Autoregressive Models

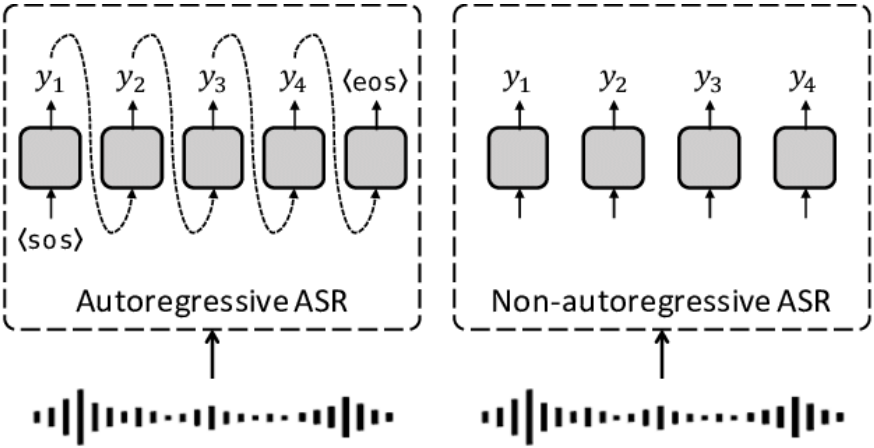
Naeemullah Khan
naeemullah.khan@kaust.edu.sa

KAUST Academy
King Abdullah University of Science and Technology

June 23, 2025

Table of Contents

Table of Contents (cont.)



Illustrations of autoregressive and non-autoregressive ASR.

The Core Problem:
How do we handle really big, complicated data?

- *Example:* A single 128x128 color image has nearly 50,000 numbers (pixels) to describe it!
- Trying to model all these numbers together is way too hard—there are just too many possibilities.
- This is called the "curse of dimensionality": as data gets bigger, the problem gets much harder.

Motivation (cont.)

Desired Properties for our Models:

- **Computational Efficiency:**
  - Efficient training (fast convergence, reasonable resource usage).
  - Efficient model representation (compact size).
- **Statistical Efficiency:**
  - Expressiveness (ability to capture complex data relationships).
  - Generalization (performing well on unseen data from the same distribution).
- **Performance Metrics:**
  - High sampling quality and speed (for generating new data).
  - Good compression rate and speed (for data storage and transmission).

⇒ Learning Outcomes

By the end of this session, you should be able to:

- Understand the basic concept of autoregressive (AR) models.
- Identify classic and modern AR model architectures.
- Know when and how to use AR models in practice.
- Recognize the limitations and challenges in AR models.

Introduction to Autoregressive Models

What are Autoregressive Models?

- **Goal:** To model and generate complex, high-dimensional data distributions.
- **Core Idea:** Factorise the joint distribution of data using the **chain rule**, expressing it as a product of conditional probabilities:

$$p(x_1, x_2, \ldots, x_n) = \prod_{i=1}^{n} p(x_i \mid x_{<i})$$

This allows us to model complex data one element (or "token") at a time, conditioned on previously generated elements.

What are Autoregressive Models? (cont.)

Why not just model everything at once?

- Real-world data (like images or text) has lots of parts working together—modeling all of them at once is super hard.
- Autoregressive models break this big problem into smaller, easier steps.

What can we do with them?

- **Create new data:** Make new images, sounds, or sentences that look real.
- **Compress data:** Store information more efficiently.
- **Spot unusual things:** Find data that doesn't fit the usual pattern.

What are Autoregressive Models? (cont.)

- **Where are Autoregressive Models Used?**
  - **Text:** Making chatbots and writing stories (like how GPT works).
  - **Images:** Drawing pictures one pixel at a time (PixelCNN, PixelRNN).
  - **Audio:** Creating realistic voices and music (WaveNet).

Practical Implementations and Considerations

Classic Approaches:

- **Bayesian Networks:** Use graphs to show how variables depend on each other.
- **Recurrent Neural Networks (RNNs):** Good for handling data that comes in a sequence, like text or time series.

Modern Neural Approaches:

- **Masked Autoencoder for Distribution Estimation (MADE):** A type of neural network that uses special masks to make sure each output only depends on earlier inputs (feed-forward).
- **Causal Masked Neural Models:** Stop the model from "seeing the future" by blocking information from later parts of the sequence.
  - **Convolutional: PixelCNN** and **WaveNet** use masked convolutions to handle images and audio step by step.
  - **Attention-based: Transformers** use masks to focus only on earlier parts of the sequence, making them flexible and powerful.

Design Considerations

Tokenization:

- Break data into small pieces (like words, image patches, or audio samples).
- Makes it easier for models to handle different types of data.

Caching:

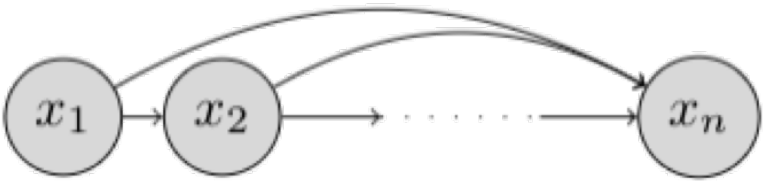
- Save and reuse previous computations.
- Speeds up generation and inference.

Architecture Choices:

- **Decoder-only:** Good for generating sequences from scratch.
- **Encoder-Decoder:** Useful when you want to condition generation on some input.
- New types of recurrent models are still being developed to improve performance.

Key Components and Architectures

- Imagine you want to predict the next value in a sequence, like the next note in a song or the next pixel in an image.
- Autoregressive models help us do this by looking at what has come before.
- They learn patterns by using previous values to guess what comes next.
- These models are widely used for things like time series forecasting and generating new data.



