# AutoRegressive Models

2019
Multicampus

ModuLabs Research Scientist
**Il Gu Yi**
**Soochul Park**

V.1.1

# Taxonomy of Generative Models

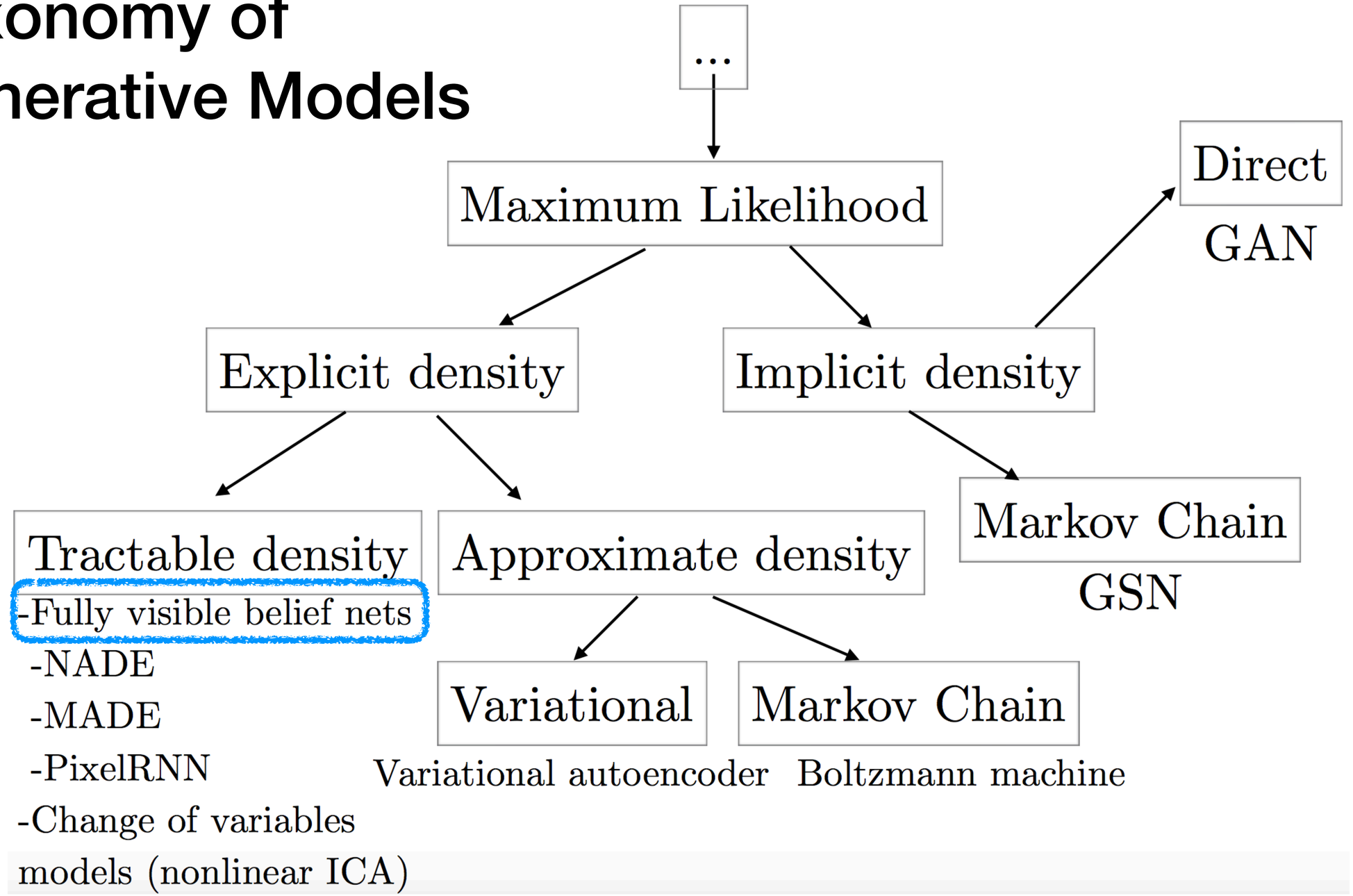
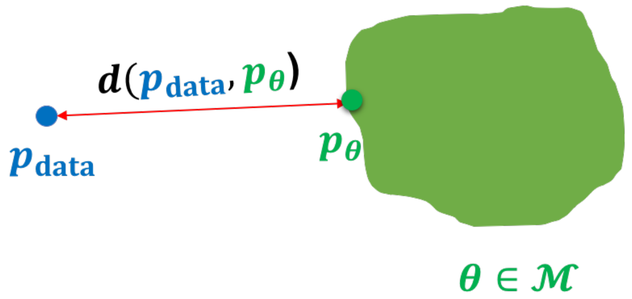



Figure credit: Ian Goodfellow, NIPS 2016 Tutorial: Generative Adversarial Networks

# Learning a generative model

- We are given a training set of examples, e.g., images of dogs

$\mathbf{x}^{(j)} \sim p_{\text{data}}$
$j = 1, 2, \ldots, |\mathcal{D}|$




- We want to learn a probability distribution $p(x)$ over images $x$ such that
  1. **Generation:** If we sample $x_{new} \sim p(x)$, $x_{new}$ should look like a dog (*sampling*)
  2. **Density estimation:** $p(x)$ should be high if $x$ looks like a dog, and low otherwise (*anomaly detection*)
  3. **Unsupervised representation learning:** We should be able to learn what these images have in common, e.g., ears, tail, etc. (*features*)
- First question: how to represent $p(x)$. Second question: how to learn it.

# Recap: Bayesian networks vs neural models

- Using Chain Rule

$$p(x_1, x_2, x_3, x_4) = p(x_1)p(x_2 \mid x_1)p(x_3 \mid x_1, x_2)p(x_4 \mid x_1, x_2, x_3)$$

  Fully General, no assumptions needed (exponential size, no free lunch)

- Bayes Net

$$p(x_1, x_2, x_3, x_4) \approx p_{\text{CPT}}(x_1)p_{\text{CPT}}(x_2 \mid x_1)p_{\text{CPT}}(x_3 \mid \cancel{x_1}, x_2)p_{\text{CPT}}(x_4 \mid x_1, \cancel{x_2, x_3})$$

  Assumes conditional independencies; tabular representations via conditional probability tables (CPT)

- Neural Models

$$p(x_1, x_2, x_3, x_4) \approx p(x_1)p(x_2 \mid x_1)p_{\text{Neural}}(x_3 \mid x_1, x_2)p_{\text{Neural}}(x_4 \mid x_1, x_2, x_3)$$

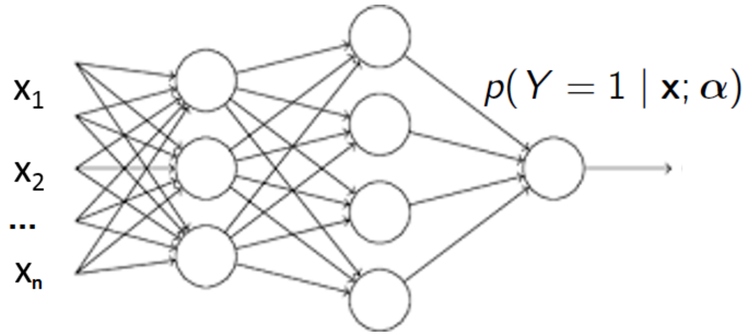
  Assumes specific functional form for the conditionals. A sufficiently deep neural net can approximate any function.

