Pixel CNN - An introduction and brief analysis

Raza Abbas and Muhammad Atif Tahir

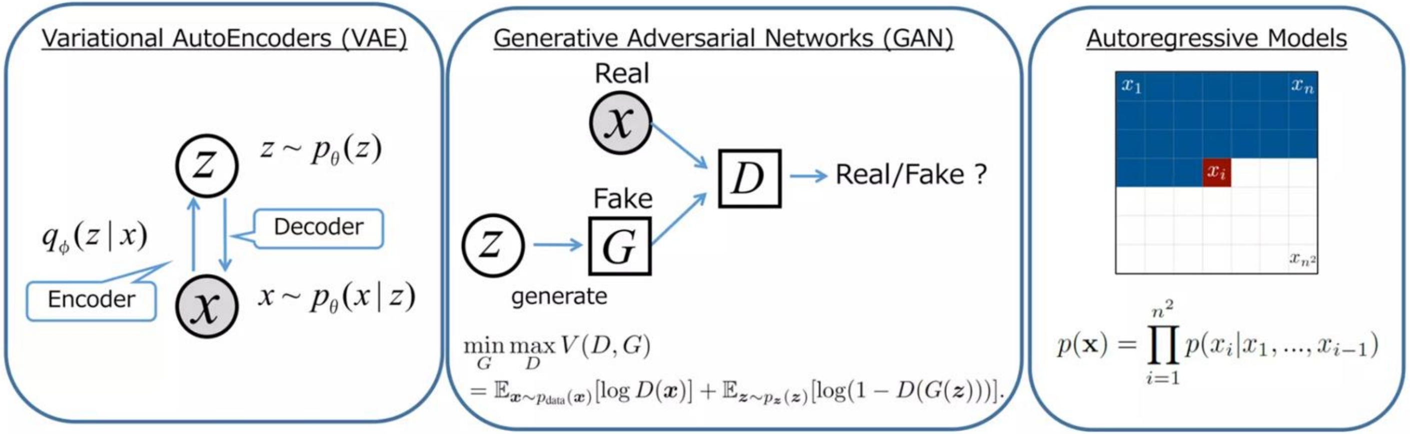
Introduction

- PixelCNN was introduced by Oord et al. in 2016
- The idea behind PixelCNN is to train a network that can generate images autoregressively
- The problem in generating sequential pixels within an image is more complex versus text because textual information follows a pattern/sequence.
- In images, however, the neighbourhood matters and there isn’t a very specific pattern with which the pixels in a sequence can exhibit any behaviour
- PixelCNN attempts to generate images pixel by pixel by predicting the likelihood of the next pixel given the pixels before it.

Introduction

Image Generation Models

-Three image generation approaches are dominating the field:



	VAE	GAN	Autoregressive Models
Pros.	- Efficient inference with approximate latent variables.	- generate sharp image. - no need for any Markov chain or approx networks during sampling.	- very simple and stable training process - currently gives the best log likelihood. - tractable likelihood
Cons.	- generated samples tend to be blurry.	- difficult to optimize due to unstable training dynamics.	- relatively inefficient during sampling

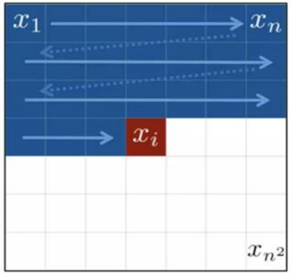
(cf. https://openai.com/blog/generative-models/)

Introduction

Autoregressive Image Modeling

- Autoregressive models train a network that models the conditional distribution of every individual pixel given previous pixels (raster scan order dependencies).

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



⇒ **sequentially predict pixels** rather than predicting the whole image at once (like as GAN, VAE)

- For color image, 3 channels are generated successive conditioning, blue given red and green, green given red, and red given only the pixels above and to the left of all channels.

Comparison

PixelCNN vs RNN

Pixel Recurrent Neural Networks.

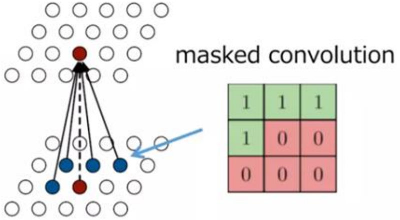
- "Pixel Recurrent Neural Networks" got best paper award at ICML2016.
- They proposed two types of models, **PixelRNN** and **PixelCNN**
(two types of LSTM layers are proposed for PixelRNN.)

PixelRNN

Row LSTM — Diagonal BiLSTM

- LSTM based models are natural choice for dealing with the autoregressive dependencies.

PixelCNN

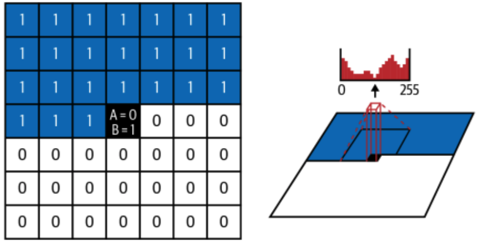


- CNN based model uses masked convolution, to ensure the model is causal.

	PixelRNN	PixelCNN
Pros.	• effectively handles long-range dependencies ⇒ **good performance**	Convolutions are easier to parallelize ⇒ **much faster** to train
Cons.	• Each state needs to be computed sequentially. ⇒ **computationally expensive**	Bounded receptive field ⇒ **inferior performance** **Blind spot problem** (due to the masked convolution) needs to be eliminated.

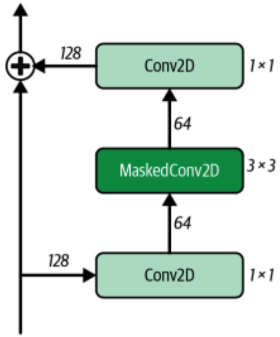
Masked Convolutional Layers

- Convolutional layers are excellent for feature detection.
- However, they can't be directly applied in an autoregressive manner to image generation due to the lack of pixel ordering.
- Unlike text data, where tokens have a clear ordering, pixels in images are treated equally.
- Recurrent models like LSTMs are suitable for text data due to this ordering.
- To apply convolutional layers in an autoregressive manner to image generation:
  - Order must be imposed on the pixels.
  - Filters should only consider preceding pixels.
- This allows for generating images pixel by pixel, predicting the next pixel value from preceding pixels using convolutional filters.



Residual Blocks

- A residual block comprises layers where the output is added to the input before proceeding to the network.
- This addition creates a "skip connection," allowing the input to bypass intermediate layers.
- The skip connection offers a direct route from input to output, aiding in preserving information.
- Including a skip connection simplifies learning by allowing the network to learn residual functions.
- If the optimal transformation is to maintain the input, this can be achieved by zeroing the weights of intermediate layers.
- Without skip connections, the network would need to learn identity mappings through the intermediate layers, which is more complex.



Example

- Consider input of image size 3x3
- I = $\begin{matrix} 2 & 3 & 4 \\ 1 & 5 & 2 \\ 6 & 7 & 8 \end{matrix}$
- By applying, Kernel of size 1x1 i.e. -2 with just one filter, the new output from Conv2D would be
- M = $\begin{matrix} -4 & -6 & -8 \\ -2 & -10 & -4 \\ -12 & -14 & -16 \end{matrix}$

Example

- For Maskedconv2D assume type A and masking at 3rd row and 1st row, thus maked image would be
- M* = $\begin{matrix} -4 & -6 & -8 \\ -2 & -10 & -4 \\ 0 & 0 & 0 \end{matrix}$
- Assume by applying, Kernel of size 1x1 for maskedconv2D i.e. -1 with just one filter, the new output from Maskedconv2D would be
- N = $\begin{matrix} 4 & 6 & 8 \\ 2 & 10 & 4 \\ 0 & 0 & 0 \end{matrix}$

Example

- Now apply 1x1 filter of +1 for conv2D, output will be same

- O = $\begin{matrix} 4 & 6 & 8 \\ 2 & 10 & 4 \\ 0 & 0 & 0 \end{matrix}$

- Adding with skip connection

- New output = I + O = $\begin{matrix} 2 & 3 & 4 \\ 1 & 5 & 2 \\ 6 & 7 & 8 \end{matrix} + \begin{matrix} 4 & 6 & 8 \\ 2 & 10 & 4 \\ 0 & 0 & 0 \end{matrix} = \begin{matrix} 6 & 9 & 12 \\ 3 & 15 & 6 \\ 6 & 7 & 8 \end{matrix}$

Analysis and Issues

Performance Impact:

- Generating new images with autoregressive models involves predicting the next pixel given all preceding pixels sequentially.
- This process is considerably slower compared to models like variational autoencoders (VAEs).
- For a 32×32 grayscale image, autoregressive models require making 1,024 sequential predictions, while VAEs require only a single prediction.
- Autoregressive models, like PixelCNN, suffer from slow sampling due to the sequential nature of prediction.

Blindspot Problem:

- Receptive fields marked in yellow illustrate how blindspots propagate through c
- GatedPixelCNN is a solution to address this issue.

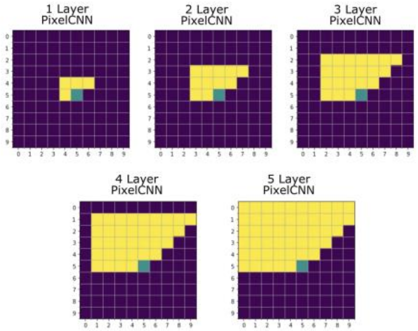


Figure 2: Evolution of the blind spot on the PixelCNN (Image by author).

References

- David Foster - Generative Deep Learning Techniques
- https://medium.com/data-science-in-your-pocket/generative-modeling-using-pixelcnn-with-codes-explained-387c95405651
- https://www.slideshare.net/suga93/conditional-image-generation-with-pixelcnn-decoders
- https://paperswithcode.com/method/pixelcnn#:~:text=A%20PixelCNN%20is%20a%20generative,as%20a%20product%20of%20conditionals.
- https://towardsdatascience.com/pixelcnns-blind-spot-84e19a3797b9

Pixel RNN

Step 1. Input & hidden state

- PixelRNN is an **autoregressive model**.
- At each pixel position i, the RNN (LSTM or Diagonal LSTM) computes a **hidden state** h_i, which summarizes all the previous pixels $(x_1, \ldots, x_{i-1})$.

