter estimation guarantees that for a family of functions expressive enough and given enough data, it results in the parameters that generate the data
- We are optimizing the loss

$$\arg\min_{\theta} loss(\theta, x_1, \ldots, x_n) = -\frac{1}{n}\sum_{i=1}^{n} \log p_\theta(x_i)$$

- This is equivalent to minimizing the Kullback-Leibler divergence (KL) between the empirical data distribution and the model

$$KL(\hat{p}_{data}||p_\theta) = \mathbb{E}_{x\sim\hat{p}_{data}}[-\log p_\theta(x)] - H(\hat{p}_{data})$$

- Maximum likelihood estimation can be solved using Stochastic Gradient Descent (SDG)
- SDG minimizes the expectations, it solves for a differentiable function f with parameters θ

$$\arg\min_{\theta} \mathbb{E}[f(\theta)]$$

- For Maximum Likelihood we have:

$$\arg\min_{\theta} \mathbb{E}_{x\sim\hat{p}_{data}}[-\log p_{\theta}(x)]$$

- The advantage of SDG is that it works with large datasets, and we can use neural networks as estimation functions

Designing the estimation function

- We are going to use neural networks as estimators, but not any architecture would do
- We need the network to be a proper probability distribution, basically
  - The sum of the probabilities for all the data sums to 1
  - The probability for any example must be ≥ 0
- We also need $\log p_\theta(x)$ to be easy to compute and differentiable respect to θ

- Bayesian networks are probabilistic graphical models that factorize a joint probability distribution function using a Directed Acyclic Graph over the variables
- Each node of the graph models the joint distribution of a subset of the variables of the problem, the variable conditioned to its parents in the graph $p(A|B, C, \dots)$
- We can model this conditioned distribution as a neural network,
  - The input of the network are the parents of the variable
  - The output is the variable that is conditioned
  - The network outputs the probability distribution of the variable

Autoregressive Models

- Given a Bayesian network structure, to transform the conditional distribution to a neural network will obtain a tractable log likelihood and gradients

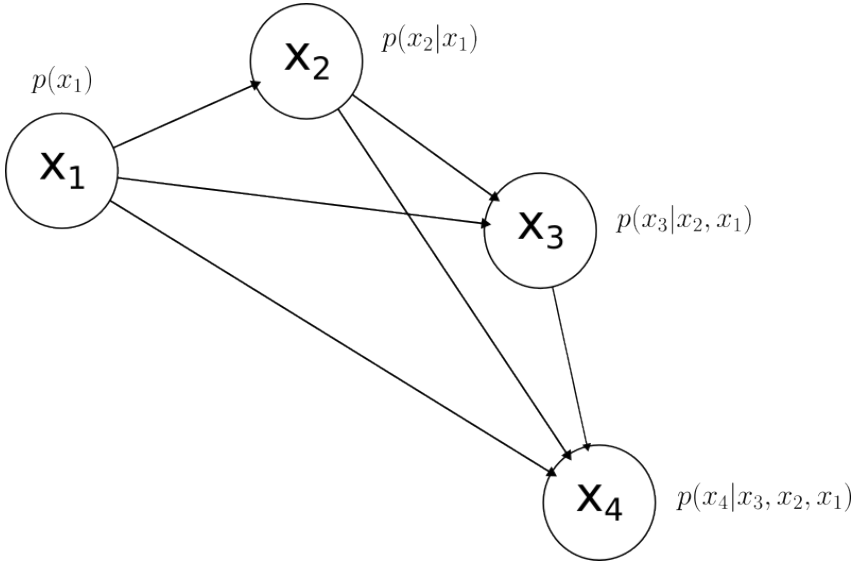
$$\log p_\theta(x) = \sum_{i=1}^{d} \log p_\theta(x_i | parents(x_i))$$

