CS/DS 552: Class 13

Jacob Whitehill

Generative models

	Shallow/Linear	Deep/Non-linear
Latent variable model (LVM)		
Continuous	probabilistic PCA†	VAE,† diffusion†
Discrete	*k*-means,*† GMM†	VQ-VAE†
Autoregressive		
Continuous	linear dynamical system,† AR(*p*)*†	RNN,*† S3 (& related) models*†
Discrete	conditional probability tables†	
Attentional		transformer*†
Adversarial		GAN*†

* Squared-error / log-loss minimization

† Maximum likelihood estimation (MLE)

Autoregressive image generation

Autoregressive image generation

- So far, the deep generative models we have seen used a neural decoder to generate an image from latent vector **z**.
    - VAE
    - GAN
    - Diffusion

- In all 3 cases, the image was generated **globally**, i.e., all pixels of the image were sampled at once.

Autoregressive image generation

- An alternative approach is to generate an image **piece-by-piece** in an **autoregressive** manner.

- This is enabled by factoring the joint probability:

$$
\begin{aligned}
p(x_1, \ldots, x_m) &= p(x_1)p(x_2 \mid x_1)p(x_3 \mid x_1, x_2) \ldots p(x_m \mid x_1, \ldots, x_{m-1}) \\
&= \prod_{i=1}^{m} p(x_i \mid x_1, \ldots, x_{i-1})
\end{aligned}
$$

Autoregressive image generation

- In homework 1, you used hard-coded conditional probability tables $p(x_1), p(x_2 \mid x_1), p(x_3 \mid x_1, x_2)\ldots$ to sample images from $p(\mathbf{x})$.

- While conceptually simple, this approach is impractical:

    - Inefficient to compute & store (exponential costs)

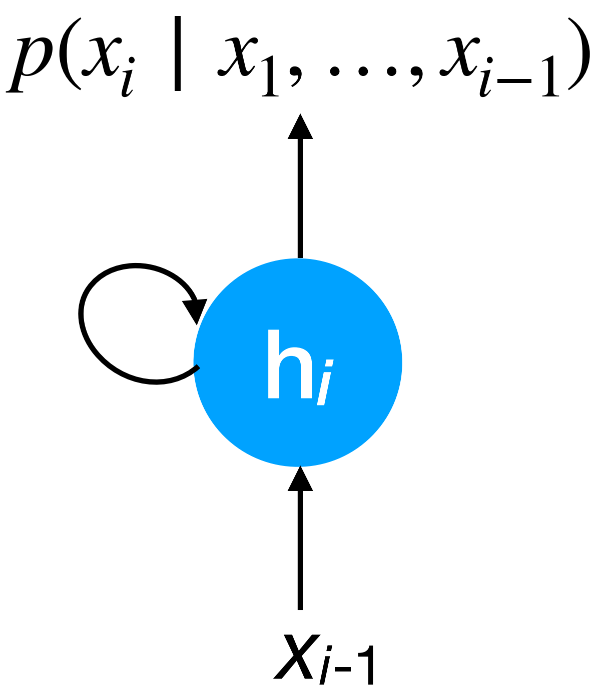
    - No ability to generalize beyond the dataset itself

Autoregressive image generation

- A more promising approach is to train a parametric model that can *approximate* each conditional distribution.

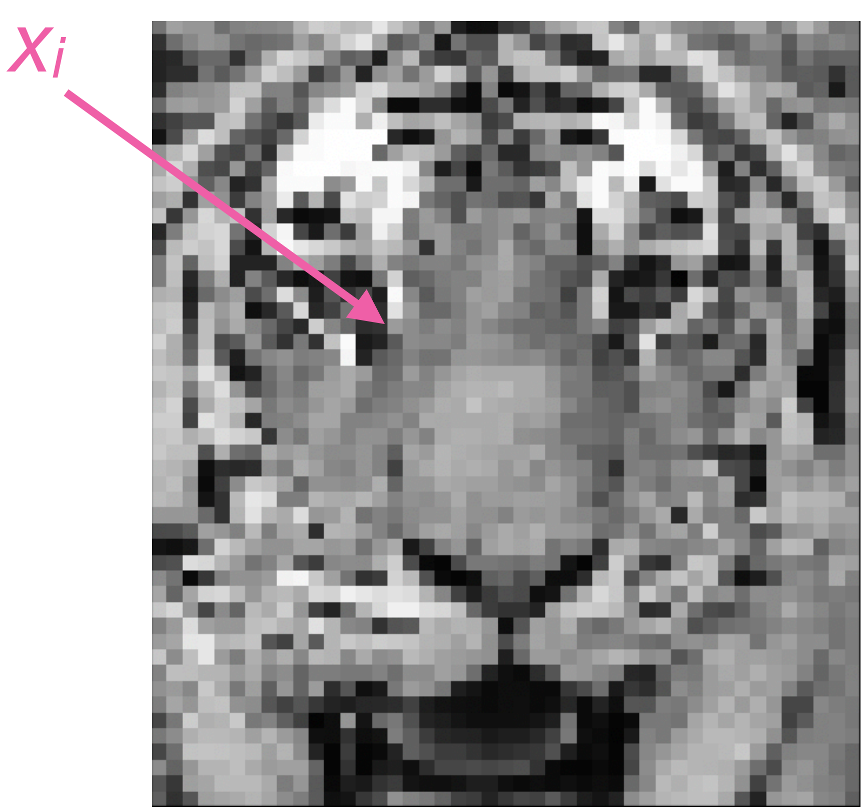
- One intuitive architecture is a *single* RNN (with parameters θ) that takes x_{i-1} (or a start symbol) as input and produces $p(x_i \mid x_1, \ldots, x_{i-1})$ as output at each timestep *i*.

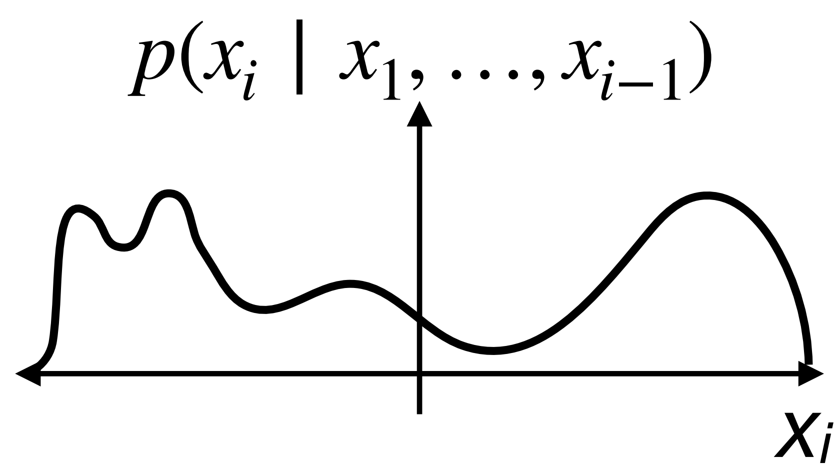
- We then sample $x_i \sim p(x_i \mid x_1, \ldots, x_{i-1})$ and iterate.

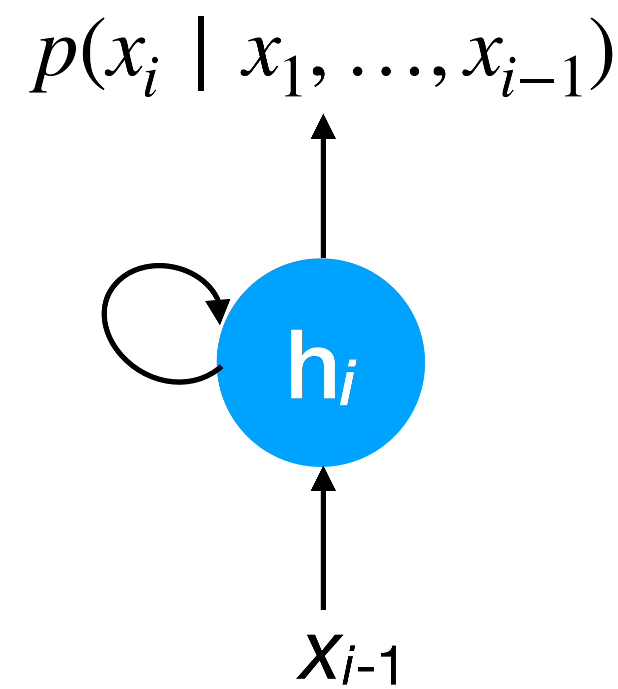


Representing $p(x_i \mid x_1, \ldots, x_{i-1})$

- Images are fundamentally *continuous* signals ($i \mapsto x_i \in \mathbb{R}$), but for autoregression, it's much easier to model *discrete* probability distributions.