Step 2. Linear transformation

- That hidden state is passed through a **linear layer** (weight matrix W + bias b):

$$\text{logits}_i = W h_i + b$$

- These logits are just raw scores (one score for each possible pixel intensity, e.g., 256 values for grayscale)

Step 3. Softmax

- The logits are fed into a **softmax function**:

$$p(x_i = v \mid x_{<i}) = \frac{\exp(\text{logits}_i[v])}{\sum_{u=0}^{255} \exp(\text{logits}_i[u])}$$

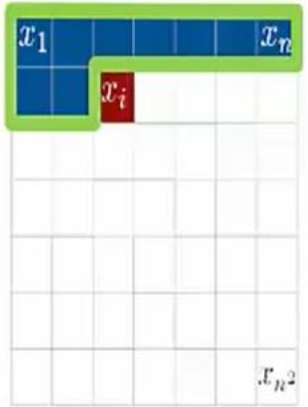
- This converts the scores into **probabilities** that sum to 1.

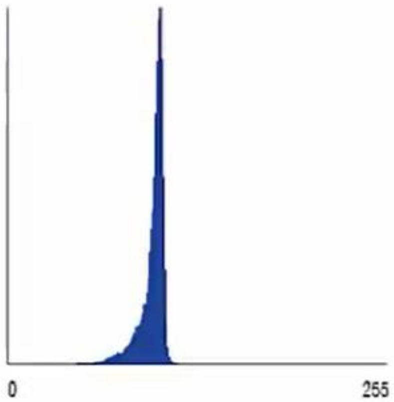
RGB case

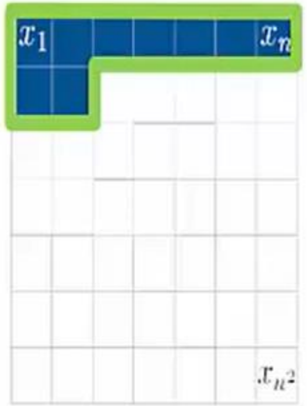
PixelRNN factorizes further:

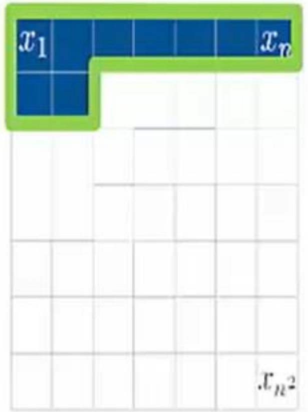
- First generate **R** with a softmax over 256 values.
- Then generate **G**, conditioned on R (again softmax).
- Then generate **B**, conditioned on R and G.

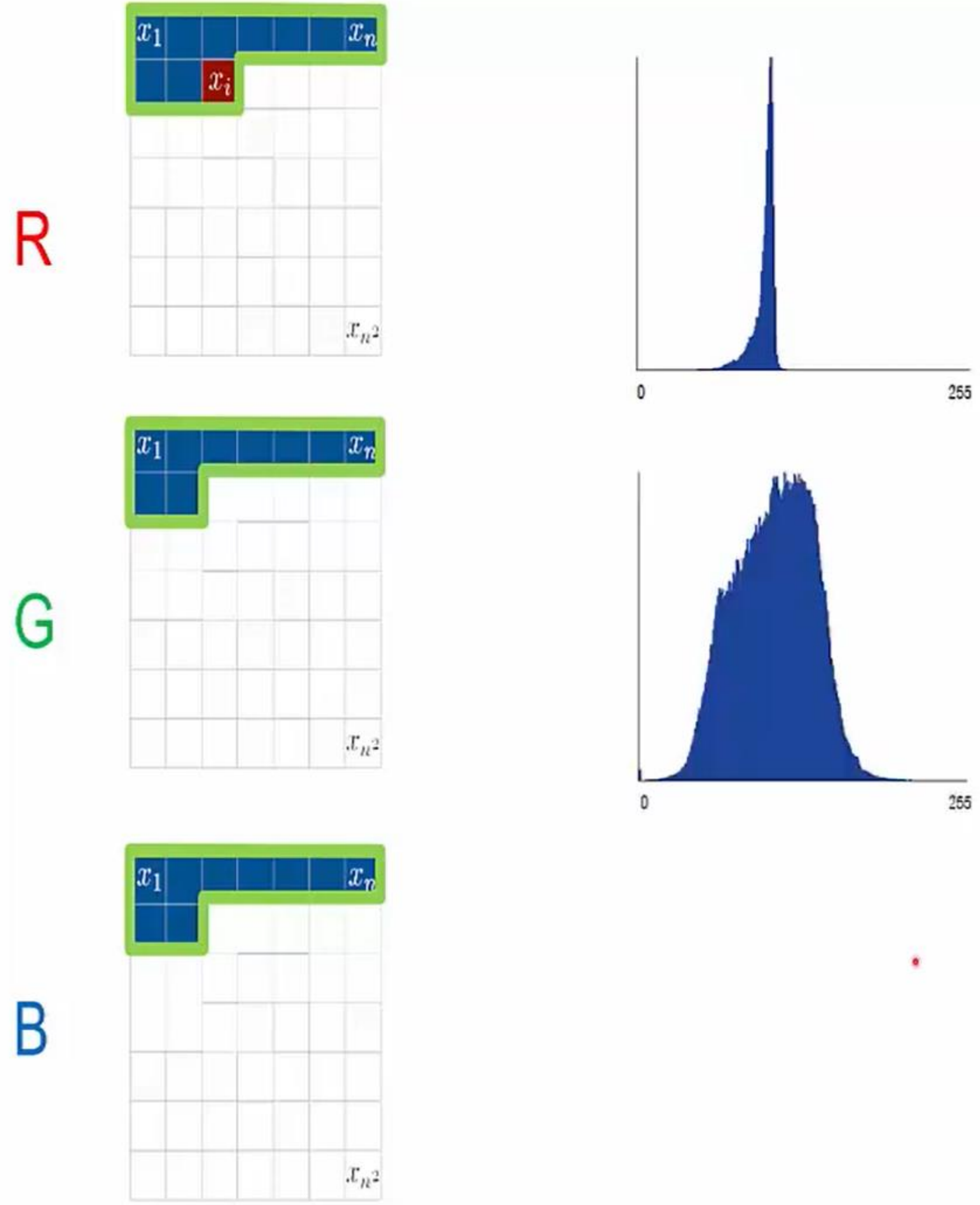
So yes — a softmax distribution is produced **for each channel of each pixel**.