Graphical model for an autoregressive network

- In the MNIST dataset example, we can set an ordering for all the random variables from top-left X_1 to bottom-right X_{784}
- Without loss of generality, we can use chain rule for factorization

$$\mathcal{P}(x_1, \cdots, x_{784}) = \mathcal{P}(x_1)\mathcal{P}(x_2|x_1)\mathcal{P}(x_3|x_1, x_2)\cdots\mathcal{P}(x_{784}|x_1, x_2, \cdots, x_{784})$$

- Parameterizing the above and using the sigmoid function for binarization,
  - $\mathcal{P}_{CPT}(X_1 = 1; \alpha^1) = \alpha^1, \mathcal{P}(X_1 = 0) = 1 - \alpha^1$
  - $\mathcal{P}_{logit}(X_2 = 1|x_1; \alpha^2) = \sigma(\alpha_0^2 + \alpha_1^2 x_1)$
  - $\mathcal{P}_{logit}(X_3 = 1|x_1, x_2; \alpha^3) = \sigma(\alpha_0^3 + \alpha_1^3 x_1 + \alpha_2^3 x_2)$

Autoregressive Models: **Historical Models**

Historical Models

Before models like **Transformers** and **PixelCNN**, some key neural autoregressive models laid the groundwork for using neural networks to estimate data distributions.
Here's a quick overview:

- **FVSBN (Fully Visible Sigmoid Belief Networks):** One of the first models to break down complex data into simpler parts, making it easier to calculate probabilities.
- **NADE (Neural Autoregressive Distribution Estimator):** Used neural networks to model binary data step by step, making the process efficient and scalable.
- **RNADE (Real-valued NADE):** Extended NADE to work with real numbers, so it could handle more types of data.

These early models showed that neural networks could be used for density estimation and inspired many of the advanced models we use today.

Fully Visible Sigmoid Belief Network (FVSBN)

Introduced by: Bengio and Bengio (2000)

Key Idea:

- A Bayesian network defined over observed variables, where each conditional distribution is modeled using a sigmoid (logistic) unit.
- All variables are visible; there are no latent or hidden variables in the model.

Key Features:

- Each variable is conditionally independent of all others given its predecessors.
- The model captures complex dependencies between variables through a factorized representation.
- It allows for efficient computation of the joint probability distribution over the observed variables.

Formulation:

- The joint probability of the vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ is factorized as:

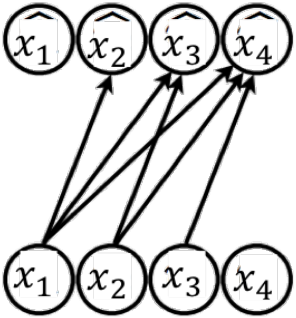
$$p(\mathbf{x}) = \prod_{i=1}^{n} p(x_i \mid x_{<i})$$

where $x_{<i} = (x_1, x_2, \ldots, x_{i-1})$.

- Each conditional is parameterized as:

$$p(x_i = 1 \mid x_{<i}) = \sigma(W_i x_{<i} + b_i)$$

where W_i is a row vector of weights for the i-th variable, b_i is a bias term, and $\sigma(z) = \frac{1}{1+e^{-z}}$ is the sigmoid function.



A fully visible sigmoid belief network over 4 variables

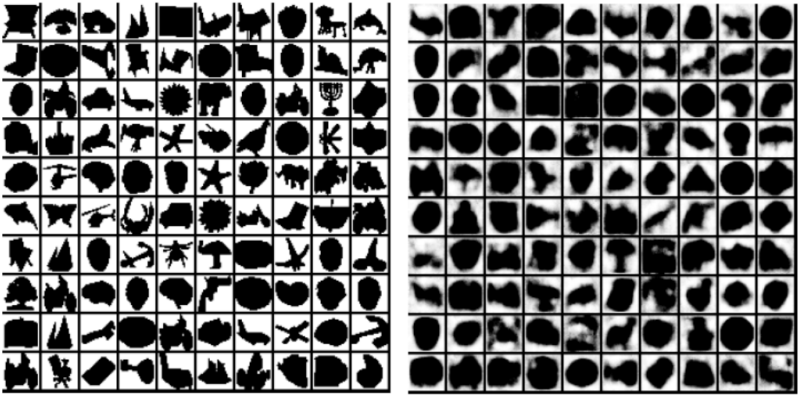
Fully Visible Sigmoid Belief Network (FVSBN) (cont.)

- The conditional variables $X_i|X_1, X_2, \cdots X_{i-1}$ are Bernoulli with parameters

$$\hat{x}_i = \mathcal{P}(X_i = 1 \mid x_1, \cdots, x_{i-1}; \alpha^i) = \mathcal{P}(X_i = 1 \mid x_{<i}; \alpha^i) = \sigma(\alpha_0^i + \sum_{j=1}^{i-1} \alpha_j^i x_j)$$

- To evaluate $\mathcal{P}(x)$, we just multiply all the conditionals.
- To sample new images, we sample each X_i chronologically.
  - Sample $\overline{x}_1 \sim \mathcal{P}(x_1)$ np.random.choice([1,0], p=$[\hat{x}, 1-\hat{x}]$)
  - Sample $\overline{x}_2 \sim \mathcal{P}(x_2|X_1 = \overline{x}_1)$
  - Sample $\overline{x}_3 \sim \mathcal{P}(x_3|X_1 = \overline{x}_1, X_2 = \overline{x}_2)$
- Total number of parameters $= 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$
- **Pros:** The likelihood is tractable and can be computed exactly.
- **Cons:** Poor scalability: each variable x_i requires its own set of weights and bias, leading to a quadratic number of parameters as the number of variables increases.