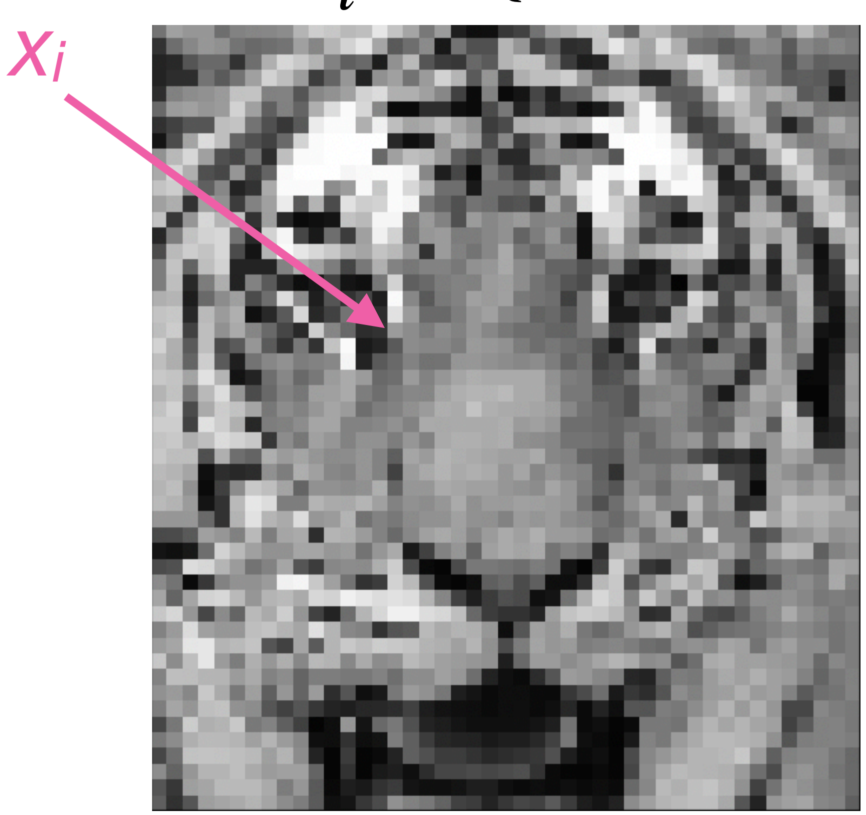


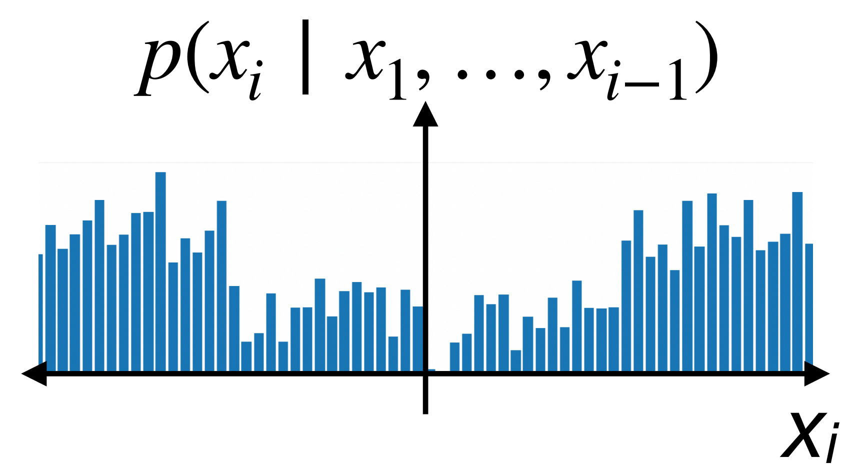


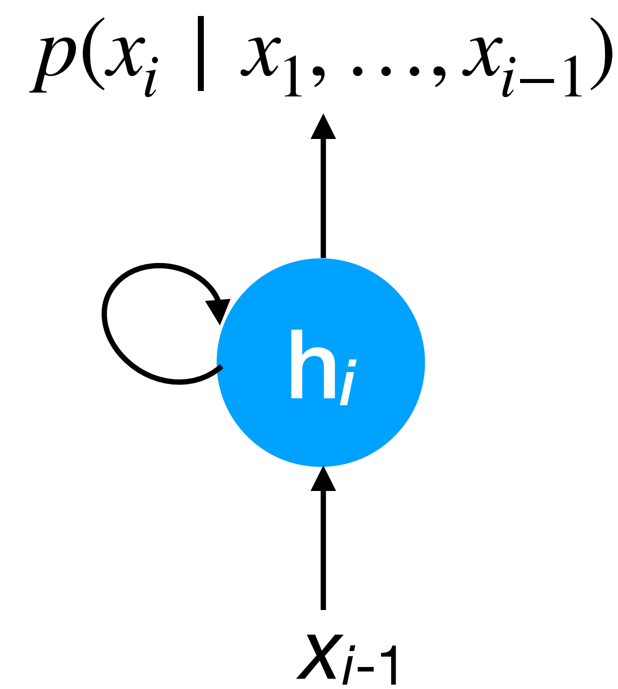
Representing $p(x_i \mid x_1, \ldots, x_{i-1})$

- Images are fundamentally *continuous* signals ($i \mapsto x_i \in \mathbb{R}$), but for autoregression, it's much easier to model *discrete* probability distributions.

- Hence, we typically quantize the pixel intensity as $x_i \in \{0,\ldots,255\}$.





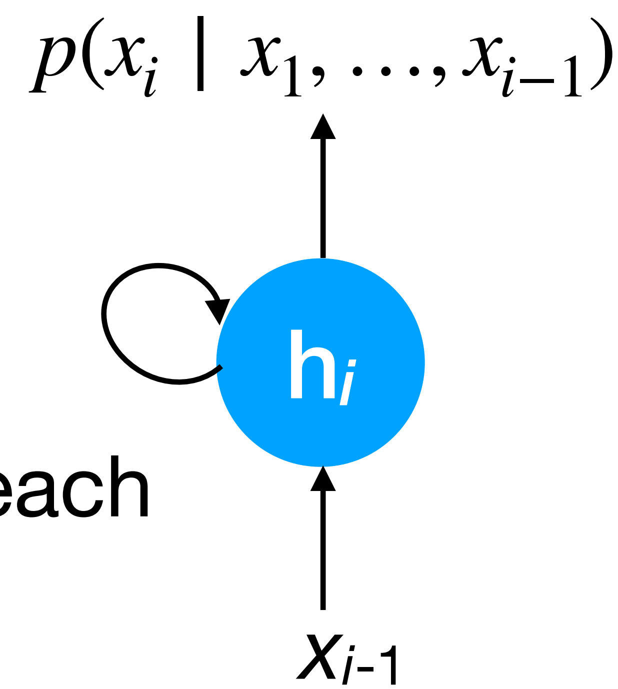


Representing $p(x_i \mid x_1, \ldots, x_{i-1})$

- Images are fundamentally *continuous* signals ($i \mapsto x_i \in \mathbb{R}$), but for autoregression, it's much easier to model *discrete* probability distributions.

- Hence, we typically quantize the pixel intensity as $x_i \in \{0,\ldots,255\}$.

- For RGB images with *m* pixels, we can model each $\mathbf{x} = (x_1, \ldots, x_{3m}) \in \{0,\ldots,255\}^{3m}$.

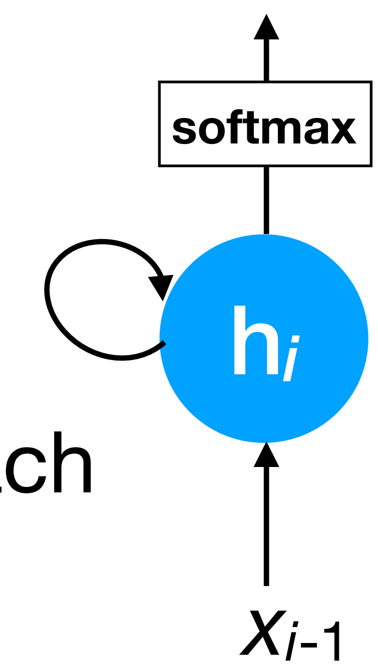


Representing $p(x_i \mid x_1, \ldots, x_{i-1})$

- Images are fundamentally *continuous* signals ($i \mapsto x_i \in \mathbb{R}$), but for autoregression, it's much easier to model *discrete* probability distributions.

$p(x_i \mid x_1, \ldots, x_{i-1})$

0	1	2	…	255
0.02	0.17	0.23	…	0.01



- Hence, we typically quantize the pixel intensity as $x_i \in \{0,\ldots,255\}$.

- For RGB images with *m* pixels, we can model each $\mathbf{x} = (x_1, \ldots, x_{3m}) \in \{0,\ldots,255\}^{3m}$.