# Neural Models for classification

- Setting: binary classification of $Y \in \{0, 1\}$ given inputs $X \in \{0, 1\}^n$
- For classification, we care about $p(Y \mid \mathbf{x})$, and assume that

$$p(Y = 1 \mid \mathbf{x}; \boldsymbol{\alpha}) = f(\mathbf{x}, \boldsymbol{\alpha})$$

- **Logistic regression**: let $z(\boldsymbol{\alpha}, \mathbf{x}) = \alpha_0 + \sum_{i=1}^{n} \alpha_i x_i$.
  $p_{\text{logit}}(Y = 1 \mid \mathbf{x}; \boldsymbol{\alpha}) = \sigma(z(\boldsymbol{\alpha}, \mathbf{x}))$, where $\sigma(z) = 1/(1 + e^{-z})$
- **Non-linear** dependence: let $\mathbf{h}(A, \mathbf{b}, \mathbf{x}) = f(A\mathbf{x} + \mathbf{b})$ be a non-linear transformation of the inputs (*features*).
  $p_{\text{Neural}}(Y = 1 \mid \mathbf{x}; \boldsymbol{\alpha}, A, \mathbf{b}) = \sigma(\alpha_0 + \sum_{i=1}^{h} \alpha_i \mathbf{h}_i)$
  - More flexible
  - More parameters: $A, \mathbf{b}, \boldsymbol{\alpha}$
  - Repeat multiple times to get a multilayer perceptron (neural network)



# Motivating Example: MNIST

- Suppose we have a dataset $\mathcal{D}$ of handwritten digits (binarized MNIST)

- Each image has $n = 28 \times 28 = 784$ pixels. Each pixel can either be black (0) or white (1).
- We want to learn a probability distribution $p(v) = p(v_1, \cdots, v_{784})$ over $v \in \{0, 1\}^{784}$ such that when $v \sim p(v)$, $v$ looks like a digit
- Idea: define a model family $\{p_\theta(v), \theta \in \Theta\}$, then pick a good one based on training data $\mathcal{D}$ (more on that later)
- How to parameterize $p_\theta(v)$?

# Fully Visible Sigmoid Belief Network

# Fully Visible Sigmoid Belief Network

- We can pick an ordering, i.e., order variables (pixels) from top-left ($X_1$) to bottom-right ($X_{n=784}$)
- Use chain rule factorization without loss of generality:

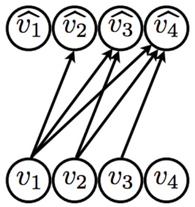
$$p(v_1, \cdots, v_{784}) = p(v_1)p(v_2 \mid v_1)p(v_3 \mid v_1, v_2) \cdots p(v_n \mid v_1, \cdots, v_{n-1})$$

- Some conditionals are too complex to be stored in tabular form. So we **assume**

$$p(v_1, \cdots, v_{784}) = p_{\text{CPT}}(v_1; \alpha^1)p_{\text{logit}}(v_2 \mid v_1; \boldsymbol{\alpha}^2)p_{\text{logit}}(v_3 \mid v_1, v_2; \boldsymbol{\alpha}^3) \cdots p_{\text{logit}}(v_n \mid v_1, \cdots, v_{n-1}; \boldsymbol{\alpha}^n)$$

- More explicitly:
  - $p_{\text{CPT}}(V_1 = 1; \alpha^1) = \alpha^1$, $p(V_1 = 0) = 1 - \alpha^1$
  - $p_{\text{logit}}(V_2 = 1 \mid v_1; \boldsymbol{\alpha}^2) = \sigma(\boldsymbol{\alpha}_0^2 + \boldsymbol{\alpha}_1^2 v_1)$
  - $p_{\text{logit}}(V_3 = 1 \mid v_1, v_2; \boldsymbol{\alpha}^3) = \sigma(\boldsymbol{\alpha}_0^3 + \boldsymbol{\alpha}_1^3 v_1 + \boldsymbol{\alpha}_2^3 v_2)$
- Note: This is a **modeling assumption**. We are using a logistic regression to predict next pixel based on the previous ones. Called **autoregressive**.

# Fully Visible Sigmoid Belief Network



FVSBN

- The conditional variables $V_i \mid V_1, \cdots, V_{i-1}$ are Bernoulli with parameters

$$\hat{v}_i = p(V_i = 1 | v_1, \cdots, v_{i-1}; \boldsymbol{\alpha}^i) = p(V_i = 1 | v_{<i}; \boldsymbol{\alpha}^i) = \sigma(\alpha_0^i + \sum_{j=1}^{i-1} \alpha_j^i v_j)$$

- How to evaluate $p(v_1, \cdots, v_{784})$? Multiply all the conditionals (factors)
  - In the above example:

$$p(V_1 = 0, V_2 = 1, V_3 = 1, V_4 = 0) = (1 - \hat{v}_1) \times \hat{v}_2 \times \hat{v}_3 \times (1 - \hat{v}_4)$$
$$= (1 - \hat{v}_1) \times \hat{v}_2(V_1 = 0) \times \hat{v}_3(V_1 = 0, V_2 = 1) \times (1 - \hat{v}_4(V_1 = 0, V_2 = 1, V_3 = 1))$$

- How to sample from $p(v_1, \cdots, v_{784})$?
  1. Sample $\overline{v}_1 \sim p(v_1)$ (`np.random.choice([1,0],p=`$[\hat{v}_1, 1 - \hat{v}_1]$`)`)
  2. Sample $\overline{v}_2 \sim p(v_2 \mid v_1 = \overline{v}_1)$
  3. Sample $\overline{v}_3 \sim p(v_3 \mid v_1 = \overline{v}_1, v_2 = \overline{v}_2) \cdots$
- How many parameters? $1 + 2 + 3 \cdots + n \approx n^2/2$

# FVSBN Results

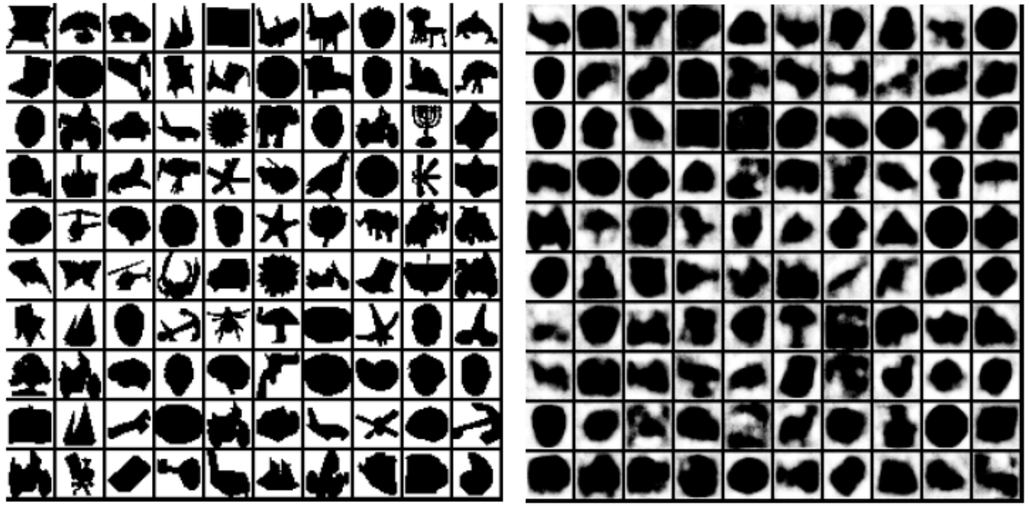


Figure from *Learning Deep Sigmoid Belief Networks with Data Augmentation, 2015*. Training data on the left (*Caltech 101 Silhouettes*). Samples from the model on the right. Best performing model they tested on this dataset in 2015 (more on evaluation later).