FVSBN: Results



FVSBN results. Left: Training data. Right: Samples generated by model (Learning Deep Sigmoid Belief Networks with Data Augmentation, 2015)

Introduced by: Larochelle and Murray (2011)

Goal: Provide an efficient and scalable version of Fully Visible Sigmoid Belief Networks (FVSBN) by using shared weights and a feedforward neural network.

Main Idea:

- Uses a masked neural network to estimate each conditional probability:

$$p(\mathbf{x}) = \prod_{i=1}^{n} p(x_i \mid x_{<i})$$

- Efficiently computes gradients via weight sharing and dynamic masking.

Advantages:

- Tractable and scalable.
- Inspired later models such as MADE and normalizing flows.

Model Structure:

- Each variable x_i is modeled conditionally on all previous variables $x_{<i}$.
- Uses a neural network to compute the conditional probabilities.
- The weights are shared across different variables, reducing the number of parameters significantly.

Formulation:

- The conditional probability is modeled as:

$$p(x_i \mid x_{<i}) = \sigma(\alpha^{(i)} h_i + b_i)$$

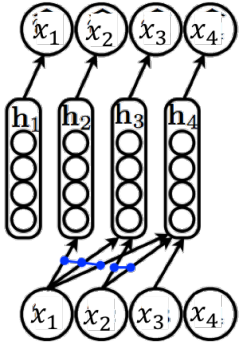
where h_i is a hidden representation computed from the previous variables, $\alpha^{(i)}$ are the weights, and b_i is a bias term.

Neural Network Layer:

NADE uses a neural network layer instead of just logistic regression to improve model performance.

$$h_i = \sigma(\mathbf{W}_{.,<i} x_{<i} + c)$$

$$f_i(x_1, x_2, \cdots, x_{i-1}) = \sigma(\alpha^{(i)} h_i + b_i)$$



NADE

A neural autoregressive density estimator over four variables. Blue connections donate shared weights.

NADE Results

NADE results. Left: Samples generated by model. Right: Conditional Probabilities $\hat{x}_i$ (The Neural Autoregressive Distribution Estimator, 2011)

- What if the output variable is not binary? For example, we have to predict pixel value between 0 and 255.
- One solution: Let $\hat{\mathbf{x}}_i$ parameterize a categorical distribution

$$h_i = \sigma(\mathbf{W}_{.,<i} x_{<i} + c) \tag{1}$$

$$\mathcal{P}(x_i | x_1, \cdots, x_{i-1}) = \text{Cat}(p_i^1, \cdots, p_i^K) \tag{2}$$

$$\hat{\mathbf{x}}_i = (p_i^1, \cdots, p_i^K) = \text{softmax}(A_i h_i + b_i) \tag{3}$$

- Softmax generalizes the sigmoid function $\sigma(\cdot)$ and transforms a vector of K numbers into a vector of K probabilities (non-negative, sum to 1).

- How to model continuous random variables $X_i \in \mathbb{R}$?
- Solution: Let $\hat{\mathbf{x}}_i$ parameterize a continuous distribution.
- This was introduced in RNADE.
- $\hat{\mathbf{x}}_i$ defines the mean and stddev of Gaussian Random Variable (μ_i^j, σ_i^j)

RNADE – Real-valued NADE

Introduced by: Uria et al. (2013)

RNADE (Real-valued Neural Autoregressive Density Estimator) extends NADE to handle continuous data.

- **Extension:** Uses a mixture of Gaussians for each conditional distribution.
- **Conditional Density:**

$$p(x_i \mid x_{<i}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x_i \mid \mu_k, \sigma_k^2)$$

- The parameters of each mixture component (weights π_k, means μ_k, variances σ_k^2) are produced by a neural network conditioned on $x_{<i}$.

RNADE – Real-valued NADE (cont.)

Formulation:

- The joint probability is factorized as:

$$p(\mathbf{x}) = \prod_{i=1}^{n} p(x_i \mid x_{<i})$$

- Each conditional is modeled as:

$$p(x_i \mid x_{<i}) = \sum_{k=1}^{K} \pi_k(x_{<i})\, \mathcal{N}(x_i \mid \mu_k(x_{<i}), \sigma_k^2(x_{<i}))$$

where π_k, μ_k, and σ_k are outputs of a neural network conditioned on $x_{<i}$.

Applications:

- Density estimation for real-valued data (e.g., audio, speech).

RNADE – Real-valued NADE (cont.)

Pros:

- Powerful for modeling complex continuous distributions.
- Influential precursor to more advanced autoregressive flows.

Cons:

- Still suffers from scalability issues with high-dimensional data.
- Requires careful tuning of the number of mixture components.

Learning Autoregressive Models

- The goal of learning is to return a model $\hat{\mathcal{M}}$ that precisely captures the distribution $\mathcal{P}_{data}$ from which our data was sampled
- This is in general not achievable because of
  - limited data only provides a rough approximation of the true underlying distribution
  - computational reasons
- Example. Suppose we represent each MNIST digit with a vector $\mathbf{X}$ of 784 binary variables (black vs. white pixel). How many possible states (= possible images) in the model? $2^{784} \approx 10^{236}$. Even 10^7 training examples provide extremely sparse coverage!
- We want to select $\hat{\mathcal{M}}$ to construct the "best" approximation to the underlying distribution $\mathcal{P}_{data}$

Learning Autoregressive Models (cont.)