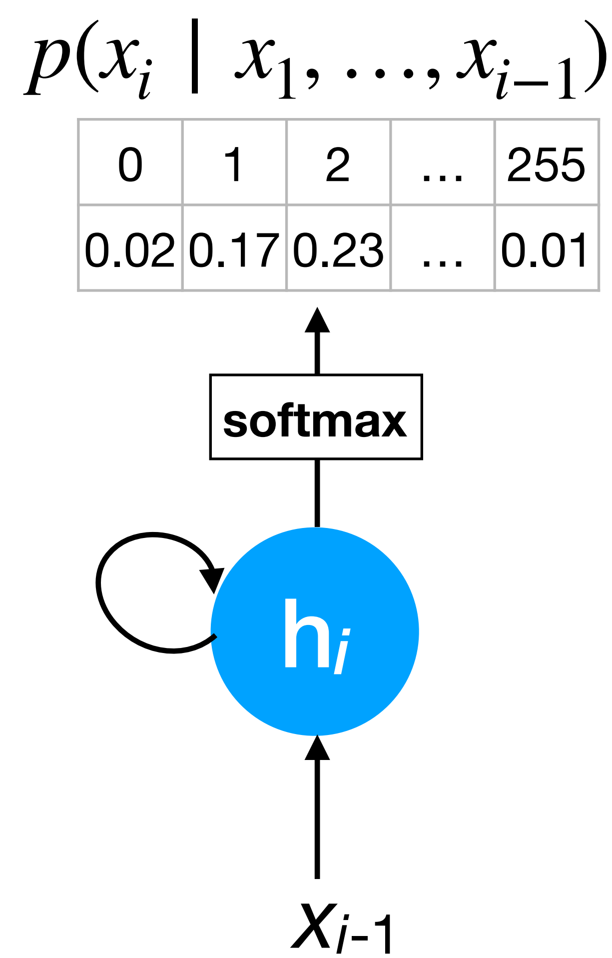
- Then we can just use a softmax with 256 outputs to represent $p(x_i \mid x_1, \ldots, x_{i-1})$.

RNN: training

- Given an image $\mathbf{x} = (x_1, \ldots, x_m)$, we can use MLE to train the RNN (with parameters θ):

$$p_\theta(\mathbf{x}) = \prod_{i=1}^{m} p_\theta(x_i \mid x_1, \ldots, x_{i-1})$$

i.e., what probability does the model assign this sequence of pixel values?



RNN: training

- Given an image $\mathbf{x} = (x_1, \ldots, x_m)$, we can use MLE to train the RNN (with parameters θ):

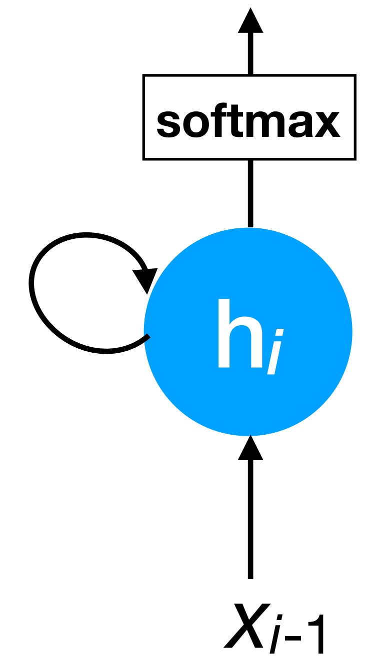
$$p_\theta(\mathbf{x}) = \prod_{i=1}^{m} p_\theta(x_i \mid x_1, \ldots, x_{i-1})$$

i.e., what probability does the model assign this sequence of pixel values?

- It turns out that maximizing this likelihood = minimizing the sum of cross-entropies…

$p(x_i \mid x_1, \ldots, x_{i-1})$

0	1	2	…	255
0.02	0.17	0.23	…	0.01



Log-loss is just MLE for binary classification

- Consider a NN (with parameters θ) for binary classification of **x** whose output $\hat{y} \in (0,1)$ estimates $p(Y=1 \mid \mathbf{x})$.

Log-loss is just MLE for binary classification

- Consider a NN (with parameters *θ*) for binary classification of **x** whose output $\hat{y} \in (0,1)$ estimates *p*(*Y*=1 | **x**).

- Given a labeled training example (**x**,*y*), we can ask: “how likely does our model think label *y* is for the input **x**?”
  - $p(y \mid \mathbf{x};\theta) = \hat{y}$ if $y = 1$
  - $p(y \mid \mathbf{x};\theta) = 1 - \hat{y}$ if $y = 0$

Log-loss is just MLE for binary classification

- Consider a NN (with parameters *θ*) for binary classification of **x** whose output $\hat{y} \in (0,1)$ estimates *p*(*Y*=1 | **x**).

- Given a labeled training example (**x**,*y*), we can ask: "how likely does our model think label *y* is for the input **x**?"
  - $p(y \mid \mathbf{x}; \theta) = \hat{y}$ if $y = 1$
  - $p(y \mid \mathbf{x}; \theta) = 1 - \hat{y}$ if $y = 0$

- Hence:
  - $p(y \mid \mathbf{x}; \theta) = \hat{y}^y (1 - \hat{y})^{1-y}$

Log-loss is just MLE for binary classification

- Now, for a single training example (**x**,*y*), where $y \in \{0,1\}$, how can we find the "best" θ?

$$p(y \mid \mathbf{x}; \theta) = \hat{y}^y (1 - \hat{y})^{1-y}$$

Log-loss is just MLE for binary classification

- Now, for a single training example (**x**,*y*), where $y \in \{0,1\}$, how can we find the "best" θ?

$$
\begin{aligned}
p(y \mid \mathbf{x}; \theta) &= \hat{y}^y (1-\hat{y})^{1-y} \\
\log p(y \mid \mathbf{x}; \theta) &= \log \hat{y}^y (1-\hat{y})^{1-y} \\
&= y \log \hat{y} + (1-y) \log(1-\hat{y}) \\
&= -f_{\log}(\theta; \hat{y}, y)
\end{aligned}
$$

Log-loss is just MLE for binary classification

- Now, for a single training example (**x**,*y*), where $y \in \{0,1\}$, how can we find the "best" θ?

$$
\begin{aligned}
p(y \mid \mathbf{x}; \theta) &= \hat{y}^y (1-\hat{y})^{1-y} \\
\log p(y \mid \mathbf{x}; \theta) &= \log \hat{y}^y (1-\hat{y})^{1-y} \\
&= y \log \hat{y} + (1-y) \log(1-\hat{y}) \\
&= -f_{\log}(\theta; \hat{y}, y)
\end{aligned}
$$

- In other words, the MLE for θ is the argmin of $f_{\log}$.

Cross-entropy is just MLE for multi-class classification

- Now consider a NN (with parameters *θ*) for multi-class classification of **x** whose output $\hat{y}_k$ estimates *p*(*Y*=*k* | **x**).

- Given a labeled training example (**x**,**y**), we can ask: "how likely does our model think label *y* is for the input **x**?"

- $p(\mathbf{y} \mid \mathbf{x}; \theta) = \prod_{k=1}^{K} \hat{y}_k^{y_k}$

Cross-entropy is just MLE for multi-class classification

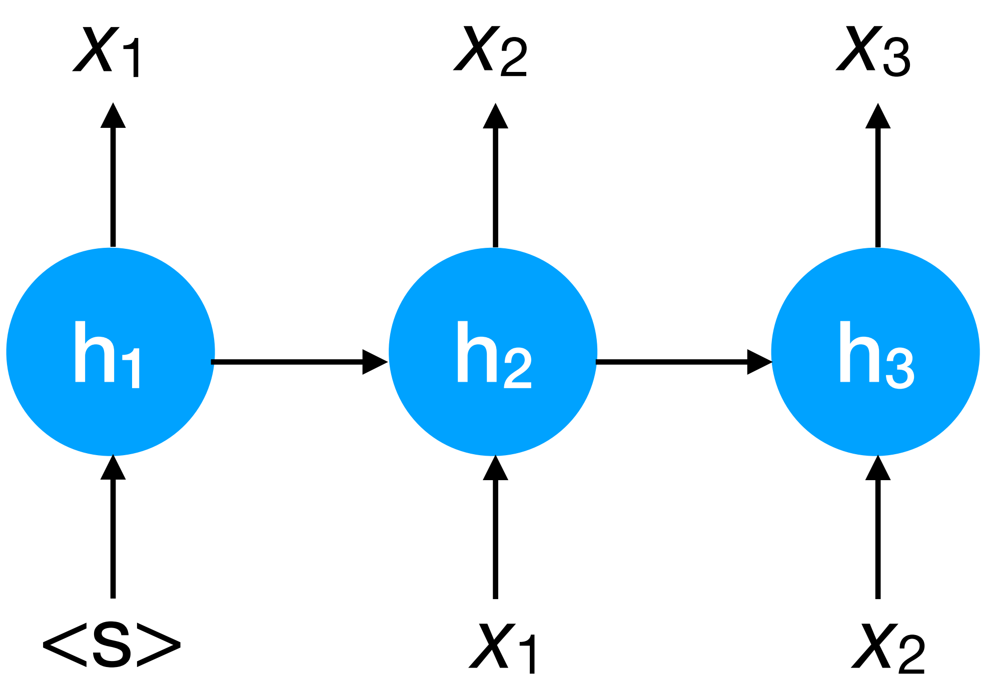
- Similarly, for $y \in \{1, \dots, K\}$, the "best" θ can be found as:

$$\begin{aligned} p(\mathbf{y} \mid \mathbf{x}; \theta) &= \prod_{k=1}^{K} \hat{y}_k^{y_k} \\ \log p(\mathbf{y} \mid \mathbf{x}; \theta) &= \log \prod_{k=1}^{K} \hat{y}_k^{y_k} \\ &= \sum_{k=1}^{K} y_k \log \hat{y}_k \\ &= -f_{\mathrm{CE}}(\theta; \hat{\mathbf{y}}, \mathbf{y}) \end{aligned}$$

- In other words, the MLE for θ is the argmin of f_{CE}.