# Fully Observable Models (AutoRegressive Generative Models)

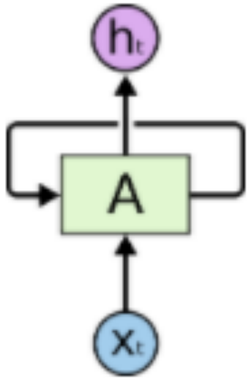
- Explicit formula based on chain rule

$$p_\theta(\mathrm{x}) = p_\theta(x_1) \prod_{i=2}^{n} p_\theta(x_i \mid x_1, \cdots, x_{i-1})$$

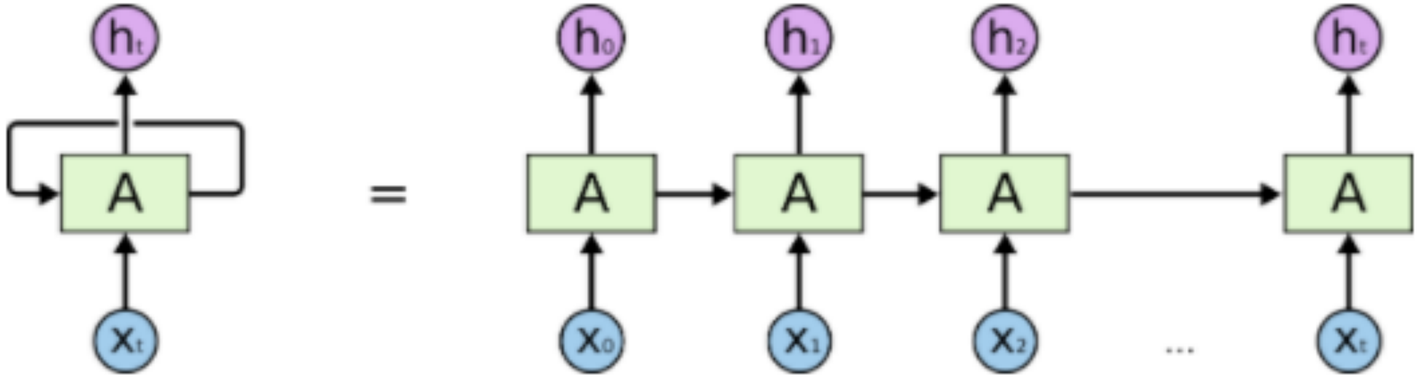
- Examples
  - PixelRNN, PixelCNN
  - WaveNets

# RNN & LSTM

# RNN



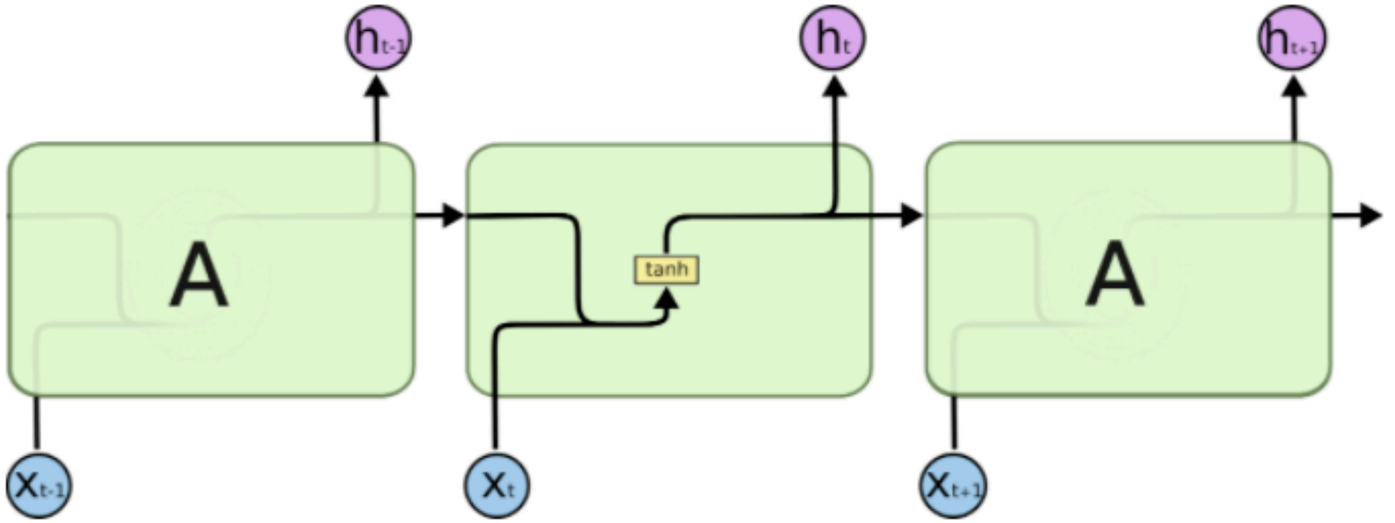

Recurrent Neural Networks have loops.




An unrolled recurrent neural network.

Understanding LSTM Networks
https://colah.github.io/posts/2015-08-Understanding-LSTMs

# RNN



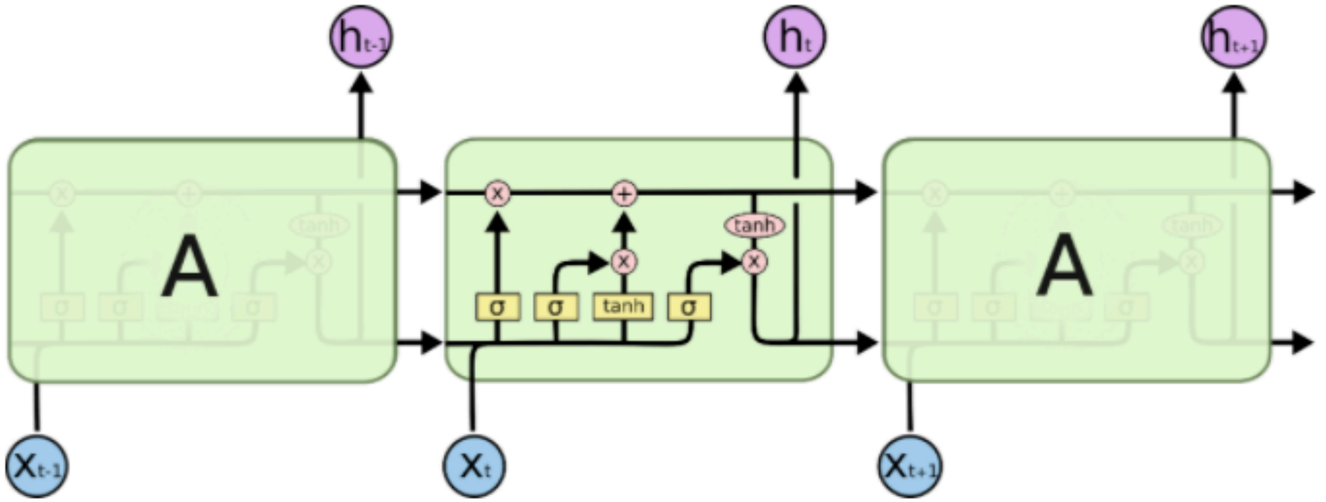

The repeating module in a standard RNN contains a single layer.

$$h_t = \sigma_h(W_h x_t + U_h h_{t-1} + b_h)$$
$$y_t = \sigma_y(W_y h_t + b_y)$$

- $x_t$: input vector
- $h_t$: hidden layer vector
- $y_t$: output vector
- $W$, $U$ and $b$: parameter matrices and vector
- $\sigma_h$ and $\sigma_y$: Activation functions

Understanding LSTM Networks
https://colah.github.io/posts/2015-08-Understanding-LSTMs

# LSTM




**The repeating module in an LSTM contains four interacting layers.**

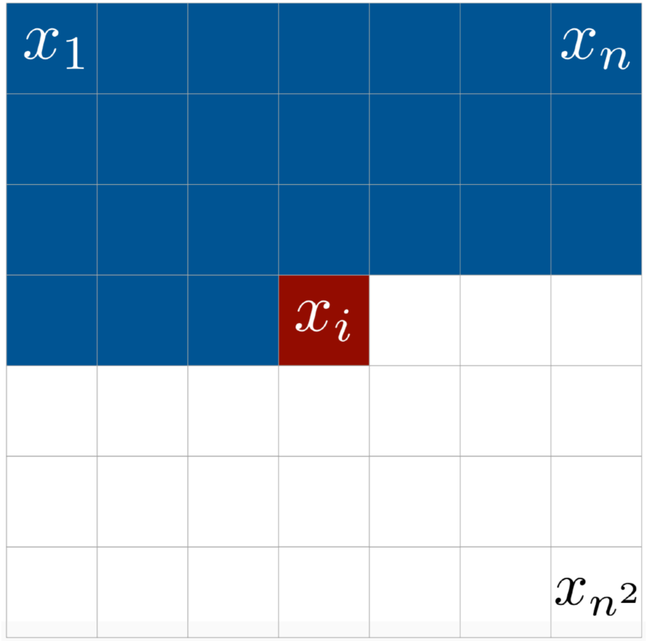
$$
\begin{aligned}
f_t &= \sigma_g(W_f x_t + U_f h_{t-1} + b_f) \\
i_t &= \sigma_g(W_i x_t + U_i h_{t-1} + b_i) \\
o_t &= \sigma_g(W_o x_t + U_o h_{t-1} + b_o) \\
\tilde{c}_t &= \sigma_c(W_c x_t + U_c h_{t-1} + b_c) \\
c_t &= f_t \circ c_{t-1} + i_t \circ \tilde{c}_t \\
h_t &= o_t \circ \sigma_h(c_t)
\end{aligned}
$$