- We want to learn the full distribution so that later we can answer any probabilistic inference query
- We want to construct $\mathcal{P}_\theta$ as "close" as possible to $\mathcal{P}_{data}$ (recall we assume we are given a dataset $\mathcal{D}$ of samples from $\mathcal{P}_{data}$)

Learning a generative model. $d(\cdot)$ is a distance measure.

KL Divergence

- How do we evaluate "closeness" between model and data distribution?
- **Kullback-Leibler divergence** (KL-divergence) is one possibility:

$$D(\mathcal{P}_{data}||\mathcal{P}_\theta) = E_{x\sim\mathcal{P}_{data}}\left[log\left(\frac{\mathcal{P}_{data}(x)}{\mathcal{P}_\theta(x)}\right)\right] = \sum_x \mathcal{P}_{data}(x) log \frac{\mathcal{P}_{data}(x)}{\mathcal{P}_\theta(x)}$$

- $D(\mathcal{P}_{data}||\mathcal{P}_\theta)$ iff the two distributions are the same.
- It measures the "compression loss" (in bits) of using $\mathcal{P}_\theta$ instead of $\mathcal{P}_{data}$.

Expected Log-Likelihood

- We can simplify this somewhat:

$$\begin{aligned} D(\mathcal{P}_{data}||\mathcal{P}_\theta) &= E_{x\sim\mathcal{P}_{data}}\left[log\left(\frac{\mathcal{P}_{data}(x)}{\mathcal{P}_\theta(x)}\right)\right] \\ &= E_{x\sim\mathcal{P}_{data}}[log\mathcal{P}_{data}(x)] - E_{x\sim\mathcal{P}_{data}}[log\mathcal{P}_\theta(x)] \end{aligned} \tag{4}$$

- The first term does not depend on $\mathcal{P}_\theta$. Then, *minimizing* KL divergence is equivalent to *maximizing* the **expected log-likelihood**

$$\begin{aligned} \arg\min_{\mathcal{P}_\theta} D(\mathcal{P}_{data}||\mathcal{P}_\theta) &= \arg\min_{\mathcal{P}_\theta} -E_{x\sim\mathcal{P}_{data}}[log\mathcal{P}_\theta(x)] \\ &= \arg\max_{\mathcal{P}_\theta} E_{x\sim\mathcal{P}_{data}}[log\mathcal{P}_\theta(x)] \end{aligned} \tag{5}$$

  - Asks that $\mathcal{P}_\theta$ assign high probability to instances sampled from $\mathcal{P}_{data}$, so as to reflect the true distribution
  - Because of log, samples x where $\mathcal{P}_\theta(x) \approx 0$ weigh heavily in objective

- Although we can now compare models, since we are ignoring $H(\mathcal{P}_{data})$, we don't know how close we are to the optimum.

- Approximate the expected log-likelihood

$$E_{x \sim \mathcal{P}_{data}}[log \mathcal{P}_\theta(x)]$$

with the empirical log-likelihood:

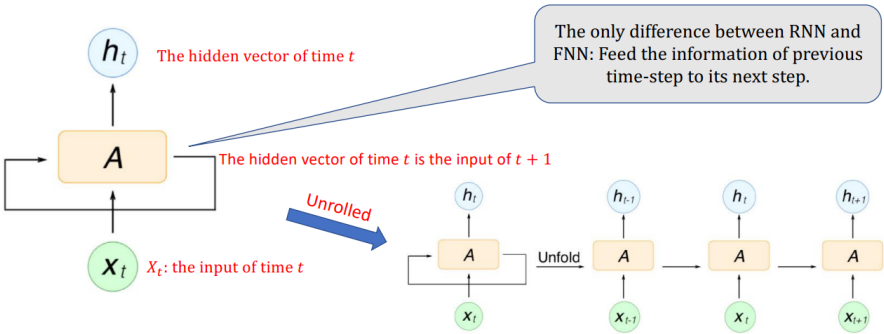
$$E_{\mathcal{D}} = \frac{1}{\mathcal{D}} \sum_{x \in \mathcal{D}} log \mathcal{P}_\theta(x)$$

- **Maximum likelihood learning** is then:

$$\arg\max_{\mathcal{P}_\theta} \frac{1}{\mathcal{D}} \sum_{x \in \mathcal{D}} log \mathcal{P}_\theta(x)$$

Recurrent Neural Networks (RNN)

- The next form of autoregressive models
- **Challenge**: model $\mathcal{P}(x_t|x_{1:t-1};\alpha^t)$. "History" $x_{1:t-1}$ keeps getting longer.
- **Idea**: keep a summary and recursively update it



Recurrent Neural Network (RNN) sample architecture.

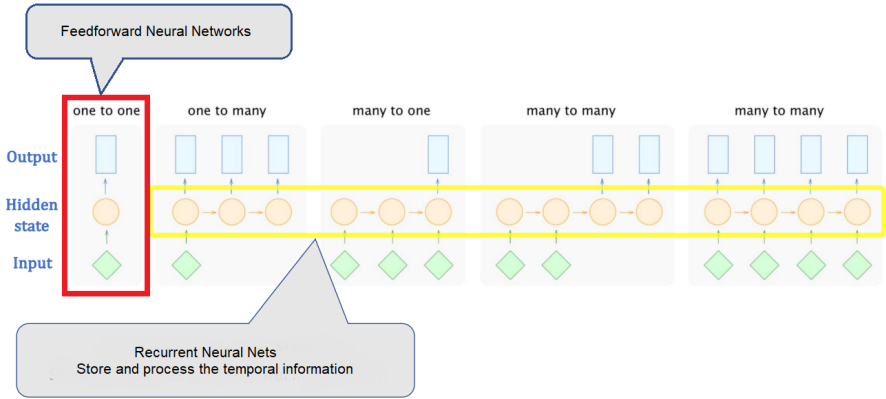
Recurrent Neural Networks (RNN) (cont.)