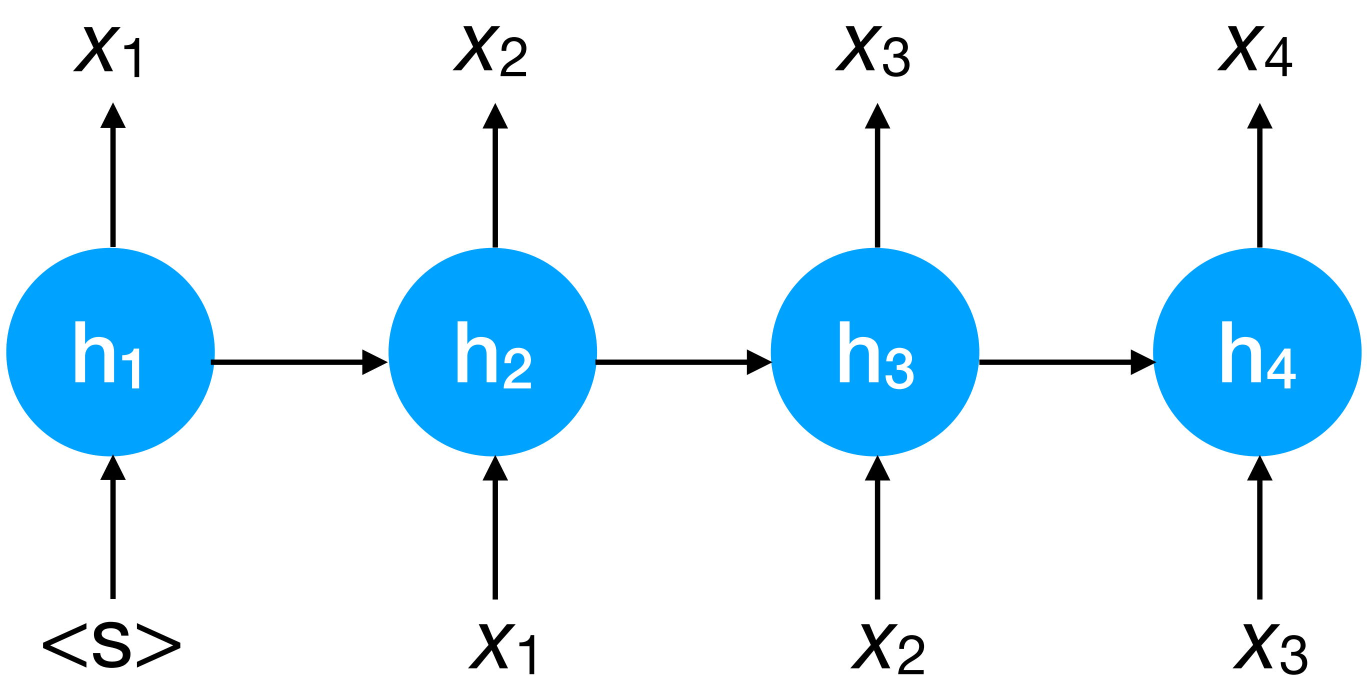
RNNs: time cost

- **Inference**: $O(1)$ per timestep i since $\mathbf{h}_{i\text{-}1}$ is already computed



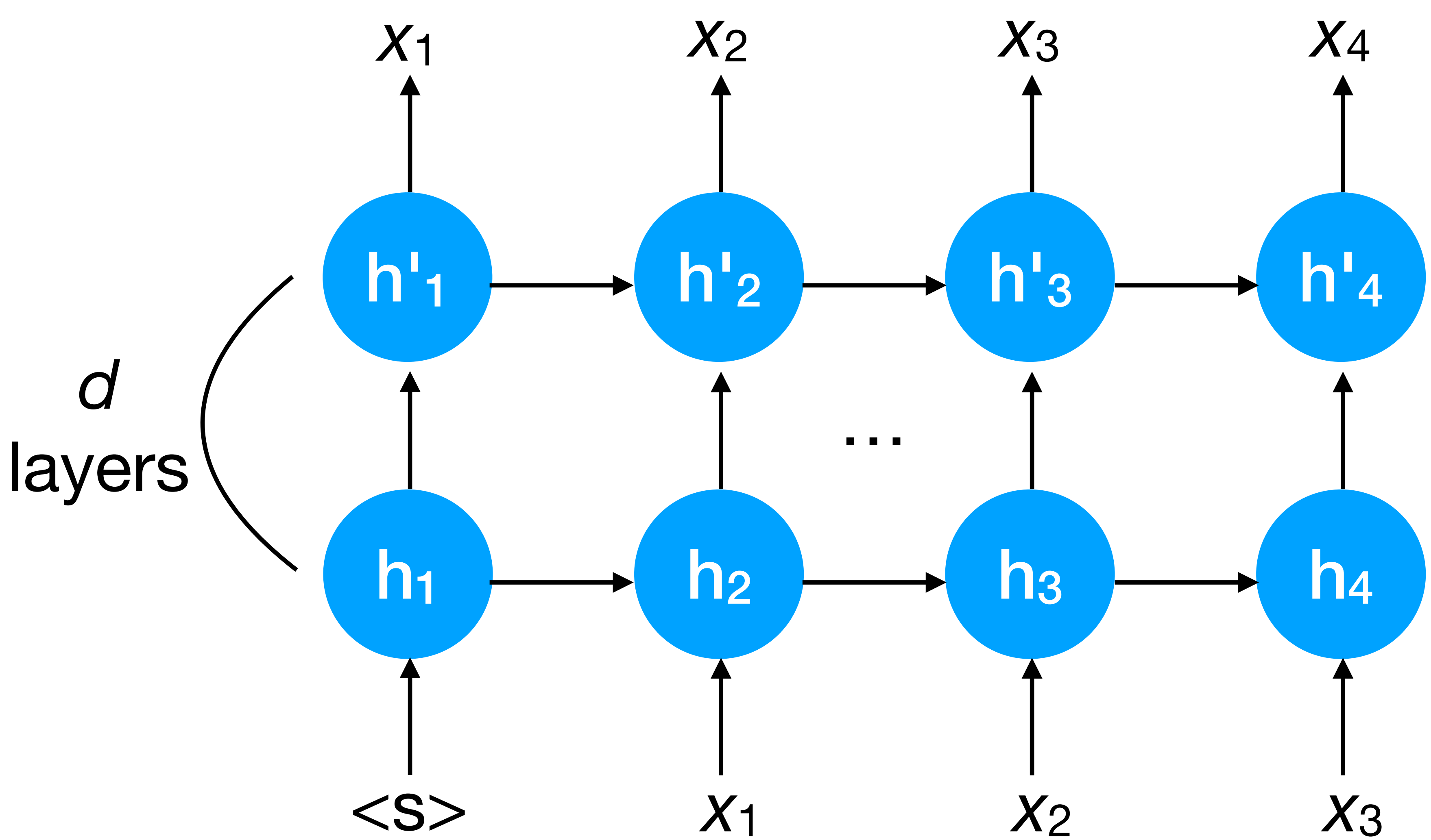
RNNs: time cost

- **Inference**: $O(1)$ per timestep i since $\mathbf{h}_{i\text{-}1}$ is already computed