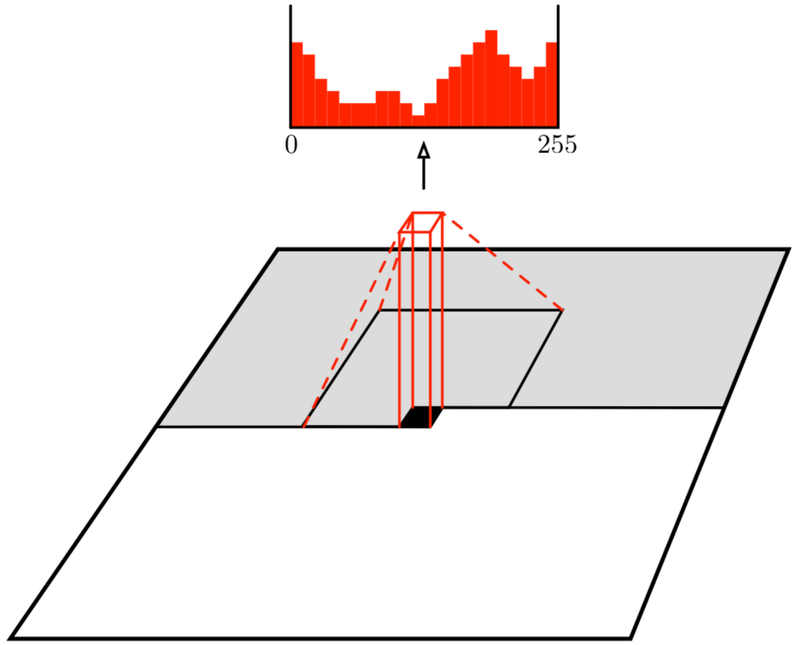
- $x_t \in \mathbb{R}^d$: input vector to the LSTM unit
- $f_t \in \mathbb{R}^h$: forget gate's activation vector
- $i_t \in \mathbb{R}^h$: input/update gate's activation vector
- $o_t \in \mathbb{R}^h$: output gate's activation vector
- $h_t \in \mathbb{R}^h$: hidden state vector also known as output vector of the LSTM unit
- $\tilde{c}_t \in \mathbb{R}^h$: cell input activation vector
- $c_t \in \mathbb{R}^h$: cell state vector
- $W \in \mathbb{R}^{h \times d}$, $U \in \mathbb{R}^{h \times h}$ and $b \in \mathbb{R}^h$: weight matrices and bias vector parameters which need to be learned during training

Understanding LSTM Networks
https://colah.github.io/posts/2015-08-Understanding-LSTMs

# PixelCNN

# PixelCNN




| | | | | |
|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 |

Figure credit: Aäron van den Oord, et al., Pixel Recurrent Neural Networks
Figure credit: Aäron van den Oord, et al., Conditional Image Generation with PixelCNN Decoders

# Results via PixelCNN

Sandbar

Sorrel horse

Lhasa Apso (dog)

Lawn mower

Figure credit: Aäron van den Oord, et al., Conditional Image Generation with PixelCNN Decoders

# PixelRNN

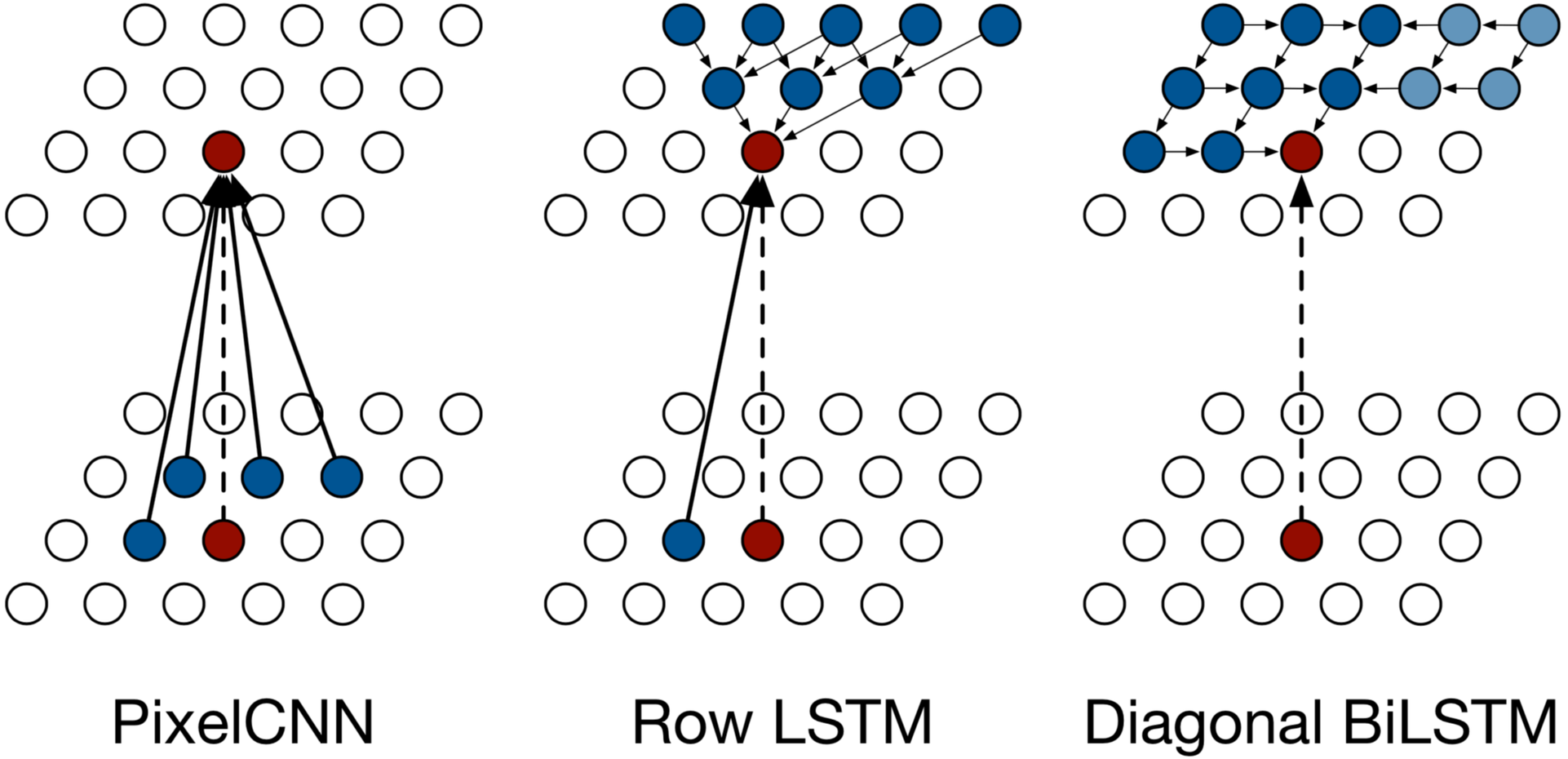
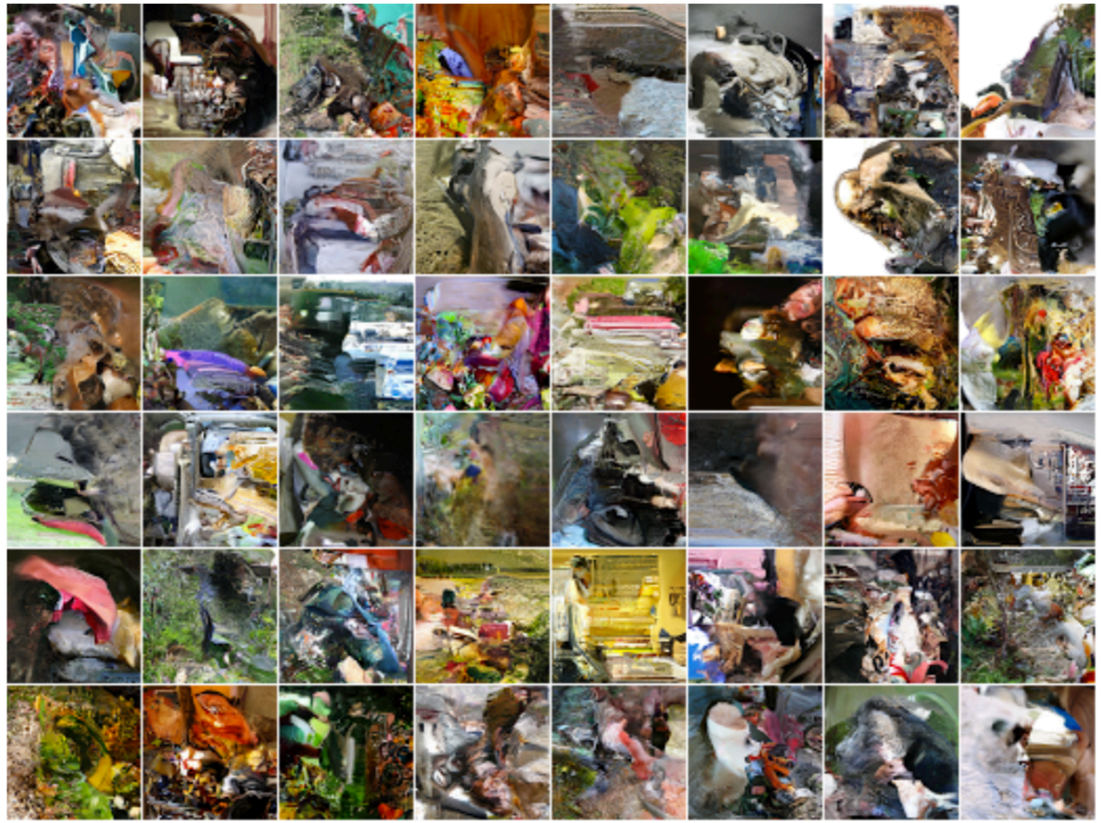