Pros:

- Can be applied to sequences of arbitrary length.
- Very general: For every computable function, there exists a finite RNN that can compute it

Cons:

- Still requires an ordering
- Sequential likelihood evaluation (very slow for training)
- Sequential generation (unavoidable in an autoregressive model)
- Can be difficult to train (vanishing/exploding gradients)



Different sequential modeling types

Image Captioning

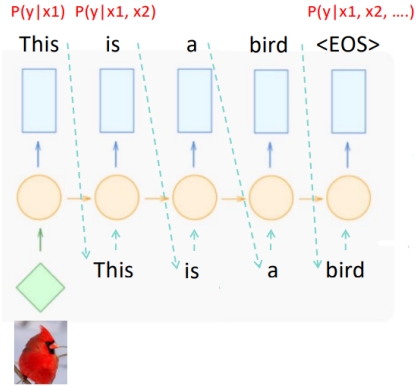
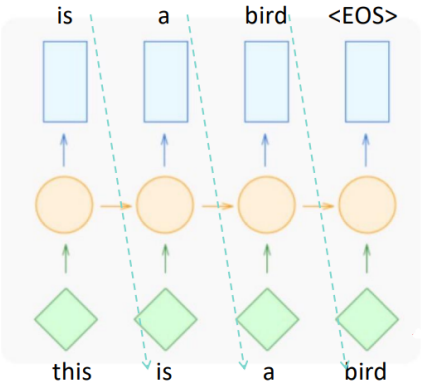


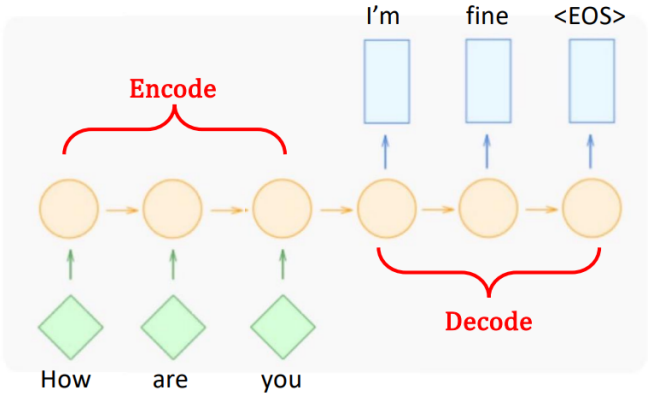
Image captioning with RNNs

Text Generation



Text generation with RNNs

Chatbot



Chatbot with RNNs

Introduced by: van den Oord et al. (2016)

What is PixelRNN?

- PixelRNN is a generative model specifically designed for image generation.
- It models the joint distribution of all pixels in an image by predicting one pixel at a time.
- Pixels are generated in a raster-scan order (left to right, top to bottom).
- Uses an autoregressive approach, where each pixel is generated conditioned on all previously generated pixels.

Core Idea: The model factorizes the image distribution as:

$$p(\mathbf{x}) = \prod_{i=1}^{n} p(x_i \mid x_1, x_2, \ldots, x_{i-1})$$

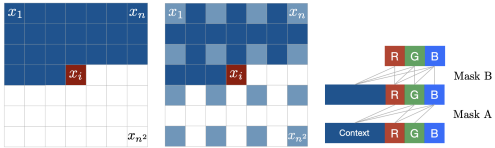
where each pixel x_i is conditioned on all previous pixels $(x_1, x_2, \ldots, x_{i-1})$.

Each pixel's value (such as its RGB channels) is predicted based on all preceding pixels using a recurrent neural network, allowing the model to capture complex dependencies and structures within the image.

Architecture of PixelRNN

1. **Input Representation:**
   - Each image is a 2D grid of pixels.
   - Each pixel can have multiple channels (e.g., RGB).
2. **Autoregressive Modeling:**
   - For each pixel, the model learns $p(x_{i,j} \mid x_{<i,j})$, where $x_{<i,j}$ are all pixels above and to the left of the current pixel (i, j).
3. **RNN Layers:**
   - Uses RNNs along the rows of the image:
     - **Row LSTM:** Processes one row at a time from left to right.
     - **Diagonal BiLSTM:** Processes diagonals in the image to improve context.
4. **Masked Convolutions:**
   - Prevent the model from "seeing the future" pixels.

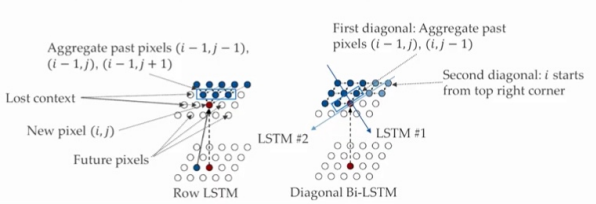
- Each convolution is masked to preserve the autoregressive property.



Left: To generate pixel xi one conditions on all the previously generated pixels left and above of xi. Center: To generate a pixel in the multi-scale case we can also condition on the subsampled image pixels (in light blue). Right: Diagram of the connectivity inside a masked convolution. In the first layer, each of the RGB channels is connected to previous channels and to the context, but is not connected to itself. In subsequent layers, the channels are also connected to themselves.

Types of RNNs in PixelRNN

- **Row LSTM:**
  - Applies a 1D LSTM across each row.
  - Information flows left-to-right in each row.
  - Maintains the autoregressive structure.
- **Diagonal BiLSTM:**
  - Applies a 2D LSTM diagonally across the image.
  - Allows pixels to be conditioned on a broader context.