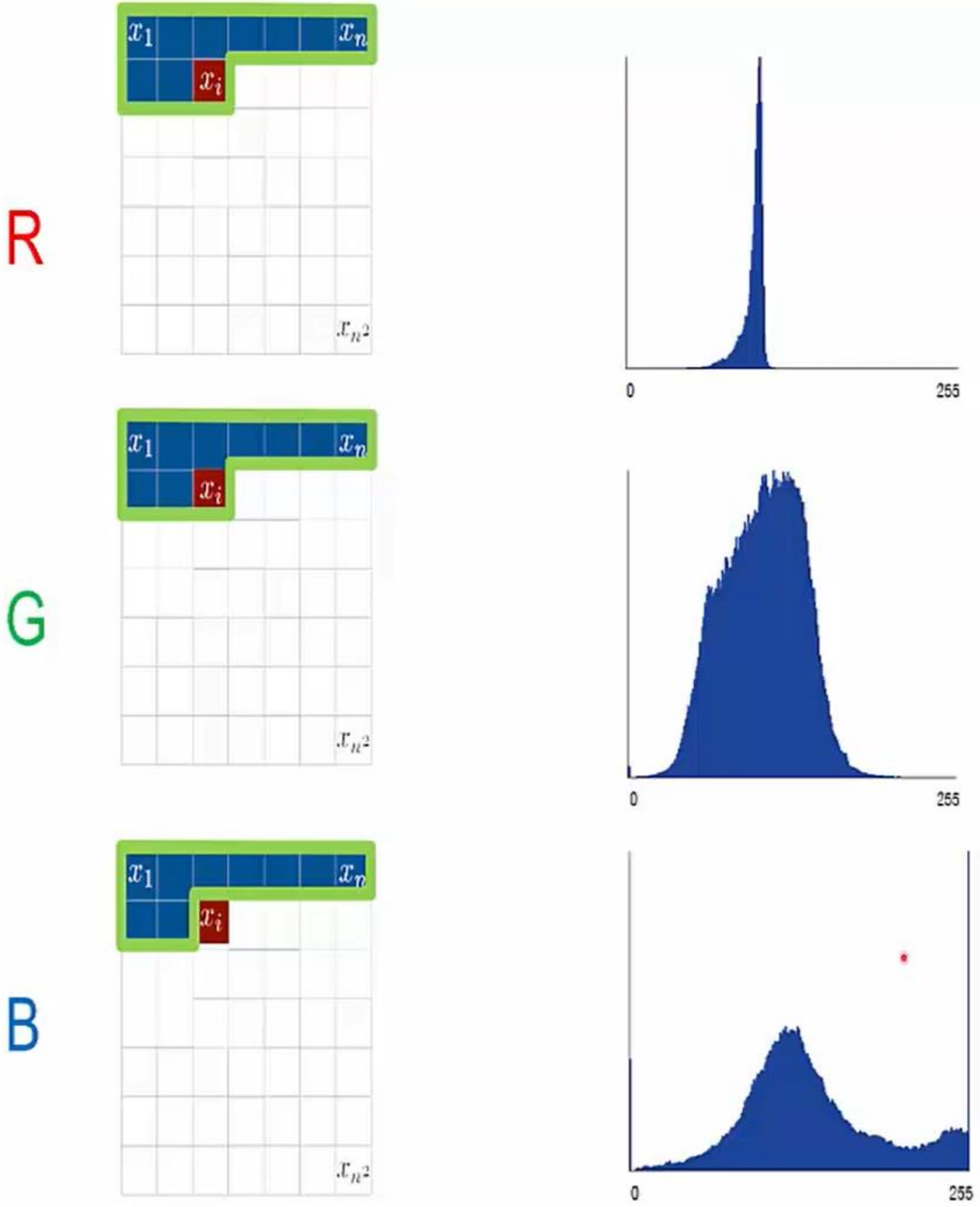

x_1
x_n
x_i
x_{n^2}


0
255

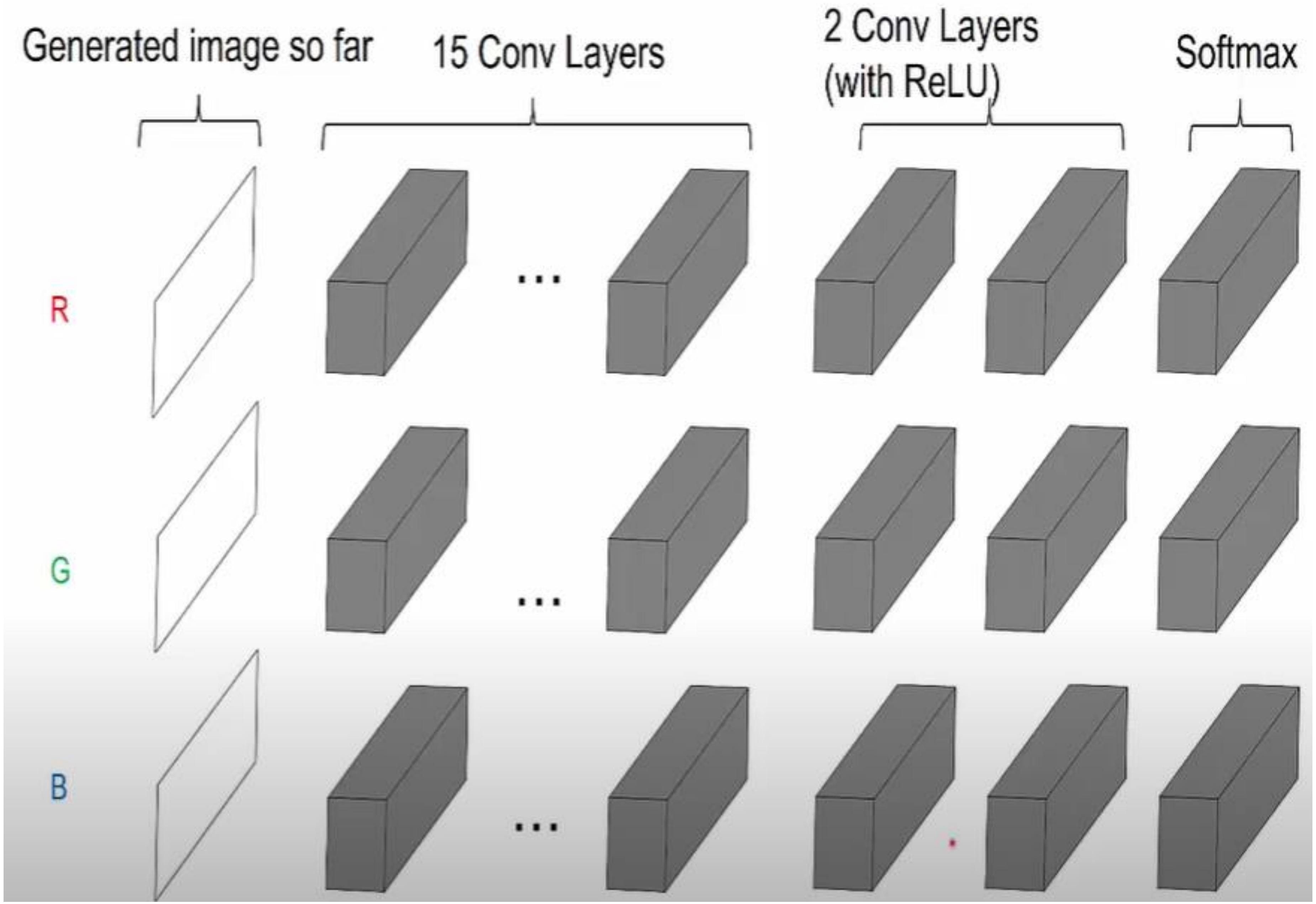

x_1
x_n
x_{n^2}

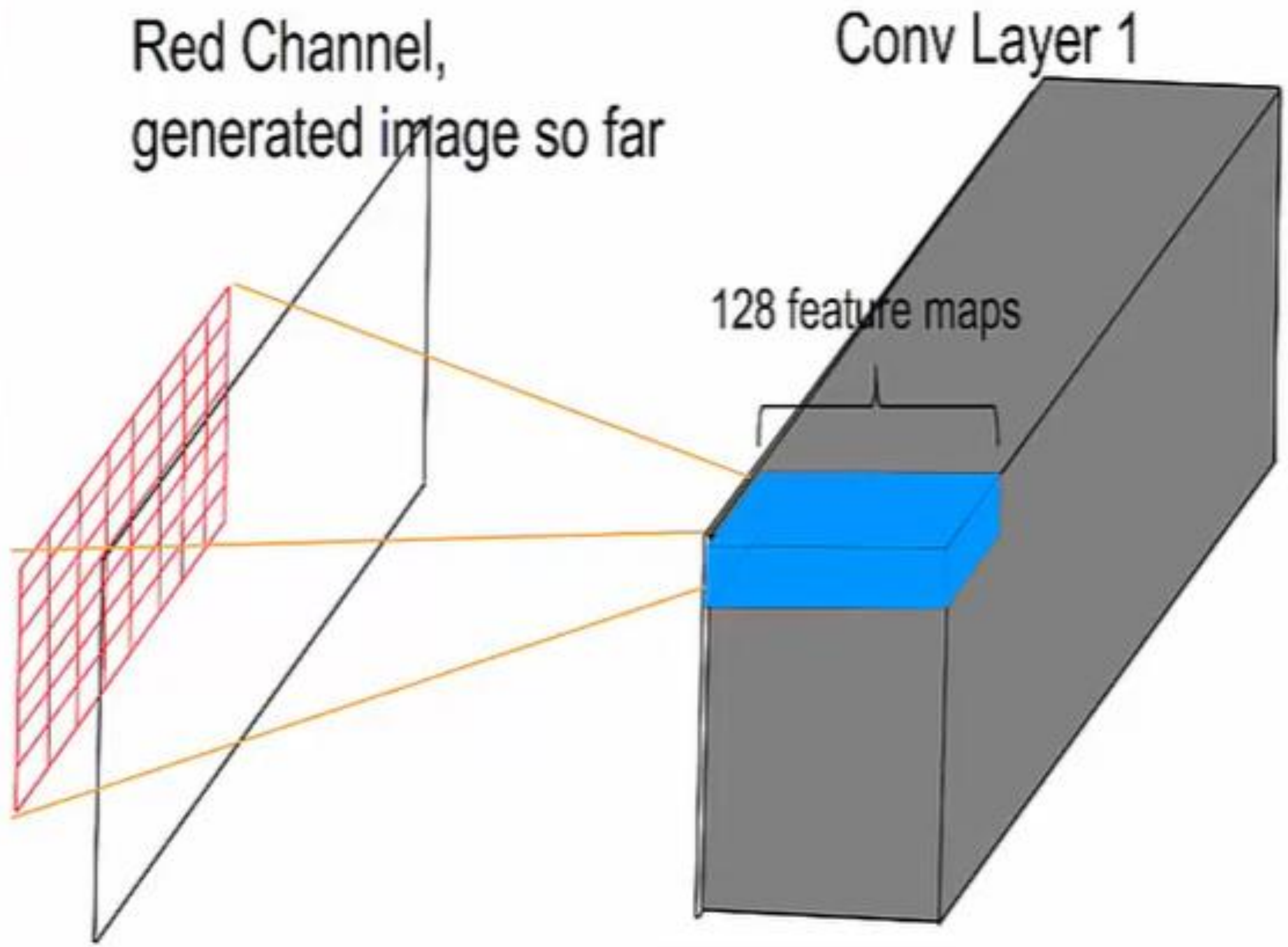

x_1
x_n
x_{n^2}

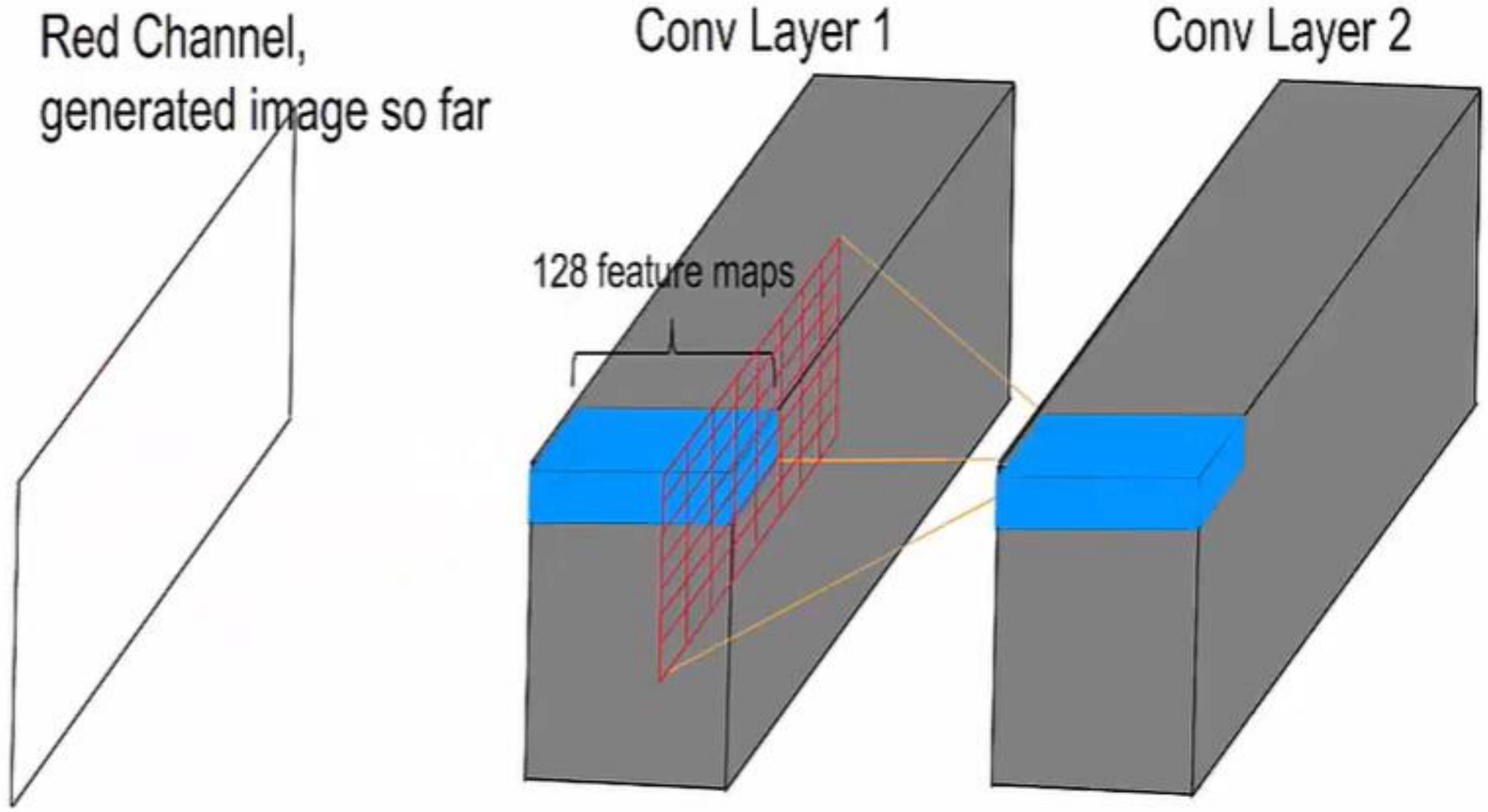

R
G
B
x_1
x_n
x_i
x_{n^2}
0
255

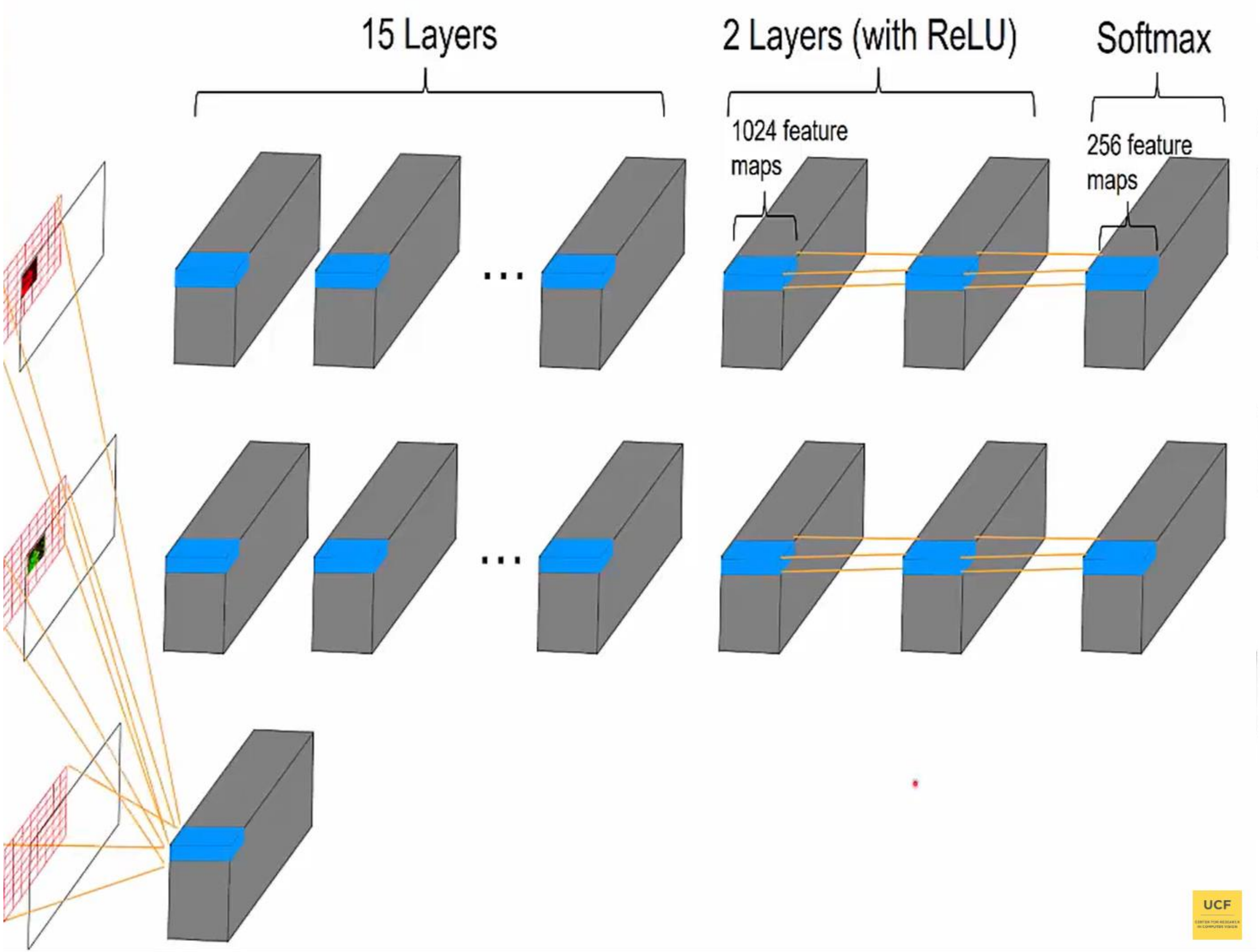

R
G
B
x_1
x_n
x_i
x_{n^2}
0
255

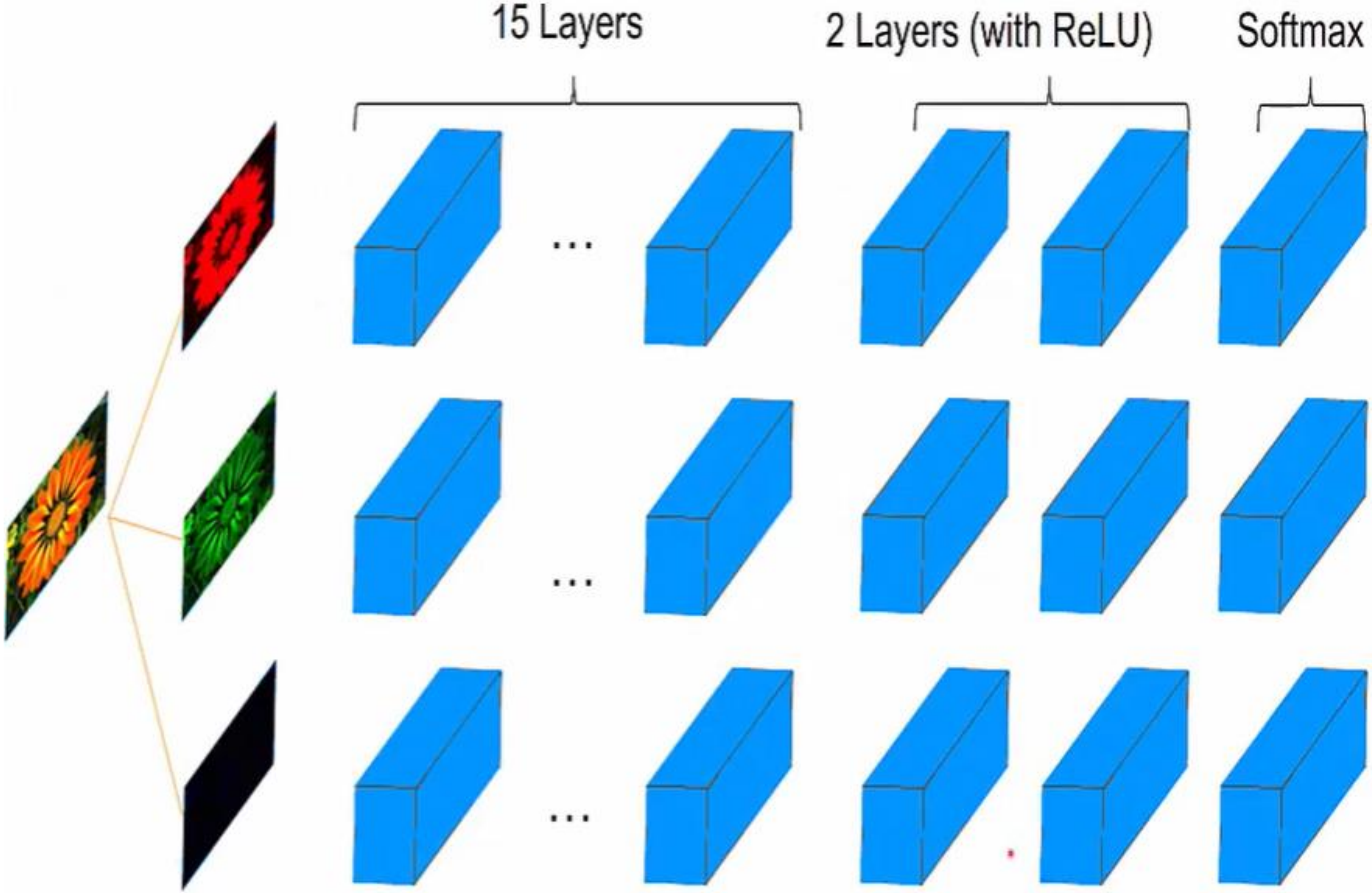
PixelCNN


Generated image so far