Figure credit: Aäron van den Oord, et al., Pixel Recurrent Neural Networks

# Results via PixelRNN



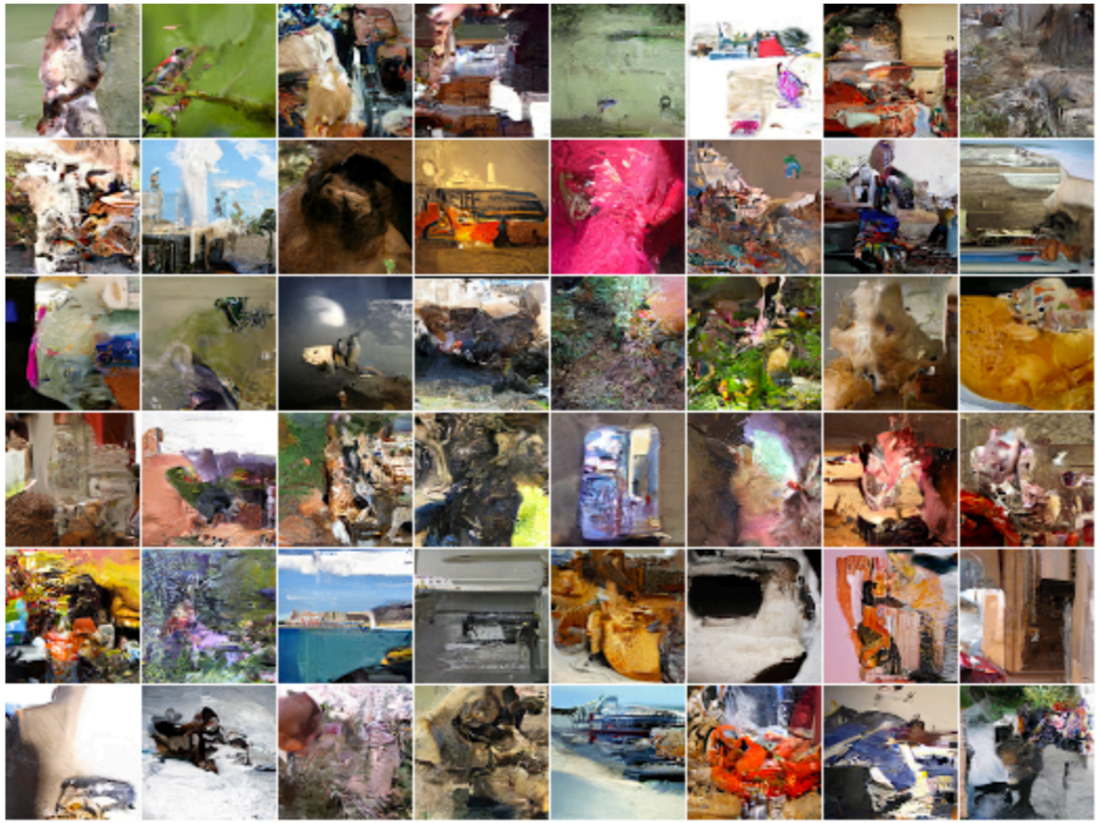
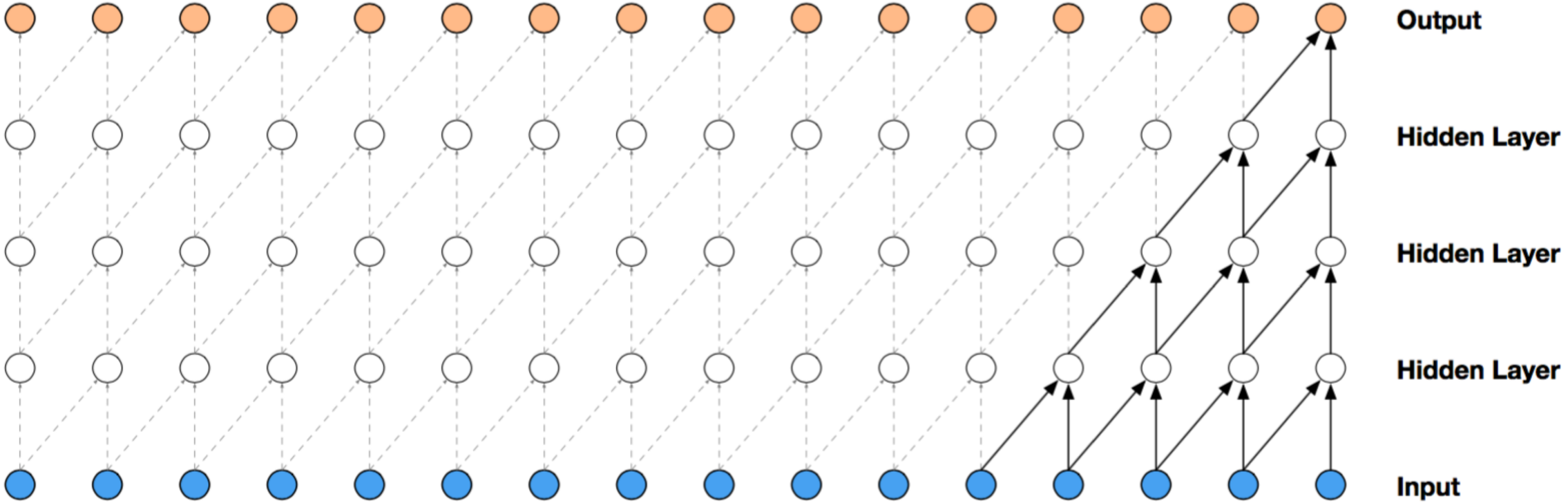