- Given a joint probability distribution the product rule allows expressing the probability function as the product of conditional probabilities

$$\log p(x) = \sum_{i=1}^{d} \log p(x_i | x_{i-1}, , \ldots, x_1)$$

- This is called an autoregressive model
- We can represent this bayesian network using neural networks for the conditional probabilities obtaining a tractable maximum likelihood problem

Autoregressive Bayesian Network



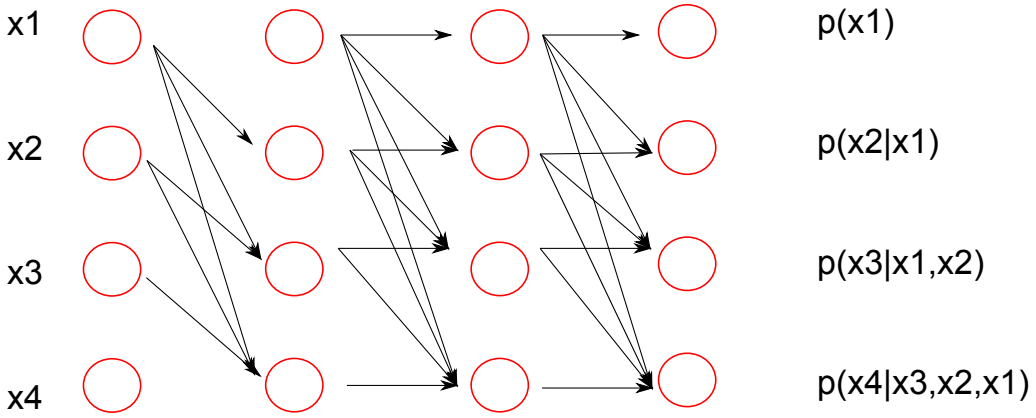
- In order to maintain a low computational cost we should be able to parallelize the training adjusting all the conditionals at the same time
- Three solutions:
  - Masked MLPs: MADE
  - Masked Convolutions
  - Transformer architectures with autoregressive masks (attention)

- Sampling is one of the weak points of these models, since it can not be paralellized
- Sampling Algorithm:
  1. Sample x_1 from $p(x_1)$
  2. Input sampled variables 1 to $i-1$ to obtain $p(x_i|x_1, \ldots, x_{i-1})$
  3. Sample variable x_i
  4. Repeat from 2 until d

Masked MLPs

- The goal is to define an architecture based on autoencoders that satisfies the autoregressive property
- We can use masks to eliminate the connections in the autoencoder so for each output x_d there are only paths that connect the $x_{<d}$ inputs
- These masks correspond to matrices that have zeros on the positions of the connections that violate that constraint
- The output layer for each variable has a sigmoid activation function for binary variables (so a $[0, 1]$ value is obtained) or a softmax for multivalued variables

MADE - Connection dependencies



MADE - Architecture connections rules

- Each neuron of the input and output layers will have a number from 1 to D
- Each layer of the hidden layers will have a number from 1 to $D-1$
- For each hidden layer each neuron only can be connected to paths that include neurons with a number lower or equal than its number
- For the output layer, each neuron is connected only to neurons with a number less than its number
- The input layer will have a neuron that will not be connected to anything and the output layer will have a neuron that will not receive any connection

- For hidden layers the computation performed is:

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

- For the output layer

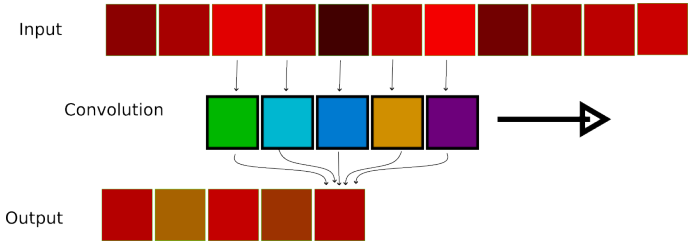
$$\hat{x} = \sigma(c + (V \odot M^V)h^L(x))$$

- Training uses different orders for the variables
- An ensemble of models can be built, averaging the results
- Different patterns of connections that follow the autoregressive order can be trained at the same time.