15 Conv Layers
2 Conv Layers
(with ReLU)
Softmax
R
G
B
...
...
...


Red Channel,
generated image so far
Conv Layer 1
128 feature maps


Red Channel,
generated image so far
Conv Layer 1
Conv Layer 2
128 feature maps


15 Layers
2 Layers (with ReLU)
Softmax
1024 feature maps
256 feature maps
...
...
UCF



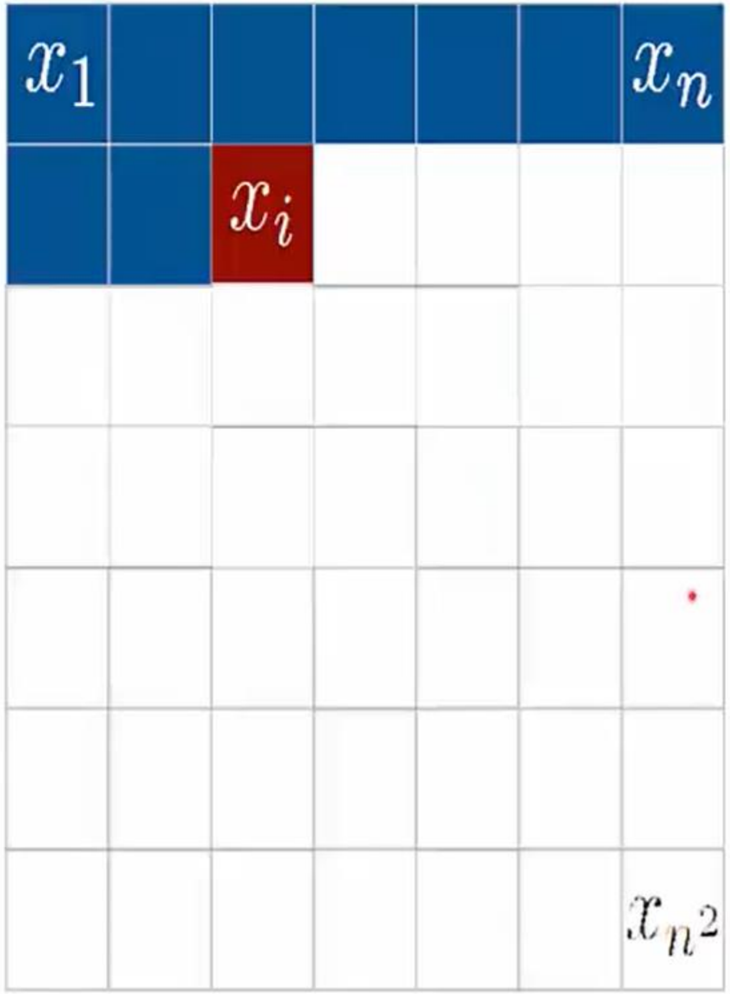
15 Layers
2 Layers (with ReLU)
Softmax
...
...
...