Figure credit: Aäron van den Oord, et al., Pixel Recurrent Neural Networks

# Wavenet

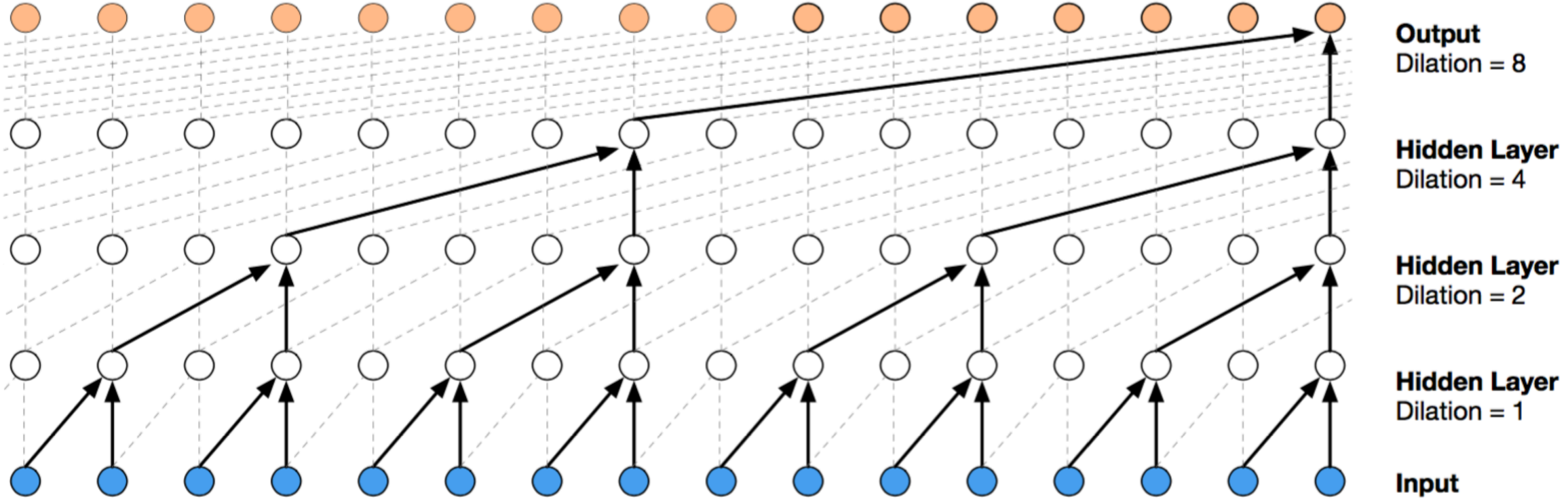
# WaveNet (Naive Causal Convolution)




$$p_\theta(\mathrm{x}) = \prod_{t=1}^{T} p_\theta(x_t \mid x_1, \cdots, x_{t-1})$$

Figure credit: Aäron van den Oord, et al., Wavenet: A Generative Model for Raw Audio

# WaveNet




$$p_\theta(\mathrm{x}) = \prod_{t=1}^{T} p_\theta(x_t \mid x_1, \cdots, x_{t-1})$$

Demo: https://www.deepmind.com/blog/wavenet-generative-model-raw-audio

Figure credit: Aäron van den Oord, et al., Wavenet: A Generative Model for Raw Audio

# WaveNet

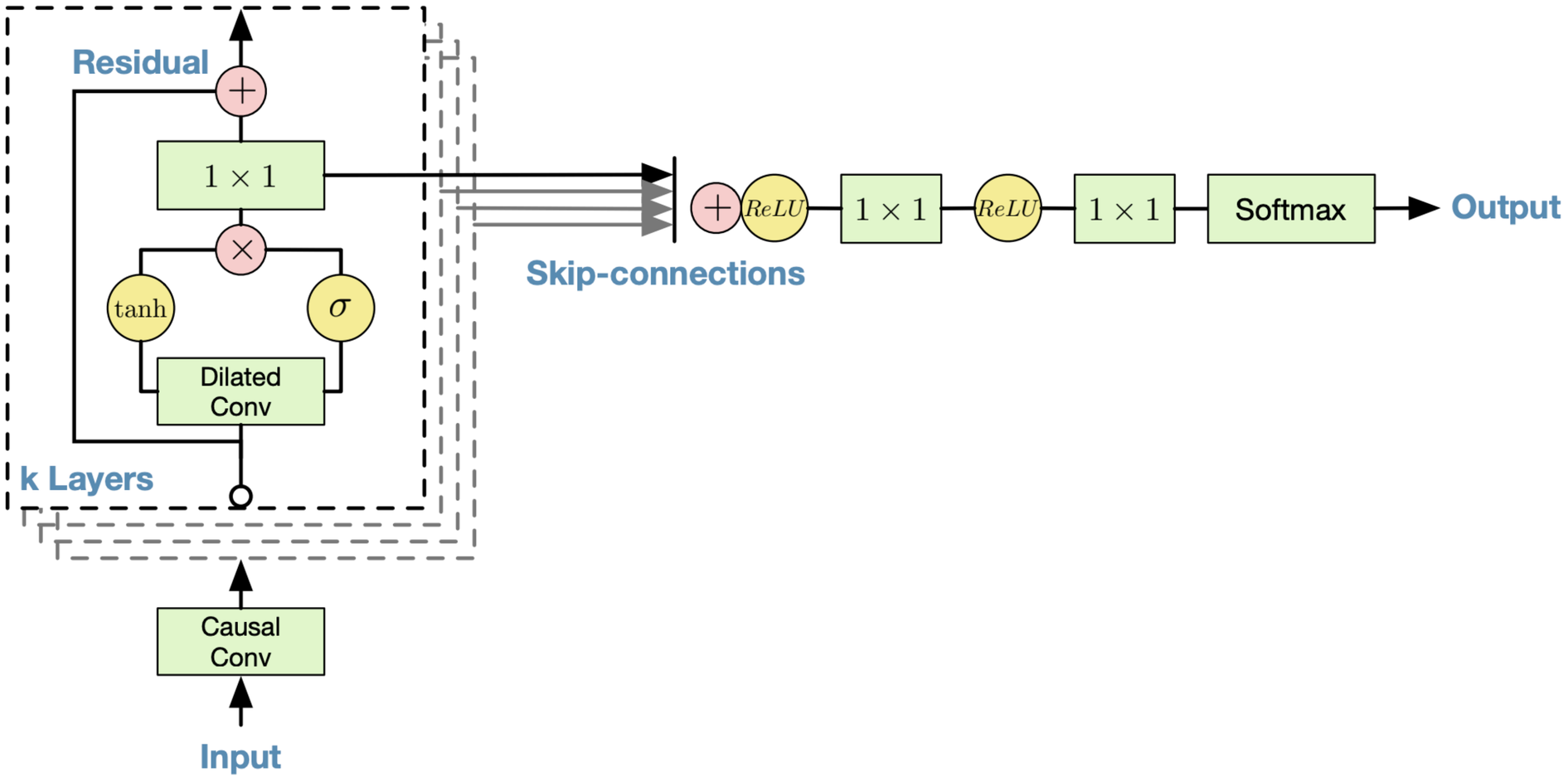



Figure credit: Aäron van den Oord, et al., Wavenet: A Generative Model for Raw Audio