Masked Convolutional Networks

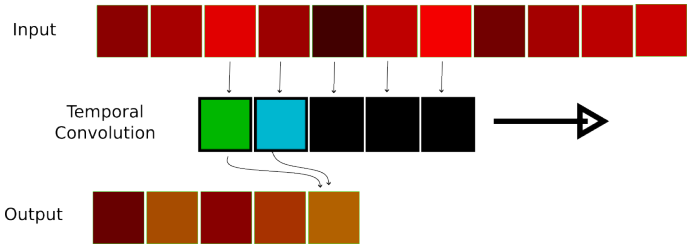
1D Convolutions

- Convolutions can be computed efficiently
- We can have several convolutional filters and have many layers that combine the results of the filters of previous layers
- Unidimensional convolutions allow focusing on a few elements of a sequence
- Each convolutional filter is applied to a subsequence: $[1 \times k]$ filter

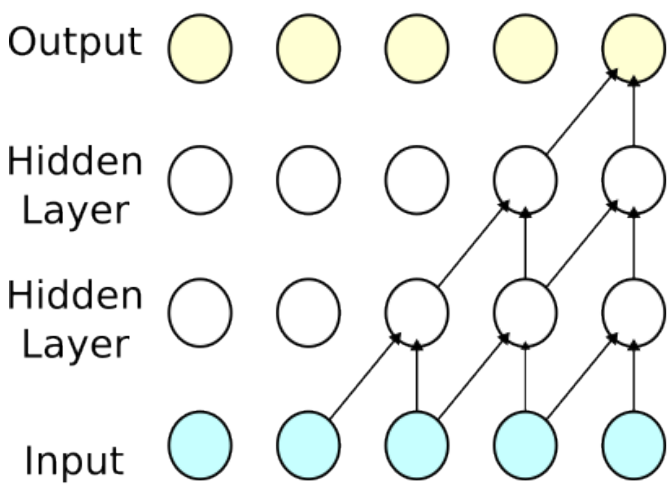


Masked Temporal 1D Convolutions

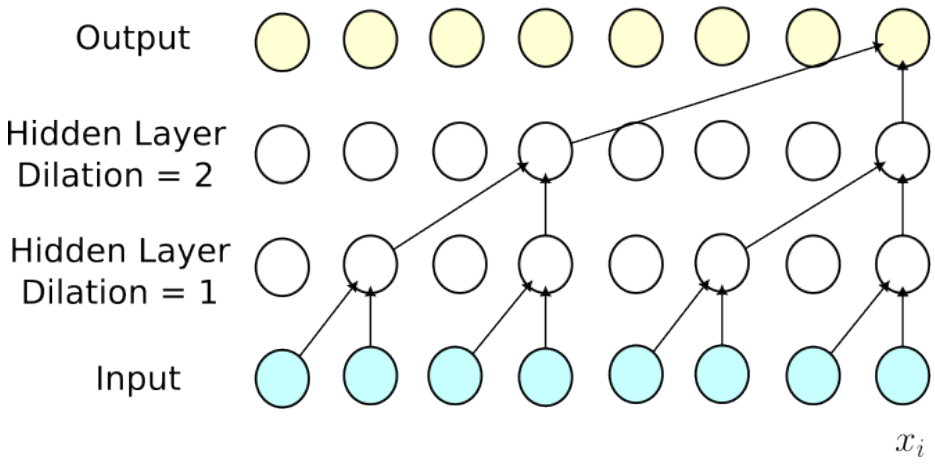
- These convolutions mask the input of the filters so no elements that correspond to the future are used to compute the next step (unlike normal convolutions), this is called causal convolutions
- Inference is also autoregressive, but is faster given that the number of connections is reduced