Kernel Mask

- We do not want to look at future pixels

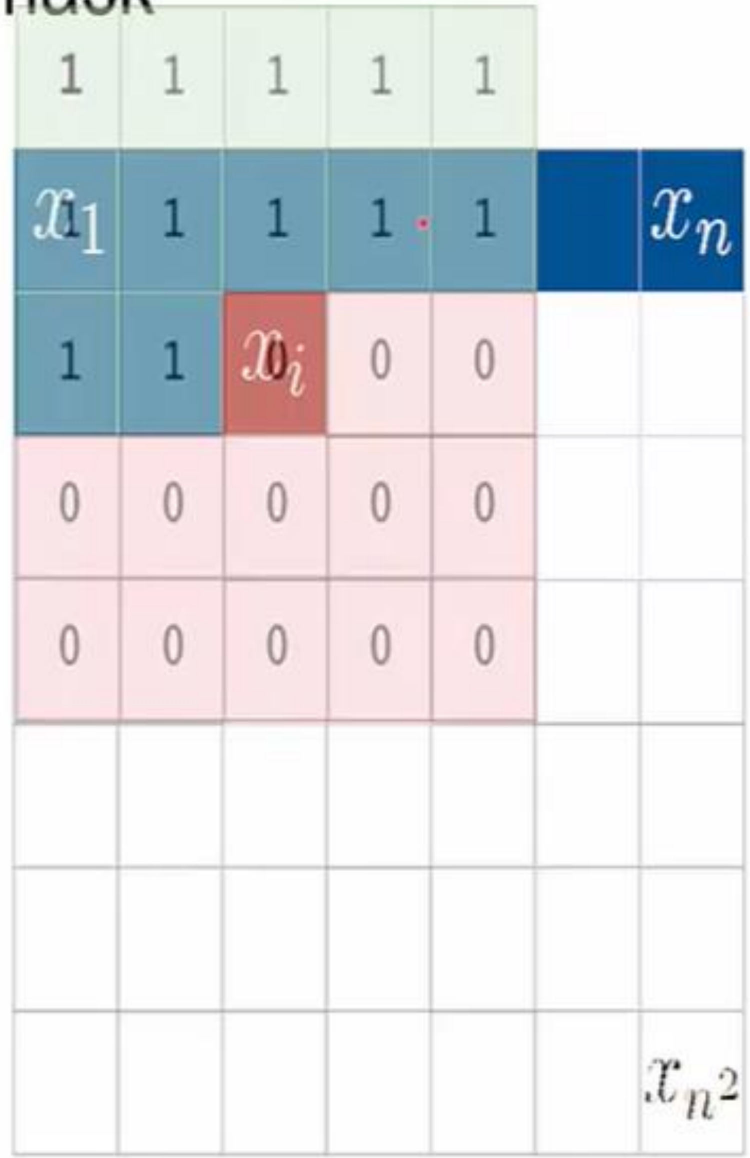
1	1	1	1	1
1	1	1	1	1
1	1	0	0	0
0	0	0	0	0
0	0	0	0	0