# Generation of Wave

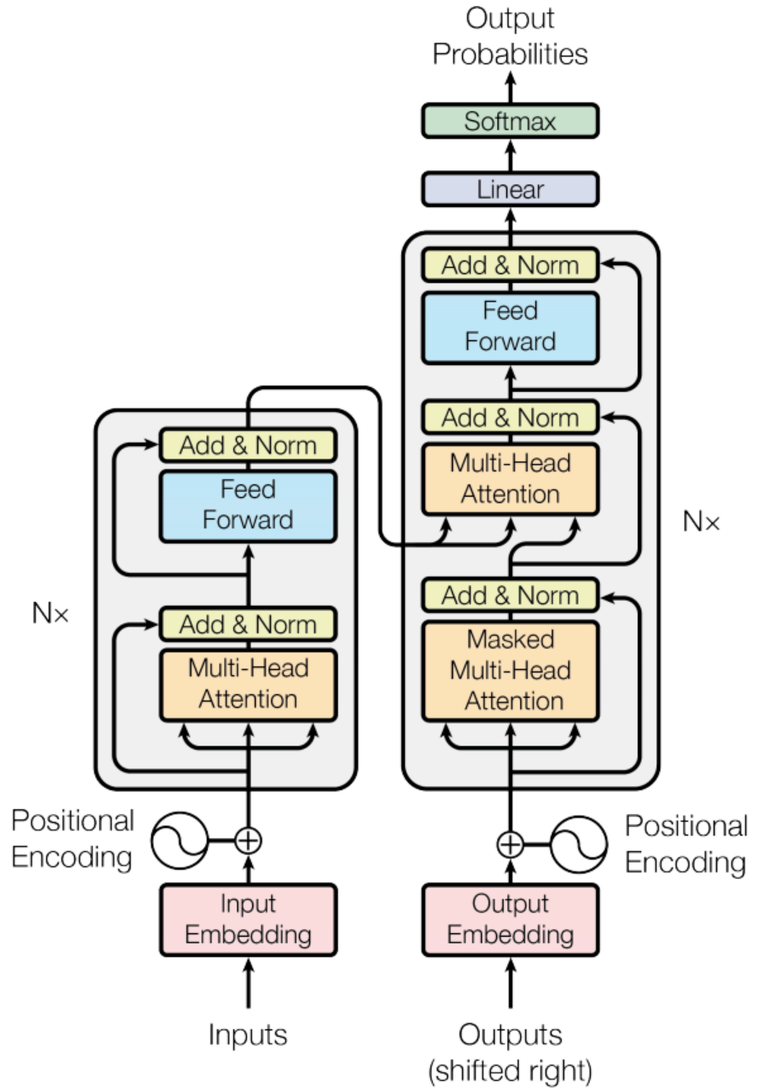
- Raw audio generation
- Neural autoregressive generative models that model complex distributions (ex. images and text)
- PixelRNN: thousands of random variables (e.g. 64x64)
- WaveNet: very high temporal resolution (16,000 samples)
- Based on the PixelCNN

# Weakness of Fully Visible Models

- It takes a long time to inference
  - ex) WaveNet: make a 1sec wave file; takes 2 minutes
- It cannot control the latent code (space)

# Transformer

# Transformer




Transformer는 크게 Inputs 데이터를 인코딩하는 Encoder와, 인코딩된 데이터를 기반으로 Outputs을 디코딩하여 Outputs의 확률분포 파라메터를 출력하는 Decoder로 구성된다.

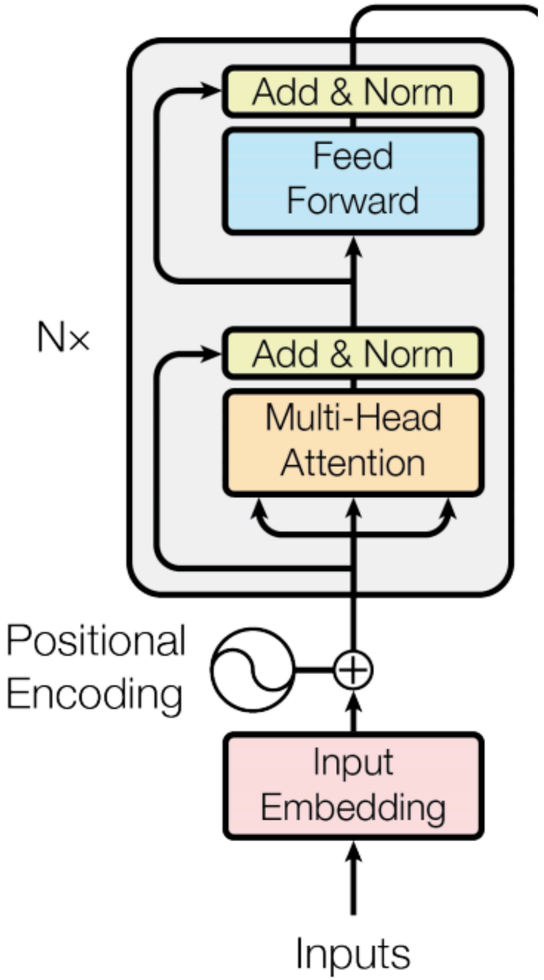
이러한 구조는 seq2seq 모델이라고 불리며 원래 언어의 문장을 대상 언어의 문장으로 변환시키는 번역 작업이나, 질의 문장을 토대로 응답 문장을 생성해내는 질의 응답 작업 등에 사용되었다.

Encoder는 원래 언어의 문장의 토큰을 입력으로 받아 임베딩 테이블을 이용해 벡터로 변환하고 여러 인코딩 블록을 거쳐 최종적으로 Decoder에 들어갈 형태로 출력한다.

Decoder도 마찬가지로 대상 언어의 문장의 토큰을 입력으로 받아 임베딩 테이블을 이용해 벡터로 변환하고 여러 디코딩 블록을 거쳐 다음 토큰에 해당하는 확률 분포의 파라메터를 출력한다.

이 때, 디코딩 블록 내에서 어텐션 메카니즘을 통해 Encoder에서 인코딩된 정보를 가져온다. 어텐션 시 여러 헤드로 분리하여 각각 다양한 부분의 정보를 가져올 수 있도록 하며, 가져온 정보들은 concatenation하여 다음 레이어로 전해진다.

# Transformer - Encoder




Encoder는 Inputs, Input Embedding, Positional Encoding, Attention Blocks으로 구성되며, Attention Blocks내부는 Multi-Head Attention, Feed forward와 Layer Normalization 으로 구성된다.

**Inputs** : 번역 작업의 경우 원본 문장, 질의 응답 작업의 경우 질의에 해당하는 입력 문자열에 해당한다. 문자열에 대응하는 토큰이 입력으로 주어진다.

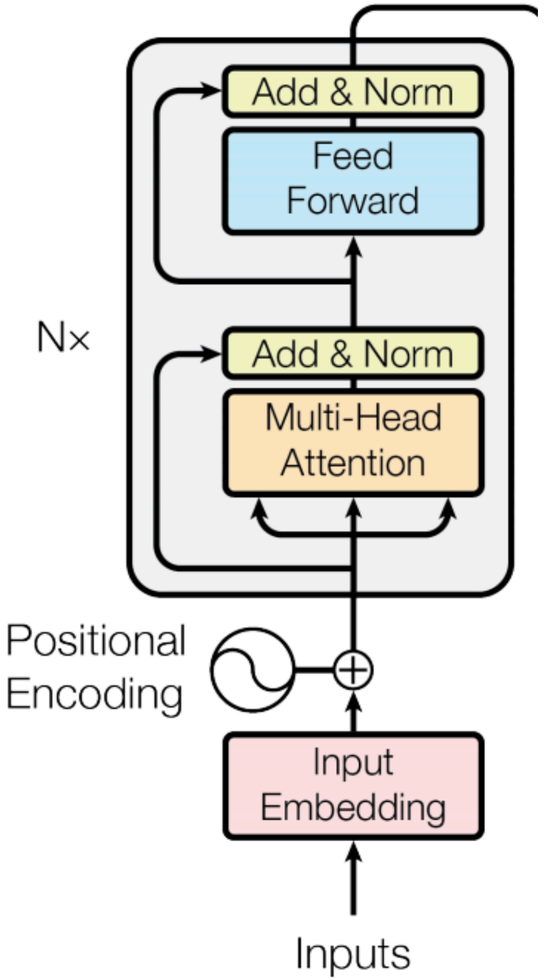
**Input Embedding** : 임베딩 테이블을 통해 각 토큰을 임베딩 벡터로 변환한다. 임베딩 테이블은 그래디언트를 받아 트레이닝 가능한 상태로 둔다.