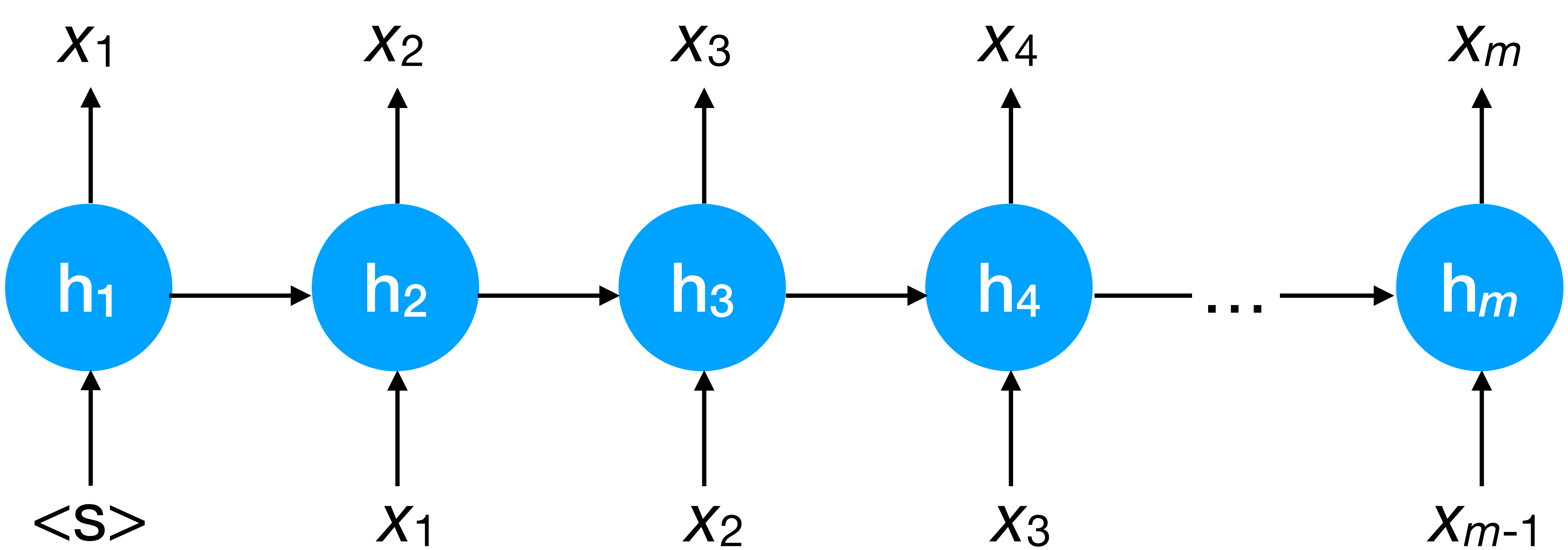
RNNs: time cost

- **Inference**: $O(1)$ per timestep i since $\mathbf{h}_{i\text{-}1}$ is already computed, or $O(d)$ for a multi-layer RNN.



RNNs: time cost

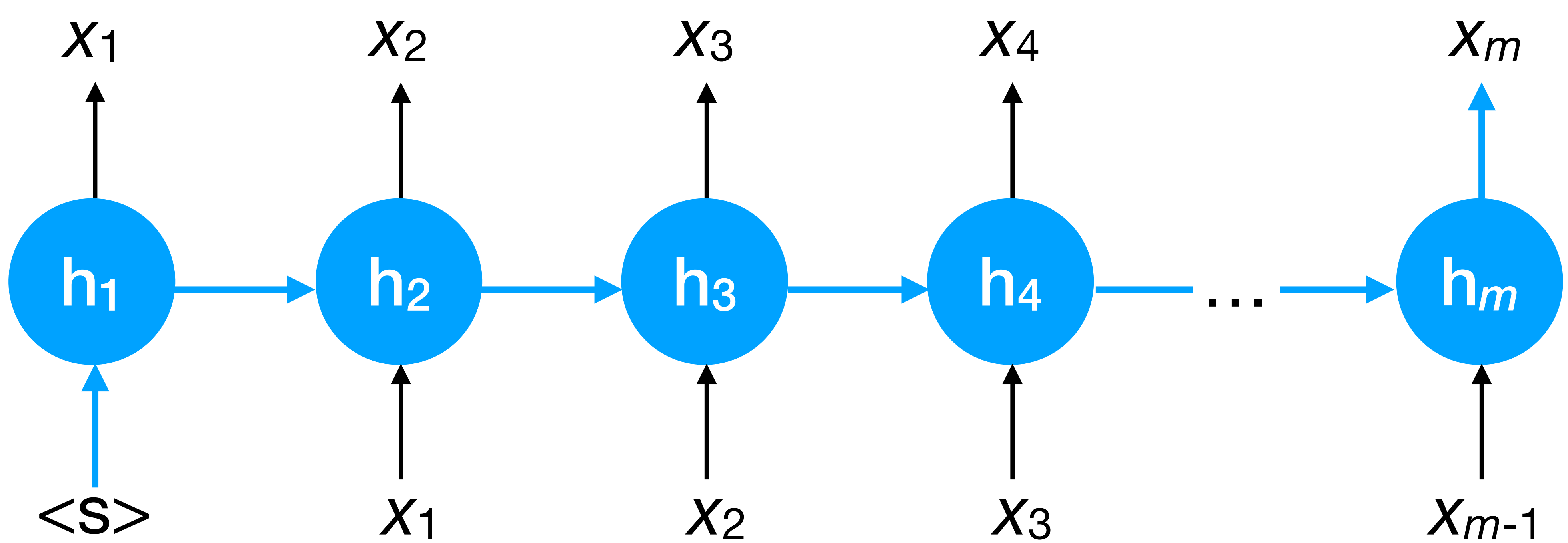
- **Inference**: $O(1)$ per timestep i since $\mathbf{h}_{i\text{-}1}$ is already computed, or $O(d)$ for a multi-layer RNN.
- **Training**: $O(d{*}m)$ total work for m pixels



RNNs: time cost

- **Inference**: $O(1)$ per timestep i since $\mathbf{h}_{i\text{-}1}$ is already computed, or $O(d)$ for a multi-layer RNN.

- **Training**: $O(d{*}m)$ total work for m pixels, but generally not parallelizable since the **span** (longest sequence of dependent computations) is $O(d{+}m)$.



RNNs: a suboptimal fit

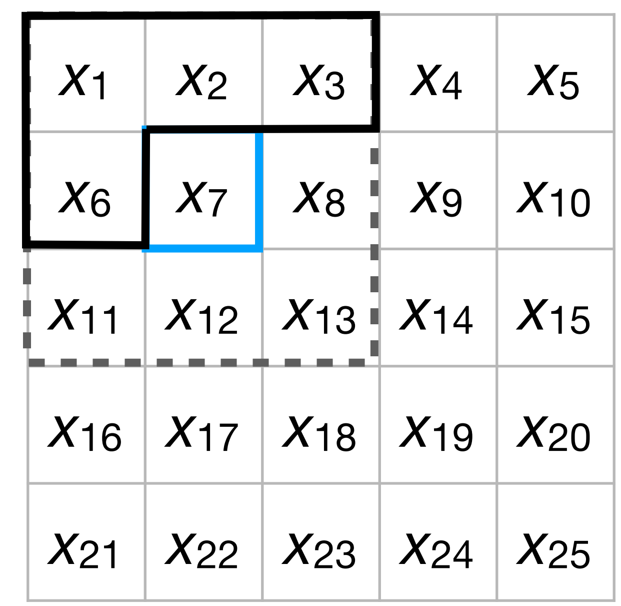
- Also, RNNs do not respect the 2-d structure of images, e.g., local pixel correlations, translation invariance.

- Consider: an RNN to model p(**x**) for a $w \times h$ image must learn to discontinuously "jump" from $p(x_w \mid x_1, \ldots, x_{w-1})$ to $p(x_{w+1} \mid x_1, \ldots, x_w)$.

- Is there an autoregressive architecture that is better suited?

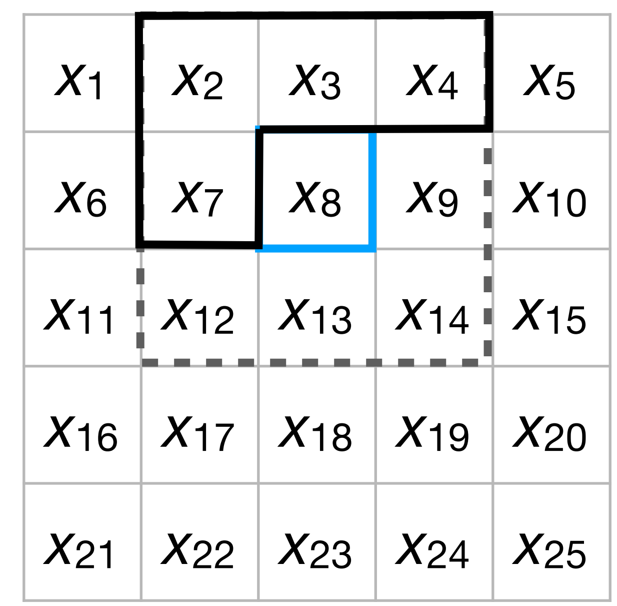
PixelCNN
(van den Oord et al. 2016)

- PixelCNN harnesses **masked convolution** to enforce translation-invariant features in the autoregression: pixel x_i is predicted from neighboring pixels in RF(i) using a fixed convolution filter.
  - RF(i) is the **receptive field** of pixel i.
  - The mask prevents information leakage from future timesteps.