Kernel mask

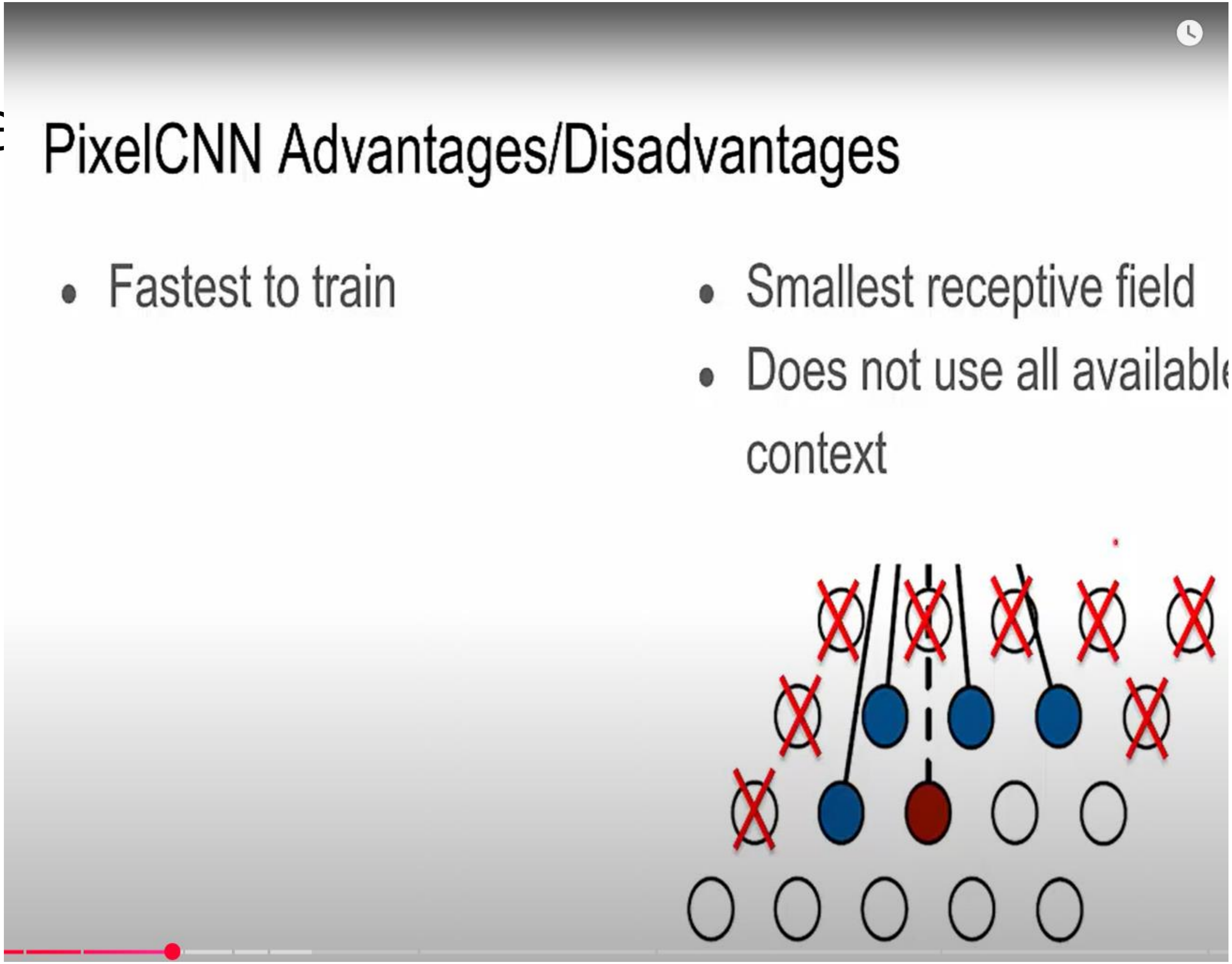


1	1	1	1	1
1	1	1	1	1
1	1	0	0	0
0	0	0	0	0
0	0	0	0	0

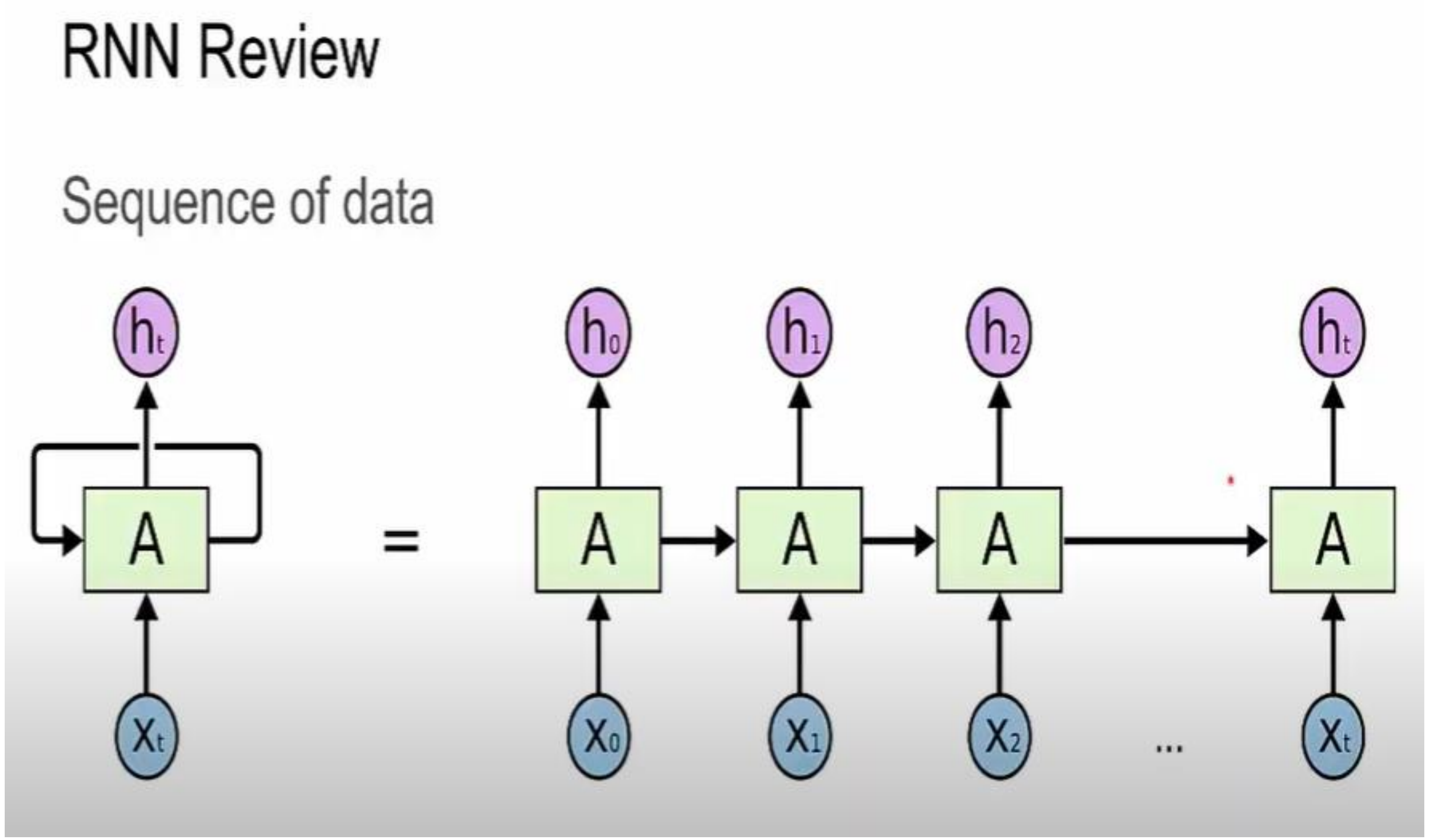
Kernel mask



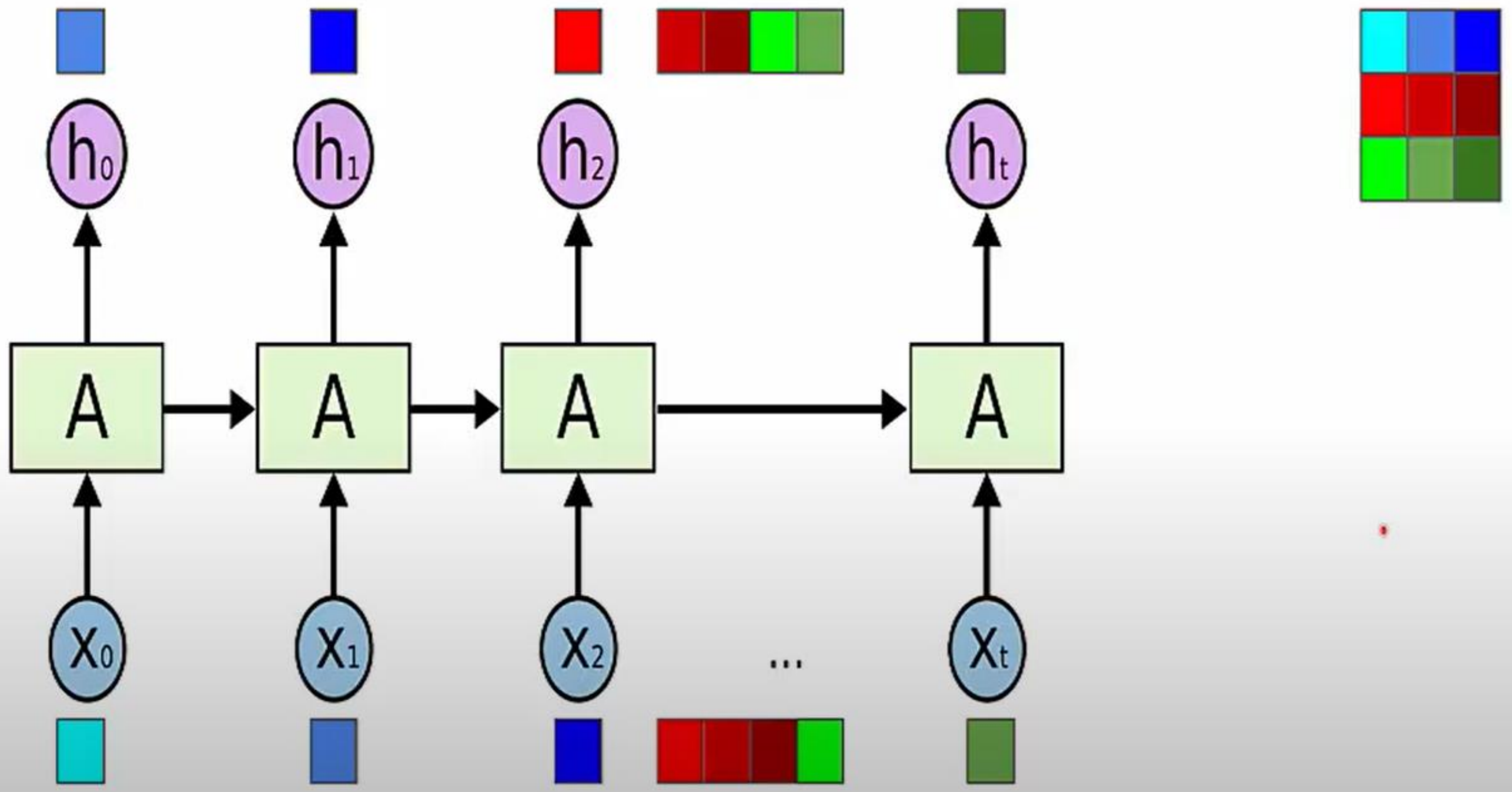
Pixe


PixelCNN Advantages/Disadvantages
• Fastest to train
• Smallest receptive field
• Does not use all availabl
context

RNN Review



RNN for Image Generation



LSTM Equations

$$i = \sigma(x_i U^i + \mathbf{h}_{i-1} W^i)$$

$$f = \sigma(x_i U^f + \mathbf{h}_{i-1} W^f)$$

$$o = \sigma(x_i U^o + \mathbf{h}_{i-1} W^o)$$

$$g = \ tanh(x_i U^g + \mathbf{h}_{i-1} W^g)$$

Gates - Control how much information is allowed through

$$c_i = c_{i-1} \circ f + g \circ i$$

$$\mathbf{h}_i = \tanh(c_i) \circ o$$

States - Hold information about all time steps up till now {0, i, …, i-1, i}

LSTM Equations

$$i = \sigma(x_i \cancel{U^i} + \mathbf{h}_{i-1} \cancel{W^i})$$

$$f = \sigma(x_i \cancel{U^f} + \mathbf{h}_{i-1} \cancel{W^f})$$

$$o = \sigma(x_i \cancel{U^o} + \mathbf{h}_{i-1} \cancel{W^o})$$

$$g = tanh(x_i \cancel{U^g} + \mathbf{h}_{i-1} \cancel{W^g})$$

$$c_i = c_{i-1} \circ f + g \circ i$$

$$\mathbf{h}_i = \tanh(c_i) \circ o$$

Like Convolutional LSTM - replaced fully-connected layer with convolutional layer

$$h_t = f(W_{hh} * h_{t-1} + W_{xh} * x_t)$$

Concept	Row LSTM notation	Generic ConvLSTM notation
Previous row / recurrent term	$W_v * h_{\text{above}}$	$k_{ss} * h_{t-1}$
Input / current pixel term	$W_h * h_{\text{left}}$	$k_{is} * x_i$

$$[\mathbf{o}_i, \mathbf{f}_i, \mathbf{i}_i, \mathbf{g}_i] = \sigma(\underline{\mathbf{K}^{ss}} \circledast \mathbf{h}_{i-1} + \underline{\mathbf{K}^{is}} \circledast \mathbf{x}_i)$$

K^{ss} = convolutional filters applied to the **previous hidden state** h_{i-1}

K^{is} = convolutional filters applied to the **current input** x_i

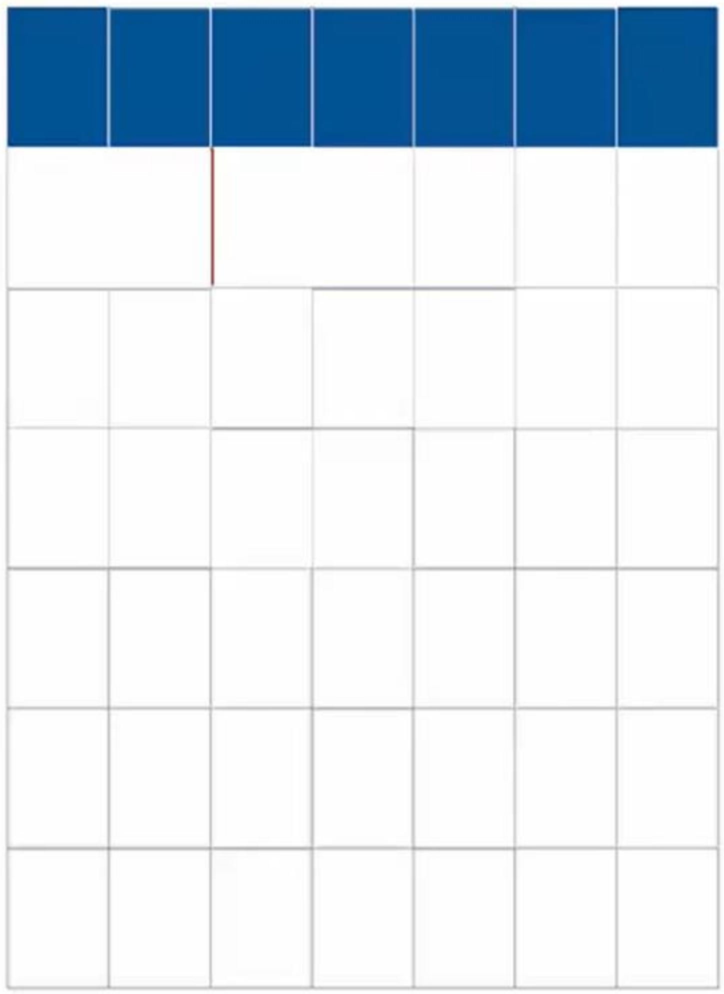
$$\mathbf{c}_i = \mathbf{f}_i \odot \mathbf{c}_{i-1} + \mathbf{i}_i \odot \mathbf{g}_i$$