Masked Temporal (1D) Convolutions



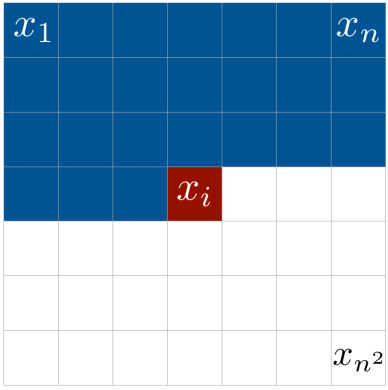
Masked Temporal Dilated (1D) Convolutions



$$p(x_{i+1}|x_{\leq i})$$

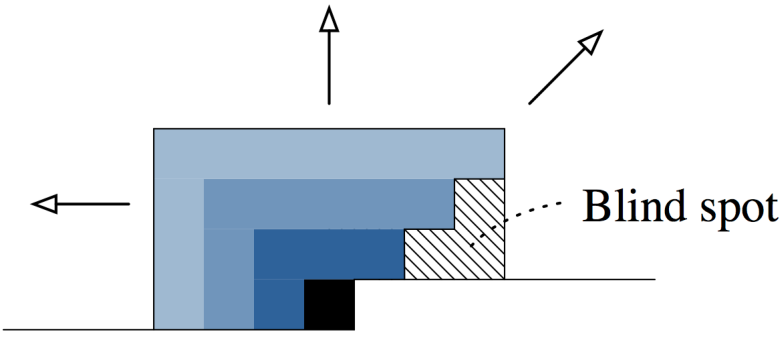
Masked Spatial (2D) Convolutions - PixelCNN

- If we linearise the pixels of a 2D image we lose a lot of information
- We can use 2D convolutions with adequate masking to capture the spatial relationships among pixels maintaining an autoregressive order
- We can use for instance the raster order



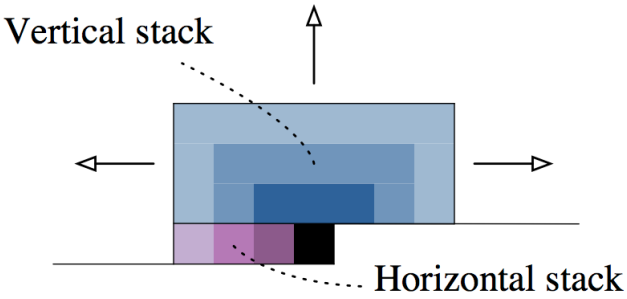
PixelCNN: Receptive field problem

- PixelCNN has a problem with the receptive field of the convolutions
- As we add more layers the upper right side of the image is not used by the convolutions

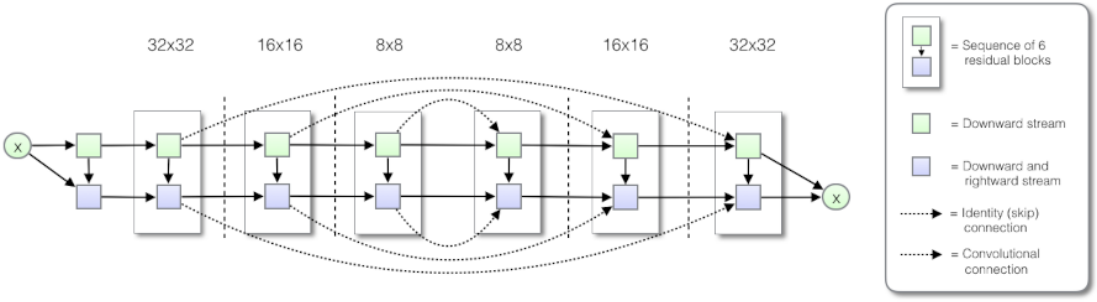


Gated Pixel CNN

- We can eliminate the blind-spot by adding two processing stacks
- A horizontal stack that performs 1-D causal convolutions
- A vertical stack that performs 2-D convolutions that conditions on the rows above the pixel to predict (only part of the receptive field is used)