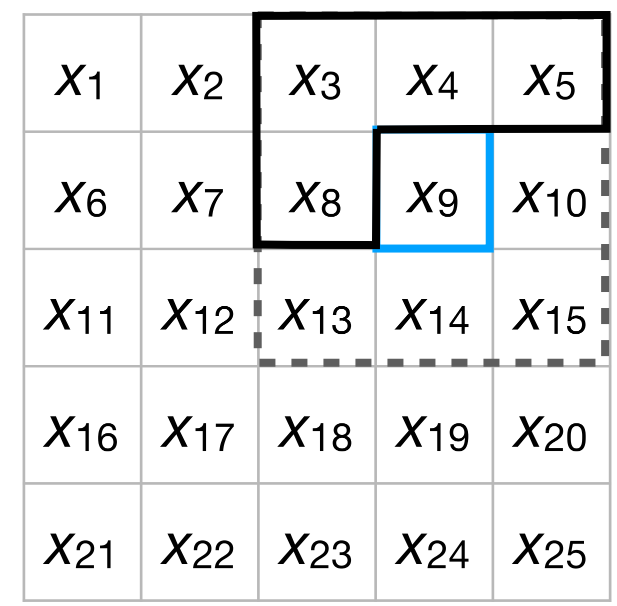
PixelCNN
([van den Oord et al. 2016](#))

- PixelCNN harnesses **masked convolution** to enforce translation-invariant features in the autoregression: pixel x_i is predicted from neighboring pixels in RF(i) using a fixed convolution filter.
  - RF(i) is the **receptive field** of pixel i.
  - The mask prevents information leakage from future timesteps.



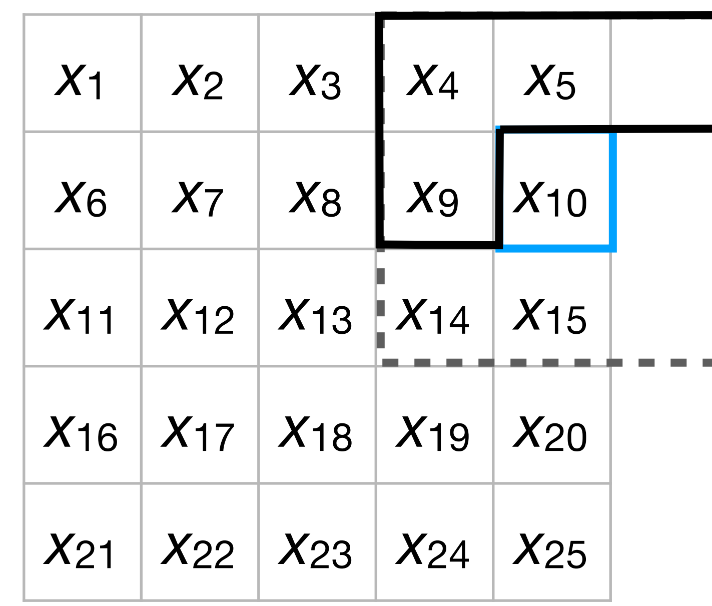
PixelCNN
(van den Oord et al. 2016)

- PixelCNN harnesses **masked convolution** to enforce translation-invariant features in the autoregression: pixel x_i is predicted from neighboring pixels in RF(i) using a fixed convolution filter.
    - RF(i) is the **receptive field** of pixel i.
    - The mask prevents information leakage from future timesteps.



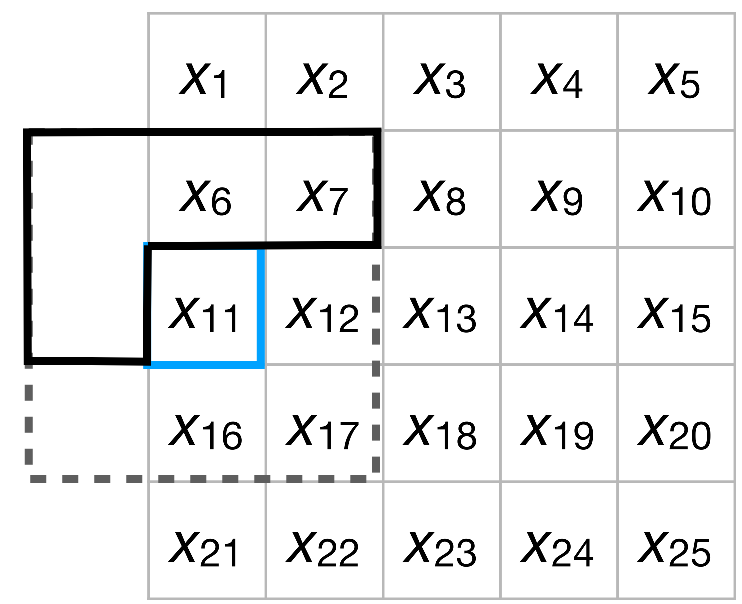
PixelCNN
(van den Oord et al. 2016)

- PixelCNN harnesses **masked convolution** to enforce translation-invariant features in the autoregression: pixel x_i is predicted from neighboring pixels in RF(*i*) using a fixed convolution filter.
  - RF(*i*) is the **receptive field** of pixel *i*.
  - The mask prevents information leakage from future timesteps.



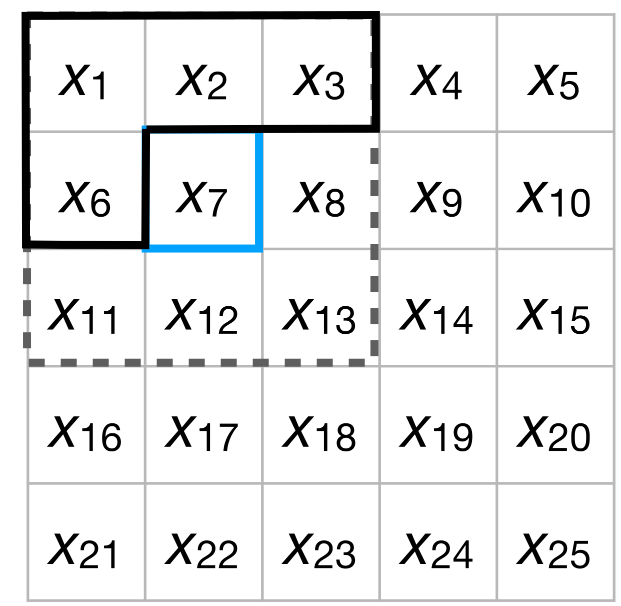
PixelCNN (van den Oord et al. 2016)

- PixelCNN harnesses **masked convolution** to enforce translation-invariant features in the autoregression: pixel x_i is predicted from neighboring pixels in RF(i) using a fixed convolution filter.
  - RF(i) is the **receptive field** of pixel i.
  - The mask prevents information leakage from future timesteps.



PixelCNN (van den Oord et al. 2016)

- PixelCNN harnesses **masked convolution** to enforce translation-invariant features in the autoregression: pixel x_i is predicted from neighboring pixels in RF(*i*) using a fixed convolution filter.
  - RF(*i*) is the **receptive field** of pixel *i*.
  - The mask prevents information leakage from future timesteps.

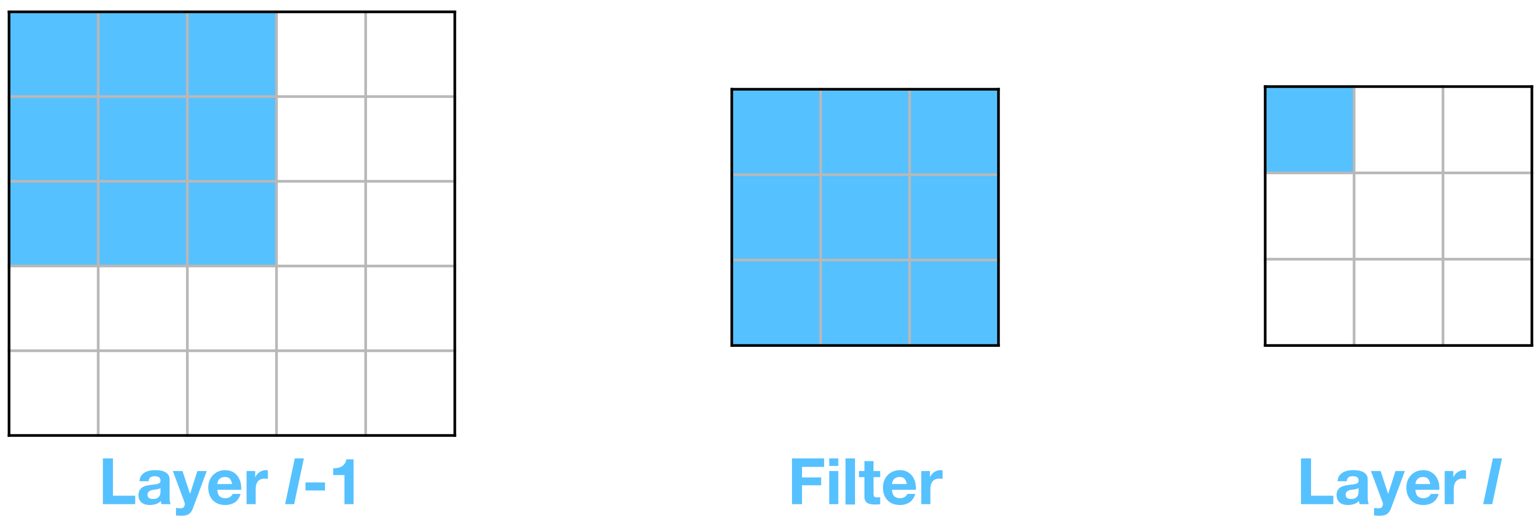


- PixelCNN approximates $p(x_i \mid x_1, \ldots, x_{i-1})$ as $p(x_i \mid \{x_j\}_{j \in \mathrm{RF}(i)})$