PixelRNN architecture with Row LSTM and Diagonal BiLSTM. The model processes pixels in a raster-scan order, ensuring that each pixel is conditioned on all previously generated pixels.

Training and Inference:

- **Training:** The model is trained using maximum likelihood estimation (MLE) to maximize the joint probability of the pixels in the training images.
- **Inference:** During inference, pixels are generated sequentially, starting from a blank canvas and conditioning each new pixel on all previously generated pixels.
- **Sampling:** The model can sample new images by repeatedly predicting the next pixel based on the current image context.

Image completions sampled from a PixelRNN.

Pros:

- Capable of generating high-quality images with complex structures.
- Effective at capturing long-range dependencies in images.

Cons:

- Computationally intensive due to sequential processing of pixels.
- Slower inference compared to parallelizable models like PixelCNN.

Autoregressive Models: **Modern Models**

Autoregressive Models: Modern Models

Modern autoregressive models are powerful tools for generating data like images, audio, and text. Let's look at some popular ones and why they matter:

- **PixelCNN/PixelSNAIL:** Great for generating images, using special convolutions to look at pixels one by one.
- **WaveNet:** Designed for audio, it can create realistic speech and music by looking at sound samples in order.
- **Transformer-based Models:** Used in language models like GPT, these use attention to understand and generate sequences (text, images, etc.).

PixelCNN/PixelSNAIL:

- PixelCNN was introduced in 2016 to generate images pixel by pixel.
- It uses masked convolutions so each pixel only depends on the ones before it.
- PixelSNAIL improves on this by adding attention, helping the model see more of the image at once.
- Both are good at making realistic images.

WaveNet:

- WaveNet was made for audio generation, like speech and music.
- It uses dilated convolutions to look at long stretches of audio.
- Each sound sample depends on the previous ones, making the output sound natural.
- WaveNet set new standards for speech synthesis.

Transformer-based Models:

- Transformers were introduced in 2017 for language tasks.
- They use self-attention to look at all parts of a sequence at once.
- This makes them fast and good at handling long texts or sequences.
- Famous models like GPT-3 and BERT use this idea.
- Transformers can also be used for images and audio, not just text.

Autoregressive Models: Modern Models (cont.)

Applications:

- Making new images (PixelCNN, PixelSNAIL)
- Generating speech and music (WaveNet)
- Writing text and language modeling (Transformers)
- Creating videos and working with multiple types of data

Pros:

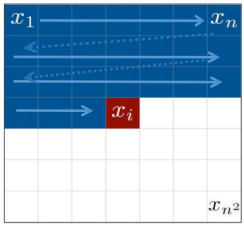
- Can generate high-quality images, audio, and text
- Good at learning complex patterns in data
- Scalable to large datasets and tasks

Cons:

- Training and running these models can be expensive
- Need lots of data to work well
- Still challenging for very large or complex data

Masked Spatial (2D) Convolution - PixelCNN

- Images can be flattened into 1D vectors, but they are fundamentally 2D.
- We can use a masked variant of ConvNet to exploit this knowledge.
- First, we impose an autoregressive ordering on 2D images:



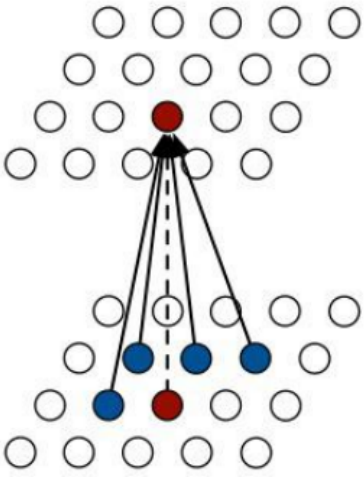
Raster scan ordering of a 2D image. The pixels are processed in a left-to-right, top-to-bottom manner, similar to reading text.

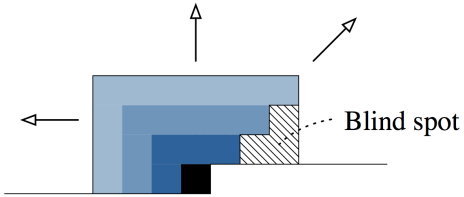
PixelCNN (cont.)

- The autoregressive ordering allows us to use a convolutional neural network (CNN) to predict the next pixel based on the previously generated pixels.
- We use masked convolutions to ensure that the model only has access to the pixels that have already been generated.
- The convolutional layers are designed to respect the autoregressive ordering, meaning that each pixel is predicted based on its left and top neighbours, but not the right or bottom neighbours.

PixelCNN (cont.)

PixelCNNs: Use the neighbour pixels to predict the new pixel.





PixelCNN-style masking has one problem: blind spot in receptive field. The model cannot see the pixels to the right and below the current pixel, which can lead to artifacts in generated images.

Image generation with pixelCNN

PixelCNN (cont.)

- PixelCNN results

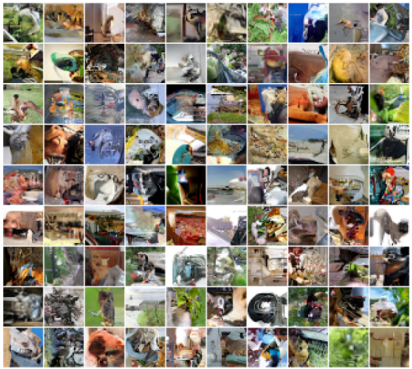
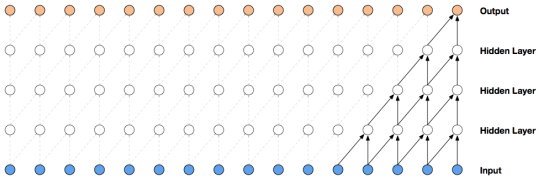


Image generation with pixelCNN. Model trained on Imagenet (32 x 32 pixels)



Visualization of a stack of causal convolutional layers.