$$\mathbf{h}_i = \mathbf{o}_i \odot \tanh(\mathbf{c}_i)$$

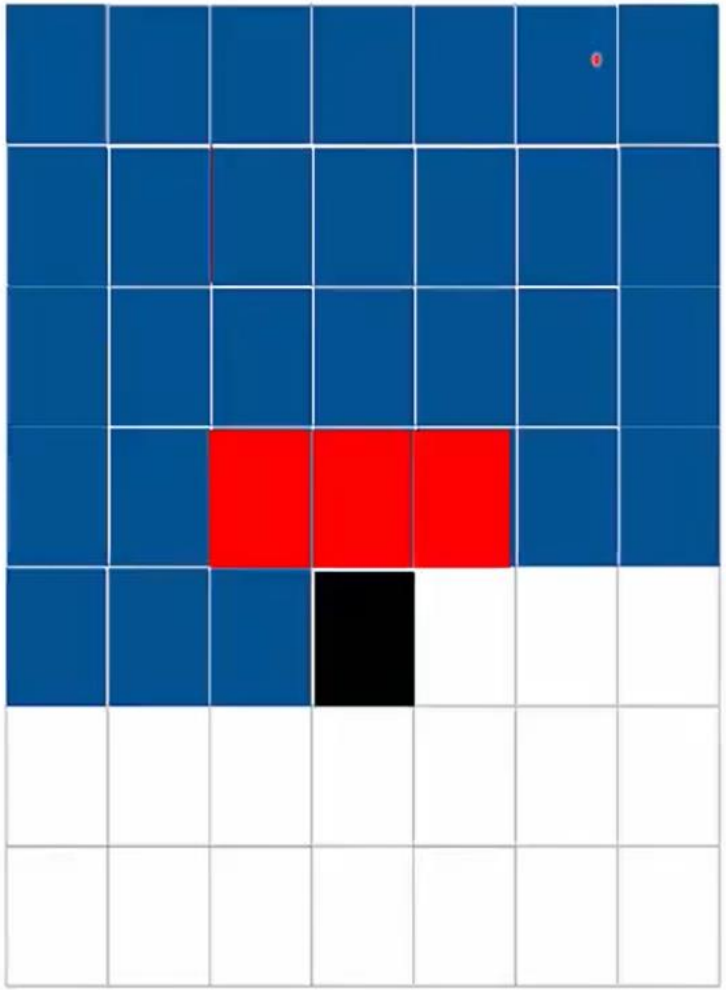
UCF

Row LSTM

- Each state represents the entire row

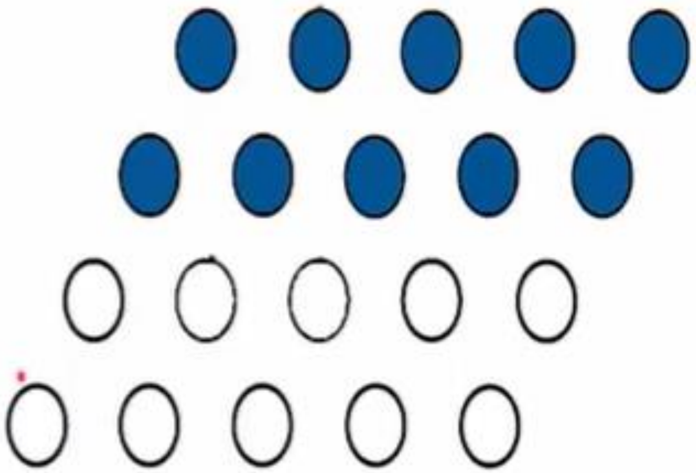


Row LSTM

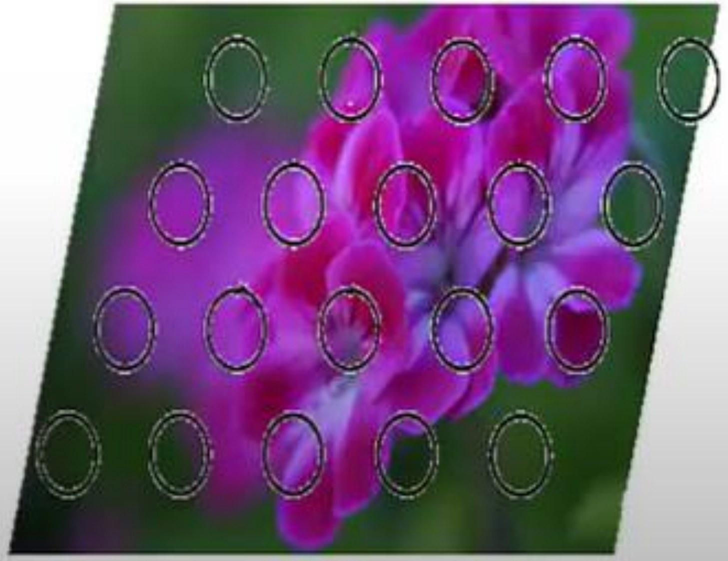


Input-to-State Component

$$[\mathbf{o}_i, \mathbf{f}_i, \mathbf{i}_i, \mathbf{g}_i] = \sigma(\mathbf{K}^{ss} \circledast \mathbf{h}_{i-1} + \underline{\mathbf{K}^{is} \circledast \mathbf{x}_i})$$

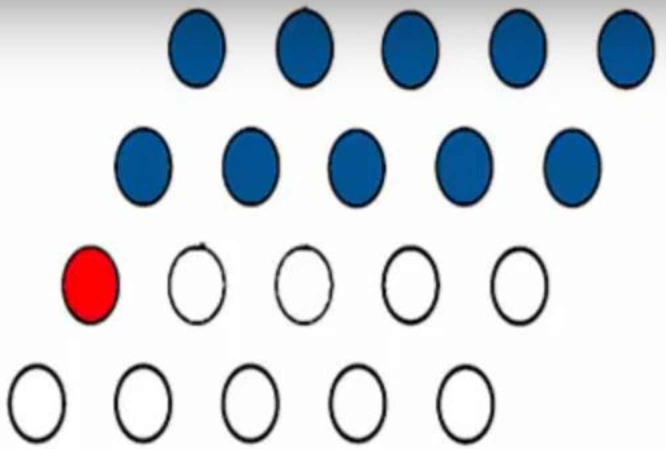


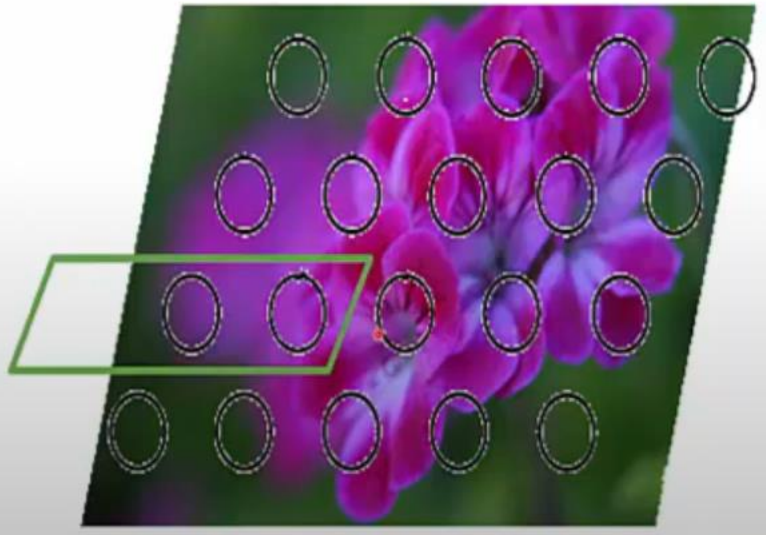
Concept	Row LSTM notation	Generic ConvLSTM notation
Previous row / recurrent term	$W_v * h_{\text{above}}$	$k_{ss} * h_{t-1}$
Input / current pixel term	$W_h * h_{\text{left}}$	$k_{is} * x_i$



Input-to-State Component

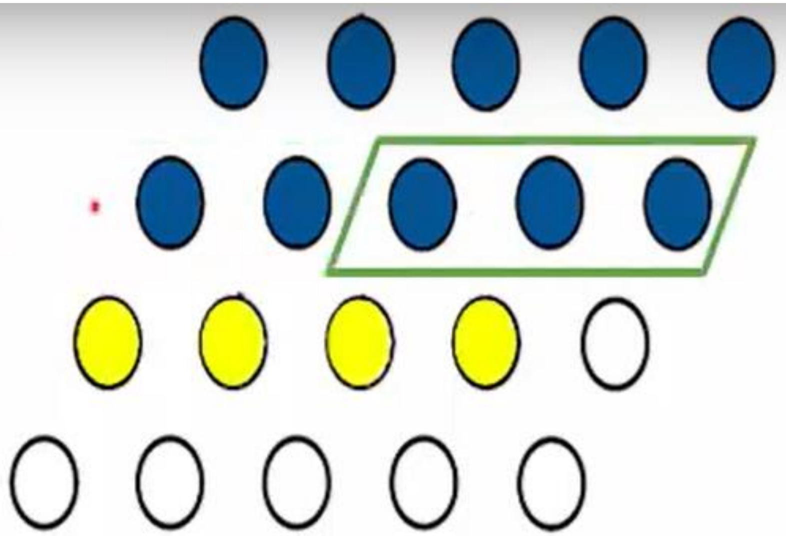
$$[\mathbf{o}_i, \mathbf{f}_i, \mathbf{i}_i, \mathbf{g}_i] = \sigma(\mathbf{K}^{ss} \circledast \mathbf{h}_{i-1} + \underline{\mathbf{K}^{is} \circledast \mathbf{x}_i})$$





State-to-State Component

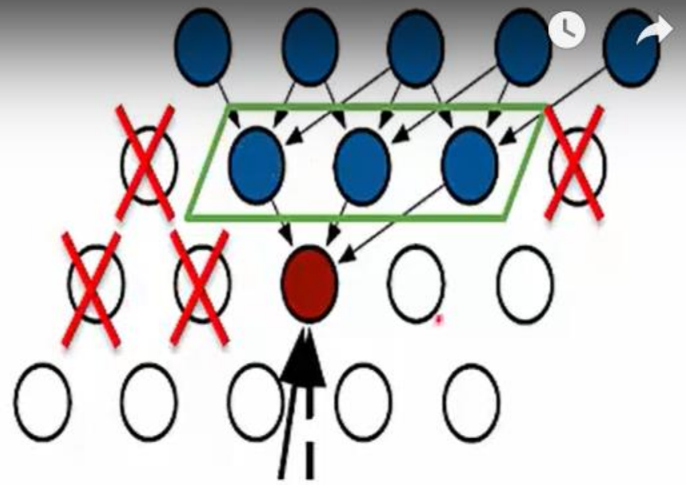
$$[\mathbf{o}_i, \mathbf{f}_i, \mathbf{i}_i, \mathbf{g}_i] = \sigma(\underline{\mathbf{K}^{ss} \circledast \mathbf{h}_{i-1}} + \mathbf{K}^{is} \circledast \mathbf{x}_i)$$



Combine State Components

$$[\mathbf{o}_i, \mathbf{f}_i, \mathbf{i}_i, \mathbf{g}_i] = \sigma(\underline{\mathbf{K}^{ss} \circledast \mathbf{h}_{i-1} + \mathbf{K}^{is} \circledast \mathbf{x}_i})$$