$$p(x_7 \mid x_1, \ldots, x_6) \approx p(x_7 \mid x_1, x_2, x_3, x_6)$$

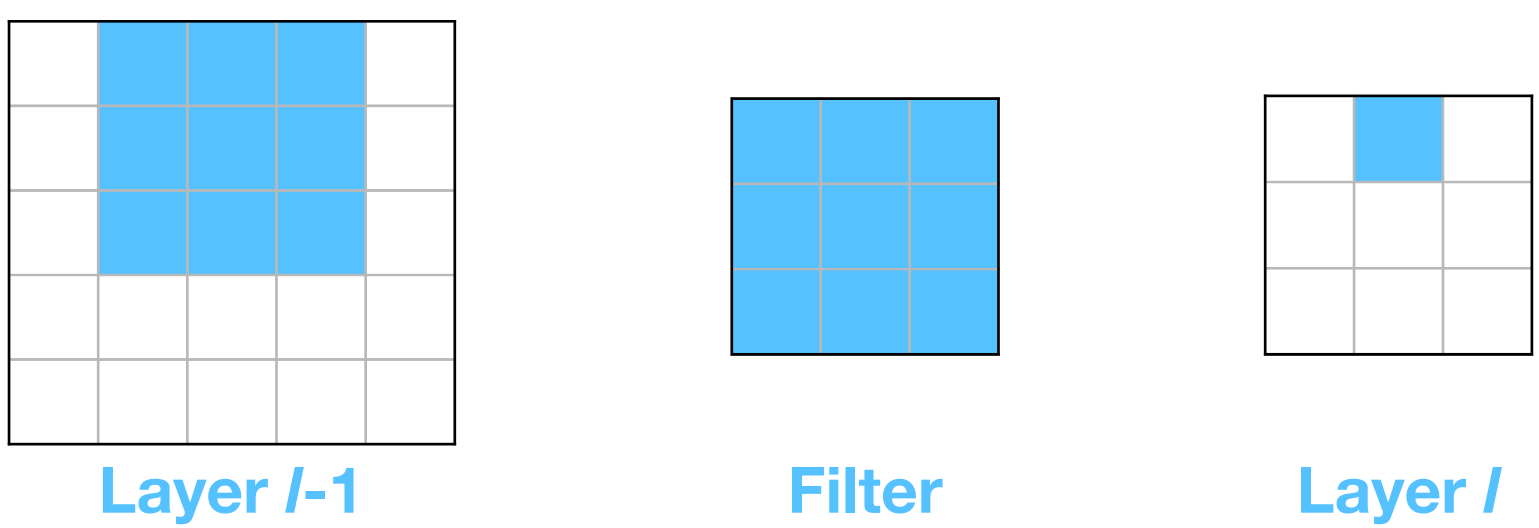
Receptive fields

- Each neuron *i* in a convolutional layer *l* depends only on neurons in a local region around *i* in the previous layer *l*-1.



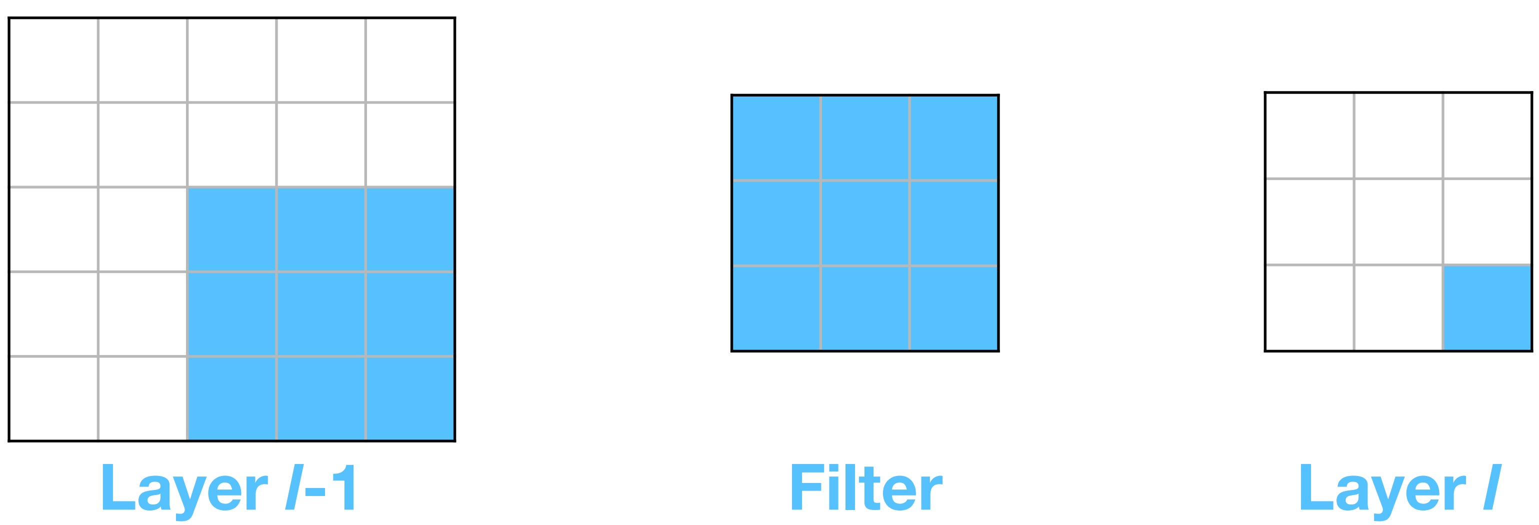
Receptive fields

- Each neuron *i* in a convolutional layer *l* depends only on neurons in a local region around *i* in the previous layer *l*-1.



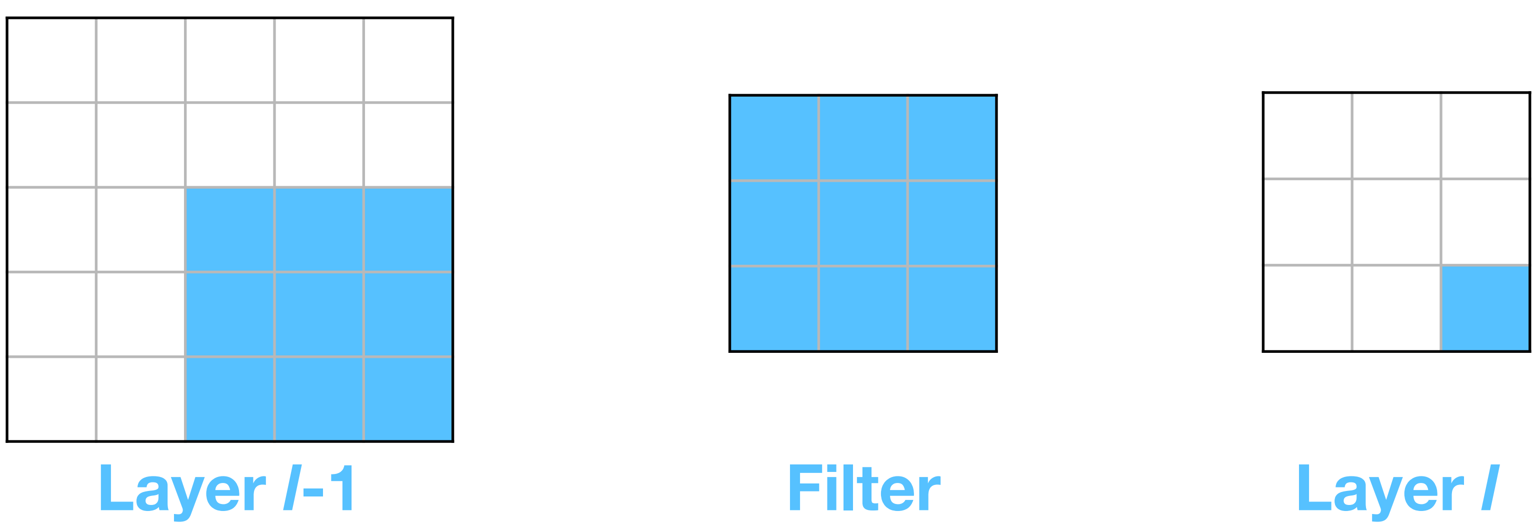
Receptive fields

- Each neuron *i* in a convolutional layer *l* depends only on neurons in a local region around *i* in the previous layer *l*-1.