[WaveNet – Van den Oord et al, 2016]

- **Causal convolution:** Each output depends only on current and previous inputs.
- Easy to implement: masking part of the convolution kernel.
- Constant parameter count for variable-length distributions.
- Efficient to compute: convolution has hyper-optimized implementations on all hardware.
  However:
  - Limited receptive field, grows linearly with the number of layers.

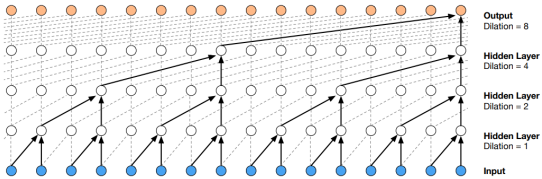
Softmax Output Distribution

- WaveNet models the conditional distribution of the next sample as a categorical distribution:

$$p(x_t|x_1, \ldots, x_{t-1}) = \mathrm{Softmax}(f(x_1, \ldots, x_{t-1}))$$

- The output at each timestep is a probability distribution over possible quantized values.
- Typically, 8-bit μ-law quantization is used for audio.

WaveNet (cont.)



Visualization of a stack of *dilated* causal convolutional layers.

[WaveNet – Van den Oord et al, 2016]

- **Dilated convolution:** Expands receptive field exponentially with depth.
- Dilation factor d:

$$y[t] = \sum_{k=0}^{K-1} w[k] \cdot x[t - d \cdot k]$$

- Enables efficient modeling of long-range dependencies.

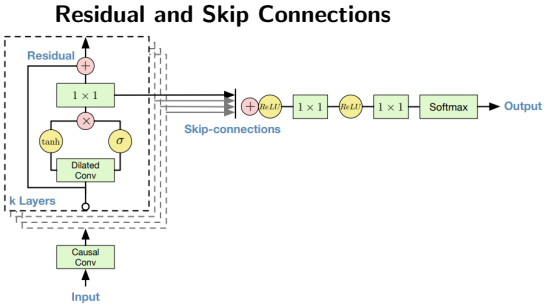
Gated Activation Units

- Each residual block uses gated activation units:

$$z = \tanh(W_f * x) \odot \sigma(W_g * x)$$

- W_f, W_g: convolution filters for filter and gate.
- $\odot$: element-wise multiplication.
- σ: sigmoid activation.
- Improves expressivity and helps gradient flow.

WaveNet (cont.)



Overview of the residual block and the entire architecture.

- **Residual connections:** Add input to output of each block.
- **Skip connections:** Aggregate outputs from all blocks before final output.
- Helps training deep networks and improves convergence.

Conditional WaveNet

- WaveNet can be conditioned on additional information (e.g., speaker identity, linguistic features).
- Conditioning vector h is added to each layer:

$$z = \tanh(W_f * x + V_f * h) \odot \sigma(W_g * x + V_g * h)$$

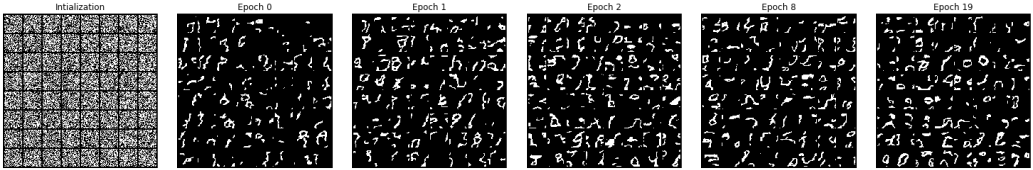
- Enables applications like text-to-speech (TTS) and voice conversion.

Context Stacks

- Multiple stacks of dilated convolutions with different dilation rates.
- Each stack captures dependencies at different temporal scales.
- Improves model's ability to capture both short-term and long-term context.

$$\text{Stack}_i: \quad d \in \{1, 2, 4, \ldots, 2^{n-1}\}$$

WaveNet on MNIST



WaveNet with Pixel Location Appended on MNIST

Append (x,y) coordinates of pixel in the image as input to WaveNet

WaveNet (cont.)

WaveNet is a deep generative model for raw audio waveforms, introduced by DeepMind in 2016. It models the conditional probability of the next audio sample given all previous samples using stacks of causal, dilated convolutional layers.

- **Autoregressive**: Predicts each audio sample sequentially, conditioning on previous samples.
- **Dilated Convolutions**: Enables the model to capture long-range temporal dependencies efficiently.
- **Gated Units, Residual/Skip Connections**: Improve expressivity and trainability.
- **Conditional WaveNet**: Allows conditioning on external features.
- **Applications**: High-quality text-to-speech (TTS), music generation, and audio synthesis.
- **Advantages**: Produces highly realistic and natural-sounding audio compared to previous methods.

Autoregressive Models: **Summary and Limitations**

Summary and Limitations

- Define a specific order for the variables in the data (e.g., $x_1, x_2, \ldots, x_n$).
- Model the joint probability as a product of conditional probabilities:

$$\mathcal{P}(X = \overline{x}) = \prod_{i=1}^{n} \mathcal{P}(x_i \mid x_{<i})$$

- Sampling is performed recursively: generate x_1 first, then x_2 conditioned on x_1, and so on.
- Efficient to compute likelihoods and log-likelihoods for both discrete and continuous variables.
- Flexible: can be extended to multi-class, multi-dimensional, and structured data.

Summary and Limitations (cont.)