**Positional Encoding** : Attention 레이어는 CNN, RNN과 다르게 위치에 대한 정보가 출력에 반영되지 않으므로 별도로 위치 정보를 임베딩하는 것이 필요하다. 위치 값을 아래와 같은 sin과 cos함수에 적용한 출력 값을 Input Embedding 결과에 더한다.

$$PE_{(pos,2i)} = \sin\left(\text{pos}/10000^{2i/d_{\text{novel}}}\right)$$

$$PE_{(\text{pos},2i+1)} = \cos\left(\text{pos}/10000^{2i/d_{\text{nodel}}}\right)$$

# Transformer - Encoder



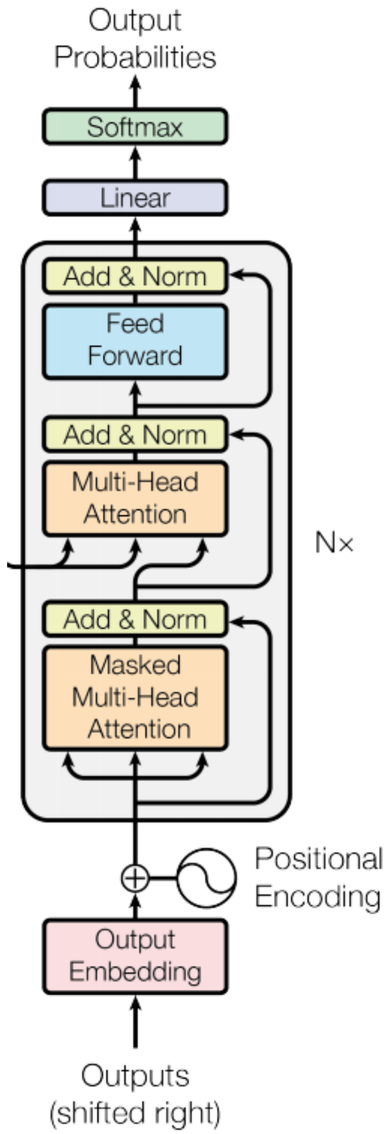

**Multi-Head Attention** : 입력으로 들어온 벡터를 여러 Head로 나누어 각각 다른 어텐션 정보를 계산한다. 이러한 작업으로 어텐션 영역 내에서 다양한 정보를 가져다 사용할 수 있는 능력을 부여한다.

**Add & Norm** : Multi-Head Attention의 결과와 입력을 더한다. 이러한 Residual Connection을 통해 깊은 네트워크를 구축하면서도 Backpropagation시 Gradient가 잘 전파되도록 돕는다. 또한 Layer Normalization을 사용하여 Add 연산이 반복됨에도 안정된 값을 출력하도록 한다.

**Feed Forward** : 두 개의 Fully Connected 레이어와 ReLU Activation으로 구성된다. 내부 레이어의 차원은 입력보다 4배로 크게 설정된다.

$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

# Transformer - Decoder




Decoder는 Outputs, Outputs Embedding, Positional Encoding, Attention Blocks으로 구성되며, Attention Blocks내부는 Masked Multi-Head Attention, Feed Forward와 Layer Normalization 으로 구성된다.

**Outputs** : 번역 작업의 경우 대상 문장, 질의 응답 작업의 경우 응답문장에 해당한다. 문자열에 대응하는 토큰이 입력으로 주어진다.

**Outputs Embedding** : Encoder의 Inputs Embedding과 같은 방식으로 작동한다.

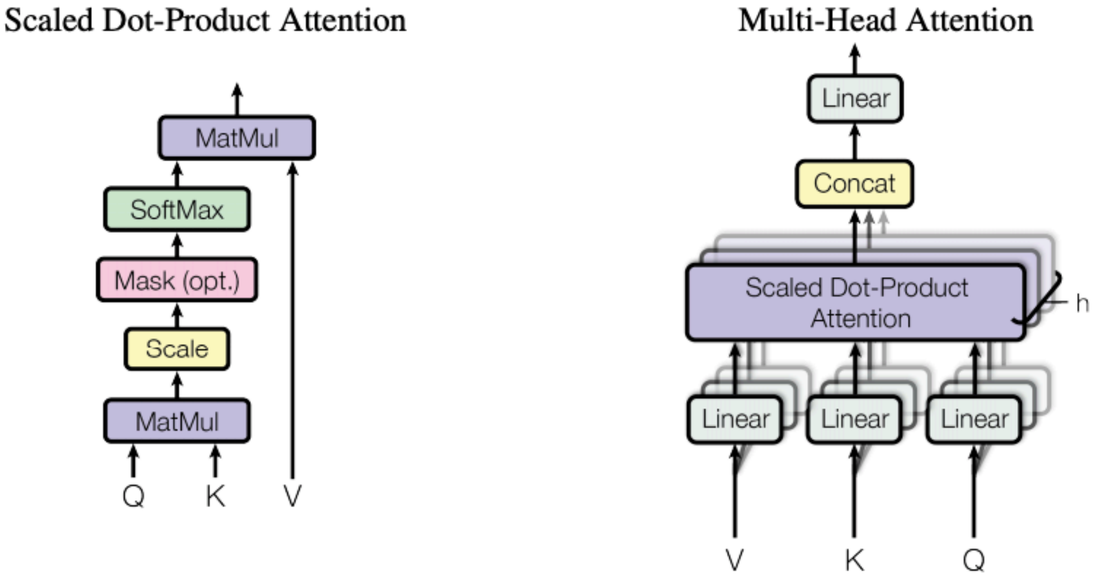
**Positional Encoding** : Encoder의 Positional Encoding과 같은 방식으로 작동한다.

**Masked Multi-Head Attention** : Encoder의 Multi-Head Attention과 같지만 현재보다 과거의 정보들만을 참조하도록 Mask를 사용한다.

**Add & Norm** : Encoder의 Add & Norm과 같은 방식으로 작동한다.

**Feed Forward** : Encoder의 Feed Forward과 같은 방식으로 작동한다.

# Transformer - Multi-Head Attention




Multi-Head Attention은 입력 벡터들을 선형 변환하여 Value, Key, Query 값을 얻고, 각 Query와 Key 값들 간에 dot-product를 취하여 얼마나 정보를 취할지 결정하는 Attention Score를 얻은 뒤, 이 값을 토대로 Value 값들을 선형 결합하여 최종적인 출력 벡터를 얻어낸다. 이 때 입력 벡터를 각 Head로 분리하고 출력 벡터를 concatenation을 통해 다시 합쳐 다양한 어텐션 가능성을 얻을 수 있도록 한다.

$$Attention(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Query와 Key의 내적을 각 Head의 차원 d_k의 루트 값으로 나누어줌으로써 Head의 차원에 관계없이 내적값이 안정되도록 도와준다.

# Summary of Autoregressive Models

- Easy to sample from
  1. Sample $\overline{x}_0 \sim p(x_0)$
  2. Sample $\overline{x}_1 \sim p(x_1 \mid x_0 = \overline{x}_0)$
  3. $\ldots$
- Easy to compute probability $p(x = \overline{x})$
  1. Compute $p(x_0 = \overline{x}_0)$
  2. Compute $p(x_1 = \overline{x}_1 \mid x_0 = \overline{x}_0)$
  3. Multiply together (sum their logarithms)
  4. $\ldots$
  5. Ideally, can compute all these terms in parallel for fast training
- Easy to extend to continuous variables. For example, can choose Gaussian conditionals $p(x_t \mid x_{<t}) = \mathcal{N}(\mu_\theta(x_{<t}), \Sigma_\theta(x_{<t}))$ or mixture of logistics
- No natural way to get features, cluster points, do unsupervised learning
- Next: learning