Advantages and Disadvantages

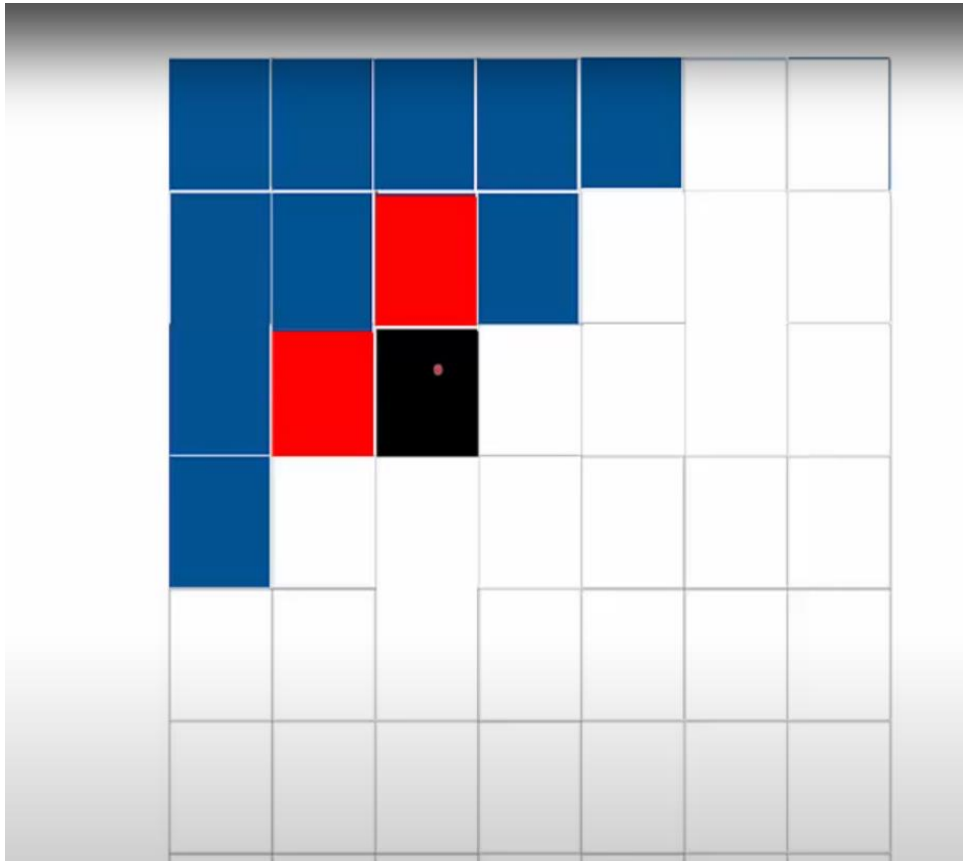


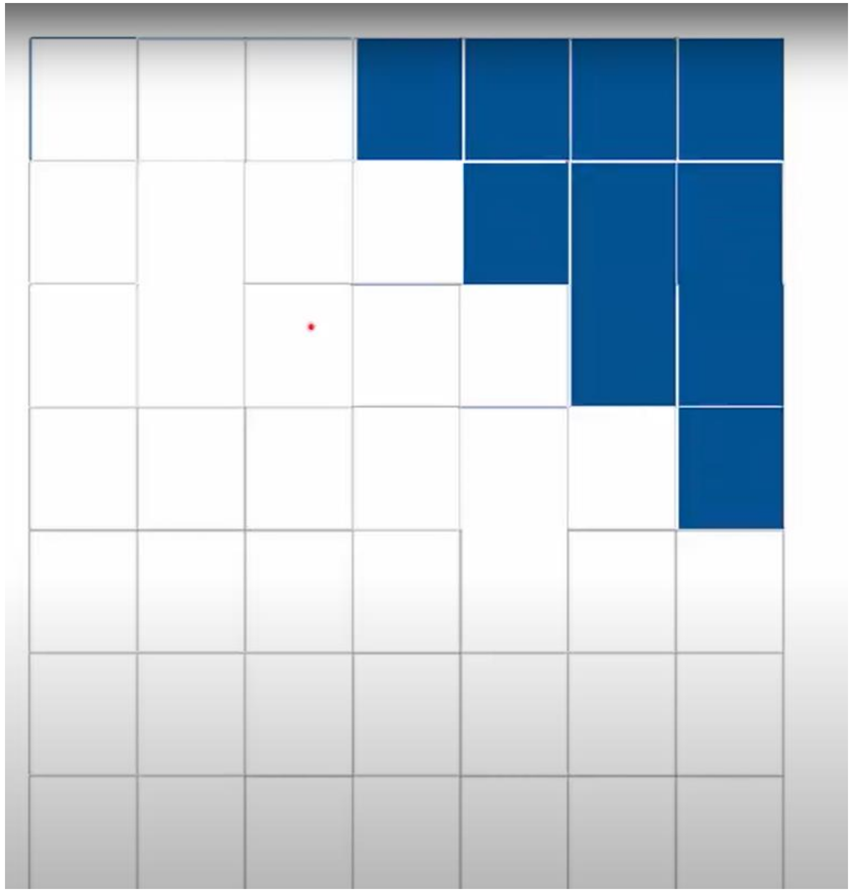
Compute the state for an entire row at once

Triangular receptive field

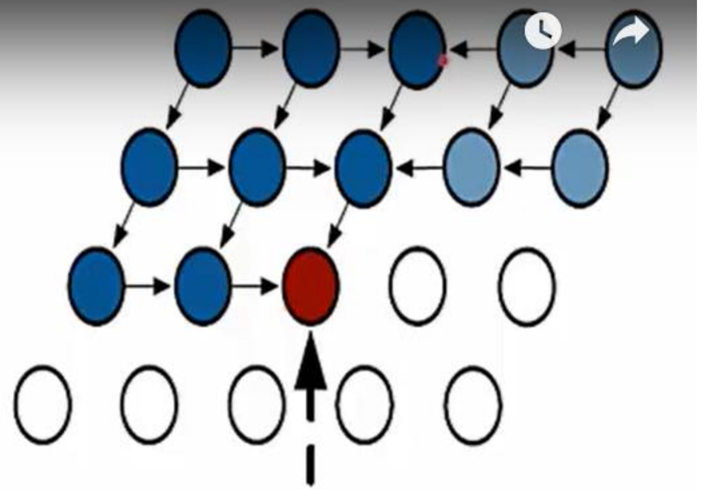
Does not use all available context

Diagonal BiLSTM





Advantages



Compute the state for an entire diagonal at once

Global receptive field

Uses all available context

PixelRNN and PixelCNN Training Comparison

1. Input

- **PixelRNN**: Feed the whole image, but RNN scans it **pixel-by-pixel** in a fixed order (e.g., row by row).
- **PixelCNN**: Feed the whole image at once into a convolutional network.

---

2. Autoregressive constraint

- **PixelRNN**: The RNN hidden state at pixel i only has access to pixels $x_{<i}$. Sequential structure enforces this.
- **PixelCNN**: Uses **masked convolutions** so each pixel's prediction depends only on previous pixels ($x_{<i}$).

---

3. Output

- **PixelRNN**: At step i, produces a **logits vector of length 256** (distribution over possible pixel values).
- **PixelCNN**: For **every pixel position simultaneously**, produces logits vectors of length 256.

---

4. Probability distribution

- **PixelRNN**: Softmax over logits → probability distribution for pixel i.
- **PixelCNN**: Softmax over logits → probability distribution for every pixel in the image.

---

5. Loss (Training signal)

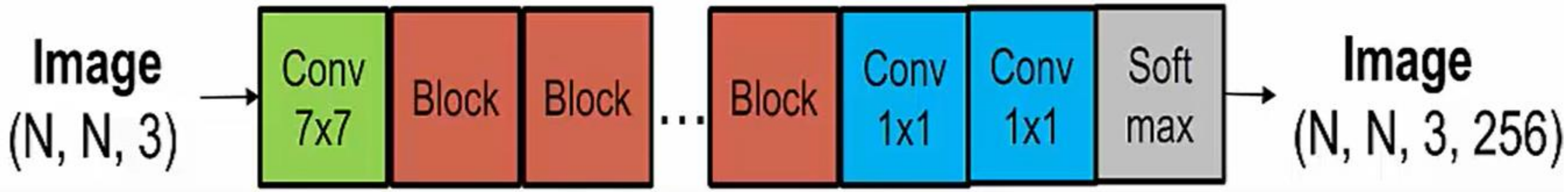
- **Both**: Use **cross-entropy loss** between predicted probability distribution and the true pixel intensity (0–255).

---

6. Training speed

- **PixelRNN**: **Slow**, must process sequentially → no parallelization across pixels.
- **PixelCNN**: **Fast**, can train on all pixels in parallel (thanks to convolutions).

Typical Architecture



What the diagram is doing (common parts)

- The model is autoregressive: it factorizes the image distribution as
  $p(x) = \prod_i p(x_i \mid x_{<i})$ (usually raster scan order).
  The network outputs a distribution for each pixel (and each color channel) — that's why the final shape is (N, N, 3, 256): 256 logits for each RGB channel intensity.
- **Conv 7×7**: an initial feature embedding that increases receptive field / transforms raw pixel values into internal feature maps.
- **Blocks … Block**: a stack of layers that build up a representation which only depends on already-generated pixels (this is where causality is enforced).
- **Conv 1×1, Conv 1×1, Softmax**: channel-mixing and the final logits (softmax over 256 intensity levels or some other output parametrization).

How PixelCNN implements the blocks

- Each “Block” = masked convolutional layers (3×3, etc.), often with gated activations and residual connections (PixelCNN, Gated PixelCNN, PixelCNN++ variants).
- Causality is enforced **by masks on convolution kernels** so that when computing features at position (i,j) the convolution never uses pixels at positions ≥ (i,j) (future positions). There are standard mask types:
  - **Mask A** (first layer): excludes the current pixel’s own value (so the network cannot peek at the pixel it must predict).
  - **Mask B** (later layers): allows use of the current pixel’s previously predicted channels (for channel ordering).

    Example 3×3 Mask A (1 = allowed, 0 = forbidden):
- Advantages: during training the entire image can be processed in parallel (masked convs are parallelizable). During sampling you still must generate pixels sequentially.
- Many practical PixelCNNs use residual blocks, gated activations, dilations, and sometimes different output parameterizations (e.g., mixtures of logistics in PixelCNN++ instead of a 256-way softmax).

How PixelRNN implements the blocks

- Each “Block” = recurrent layers that run across the image: **Row LSTMs** (scan left→right row by row) or **Diagonal BiLSTMs** (scan along diagonals to get more context).
- Causality is enforced by the recurrence / scan order itself: the LSTM state at (i,j) only contains information from previously visited pixels.
- PixelRNNs may also include convolutions for local feature extraction, but the core dependency modeling is recurrent rather than masked convolution.
- Disadvantages: less parallelism (row/diagonal recurrences are more sequential than masked convs) so training is slower in practice. They can capture very long-range dependencies well however.

Practical differences (why choose one over the other)

- **Parallelism / speed**: PixelCNN (masked convs) trains faster because operations are parallel across pixels. PixelRNN is more sequential and slower.
- **Receptive field / dependencies**: PixelRNN can model certain long-range patterns naturally via recurrence; PixelCNN achieves similar receptive fields by stacking many masked convs, using dilations, or large kernels.
- **Sample quality**: historically PixelRNN sometimes produced slightly different samples, but later PixelCNN variants (PixelCNN++) matched or exceeded performance while keeping parallel training.
- **Output parametrization**: both families can use softmax over 256 bins or continuous parametrizations (mixtures of logistics) — that is independent of whether blocks are convs or RNNs.

Mapping the diagram to each model

- If you label each orange **Block** = “masked conv + gated nonlinearity + residual” → that picture is a PixelCNN chart.
- If you label each orange **Block** = “row/diagonal LSTM layer (possibly with convs)” → that picture is a PixelRNN chart.
- The final 1×1 convs + softmax are common: they convert internal features into per-pixel (and per-channel) logits.