- Different improvements are added to the architecture apart from the two stream of vertical and horizontal convolutions
  - U-net style skip connections
  - Downsamplig and upsampling

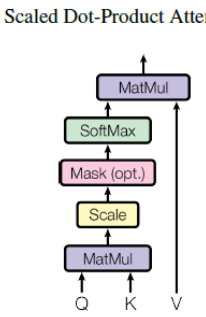


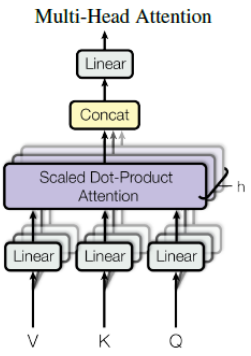
Masked Attention Transformer networks

- Convolutions have the problem of a limited receptive field, that reduces how long are the dependencies that we can capture
- Self Attention provides a better solution
  - It has an unlimited receptive field
  - The number of parameters does not increase with the number of dimensions
  - Computations can be done in parallel

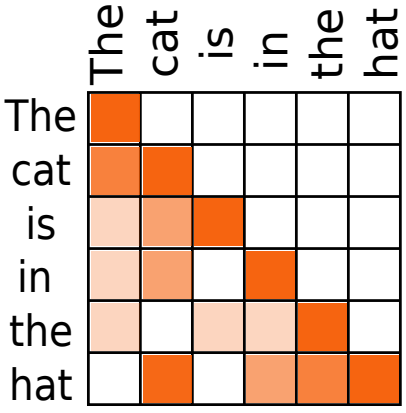
Self Attention

- Attention is a specialized layer developed for NLP
- It allows working with sequences
- Computes for an output sequence what is the influence of the elements from the input sequence for predicting its values



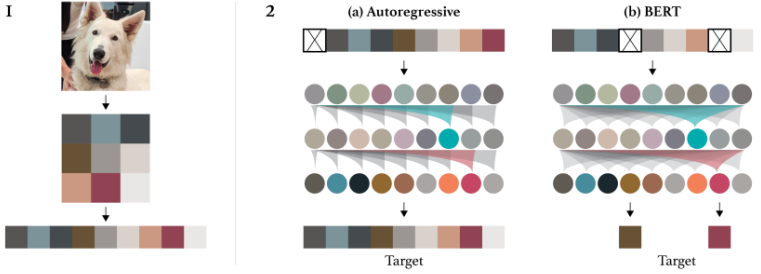


- Basically we compute probabilities from the similarity of elements of two sequences that are then applied as weights for other sequence
- We can apply a mask to decide what elements are used in the combination (e.g. autorregressive order)
- The mask can be in any order (unlike for convolutions)



ImageGPT (Open AI)

- ImageGPT is based on the transformer architecture of GPT2 (attention blocks) that only has a decoder
- Images are transformed to low resolution (tokens) and linearized, this makes the input sequence as in a language model
- The model is trained autoregressively (GPT) or also to predict masked tokens (BERT)



Autoregressive Image generation at scale

- The main issue with autoregressive generation is the length of the sequence
- Modern autoregressive models work with tokens in a latent space
- An additional model (tokenizer) is used to compress the input and the autoregressive model learns from that representation
- Usually the model used form compression is a Variational Autoencoder or a Vector Quantizer Variational autoencoder
- More innovations are introduced to improve image quality and speed as multiscale autoregression, better tokenizers, exploiting locality and parallelizing decoding

This Python Notebook shows examples of autoregressive models

- Autoregressive Models Notebook (click here to open the notebook in colab)

If you download the notebook you will be able to use it locally (run jupyter notebook/visual code to open the notebooks)
This notebook needs Torch for GPU (with CUDA) for a fast execution.

BSC AI Factory

AI Factories

Co-funded by the European Union

Plan de Recuperación, Transformación y Resiliencia

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 101234399 and Spain, Portugal, Romania and Türkiye. The JU receives support from the European Union's Horizon Europe Programme.

BSC AI Factory Partners

Affiliated entities