- Limitations:
  - No natural way to extract global features or representations.
  - Not inherently suited for clustering or unsupervised learning tasks.
  - The choice of variable ordering can significantly affect performance.

References

References

[1] Bishop, Christopher M. *Pattern Recognition and Machine Learning*. Springer, 2006.

[2] Larochelle, Hugo and Murray, Iain. "Neural autoregressive distribution estimation." In *AISTATS*, 2011, pp. 29–37.

[3] Uria, Benigno, Murray, Iain, and Larochelle, Hugo. "Neural Autoregressive Distribution Estimation." *JMLR*, 17(205):1–37, 2016.

[4] Uria, Benigno, Murray, Iain, and Larochelle, Hugo. "RNADE: The real-valued neural autoregressive density-estimator." In *NeurIPS*, 2013, pp. 2175–2183.

[5] van den Oord, Aaron, Kalchbrenner, Nal, and Kavukcuoglu, Koray. "Pixel Recurrent Neural Networks." In *ICML*, 2016, pp. 1747–1756.

References (cont.)

[6] van den Oord, Aaron, Kalchbrenner, Nal, and Kavukcuoglu, Koray. "Conditional Image Generation with PixelCNN Decoders." In *NeurIPS*, 2016, pp. 4790–4798.

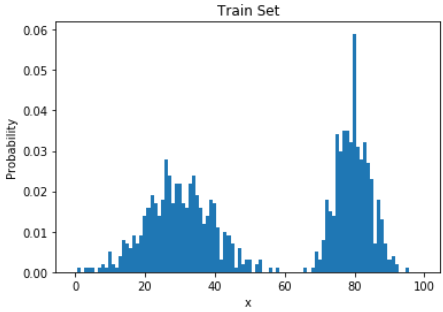
[7] van den Oord, Aaron, Dieleman, Sander, Zen, Heiga, Simonyan, Karen, Vinyals, Oriol, Graves, Alex, Kalchbrenner, Nal, Senior, Andrew, and Kavukcuoglu, Koray. "WaveNet: A Generative Model for Raw Audio." *arXiv preprint arXiv:1609.03499*, 2016.

[8] Salimans, Tim, Karpathy, Andrej, Chen, Xi, and Kingma, Diederik P. "PixelCNN++: Improving the PixelCNN with Discretized Logistic Mixture Likelihood and Other Modifications." *arXiv preprint arXiv:1701.05517*, 2017.

Autoregressive Models: **Appendix**

Learning: Estimate frequencies by counting

- Recall: the goal is to estimate p_{data} from samples $x^{(1)}, \ldots, x^{(n)} \sim p_{\text{data}}(x)$
- Suppose the samples take on values in a finite set $\{1, \ldots, k\}$
- The model: a histogram
  - (Redundantly) described by k nonnegative numbers: $p_1, \ldots, p_k$
- To train this model: count frequencies
- $p_i = \frac{\#\text{ times } i \text{ appears in the dataset}}{\#\text{ points in the dataset}}$



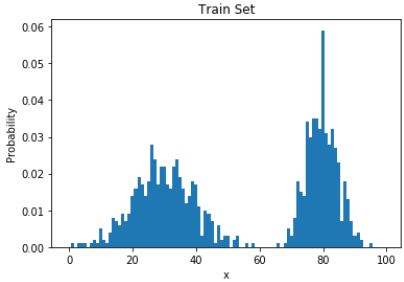
Inference and Sampling

- **Inference (querying p_i for arbitrary i):** simply a lookup into the array $p_1, \ldots, p_k$
- **Sampling (lookup into the inverse cumulative distribution function):**
  - From the model probabilities $p_1, \ldots, p_k$, compute the cumulative distribution:

$$F_i = p_1 + \cdots + p_i \quad \text{for all } i \in \{1, \ldots, k\}$$

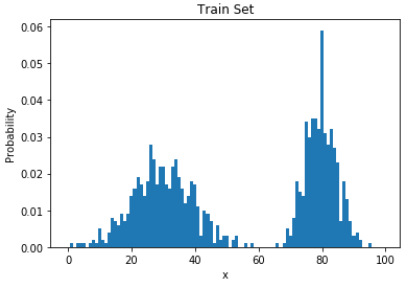
  - Draw a uniform random number $u \sim [0, 1]$
  - Return the smallest i such that $u \leq F_i$
- **Are we done?**

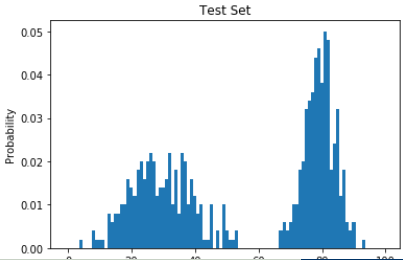
Problem: Lack of Generalization

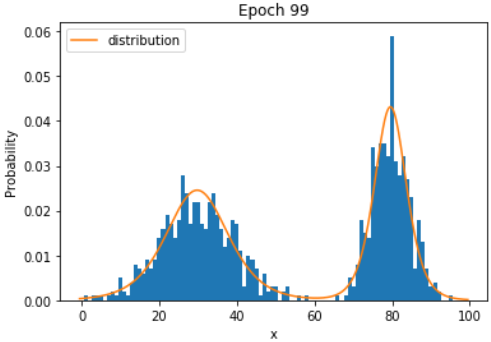


learned histogram = training data distribution
→ often poor generalization

Solution: Parameterized Distributions







Fitting a parameterized distribution often generalizes better than a histogram.

Credits

Dr. Prashant Aparajeya

Computer Vision Scientist — Director(AISimply Ltd)

p.aparajeya@aisimply.uk

This project benefited from external collaboration, and we acknowledge their contribution with gratitude.