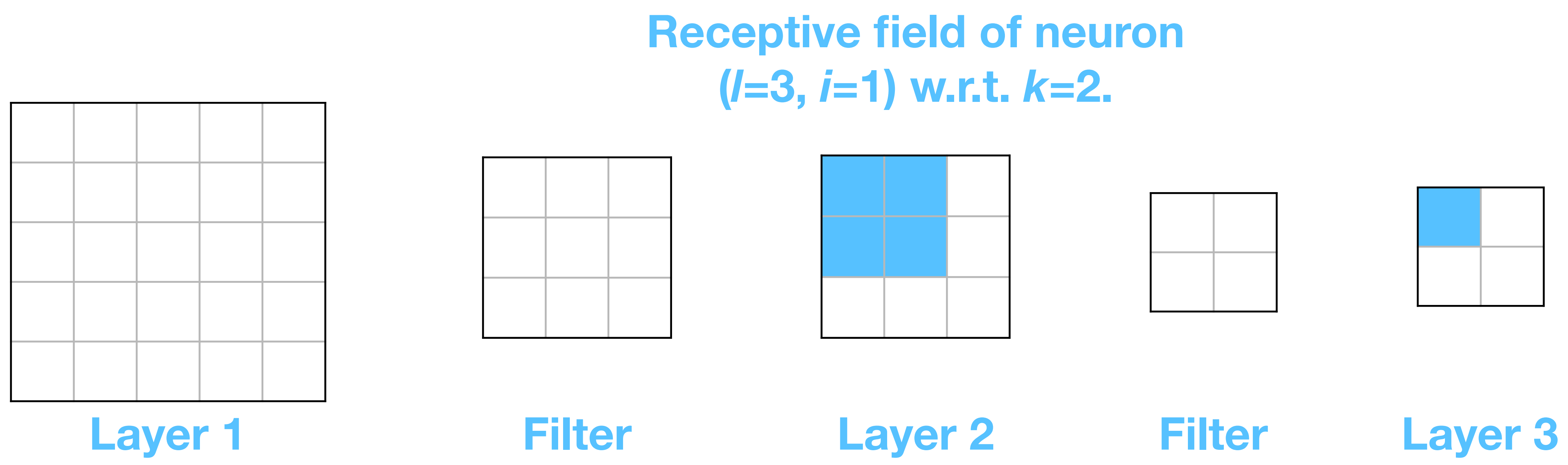
Receptive fields

- In general, we define the **receptive field** of neuron *i* in layer *l* w.r.t. layer *k* as the set of neurons in *k* that influence *i* in *l*.



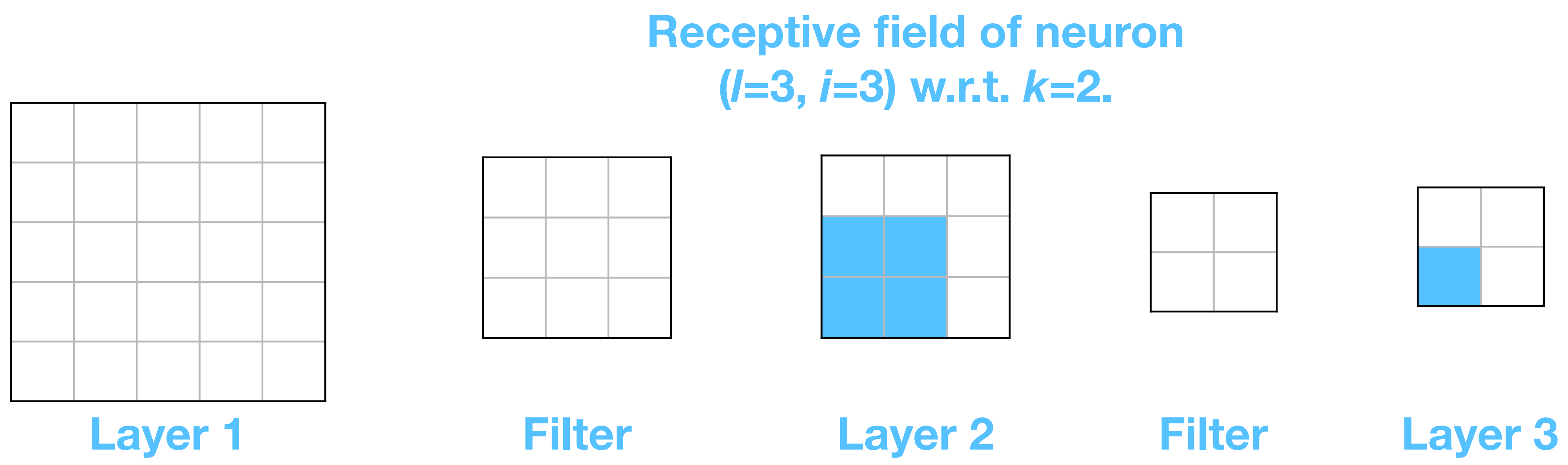
Receptive fields

- In general, we define the **receptive field** of neuron *i* in layer *l* w.r.t. layer *k* as the set of neurons in *k* that influence *i* in *l*.



Receptive fields

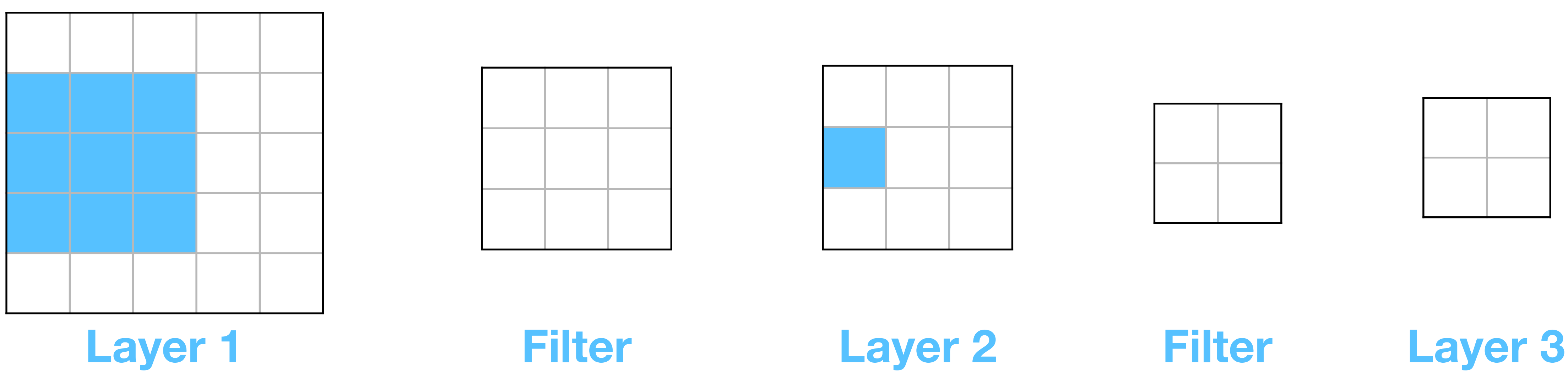
- In general, we define the **receptive field** of neuron *i* in layer *l* w.r.t. layer *k* as the set of neurons in *k* that influence *i* in *l*.



Receptive fields

- In general, we define the **receptive field** of neuron *i* in layer *l* w.r.t. layer *k* as the set of neurons in *k* that influence *i* in *l*.

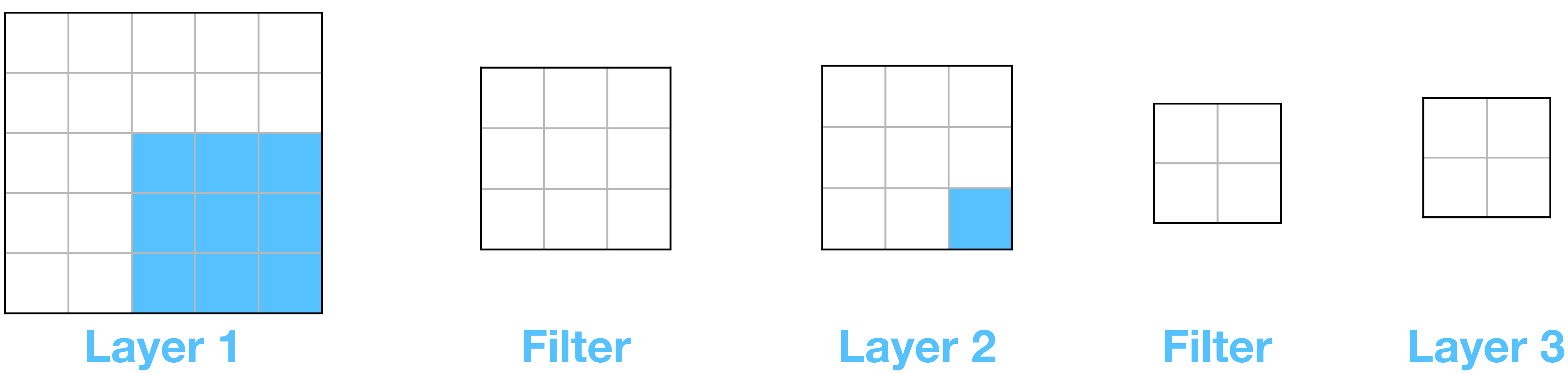
Receptive field of neuron (l=2, i=4) w.r.t. k=1.



Receptive fields

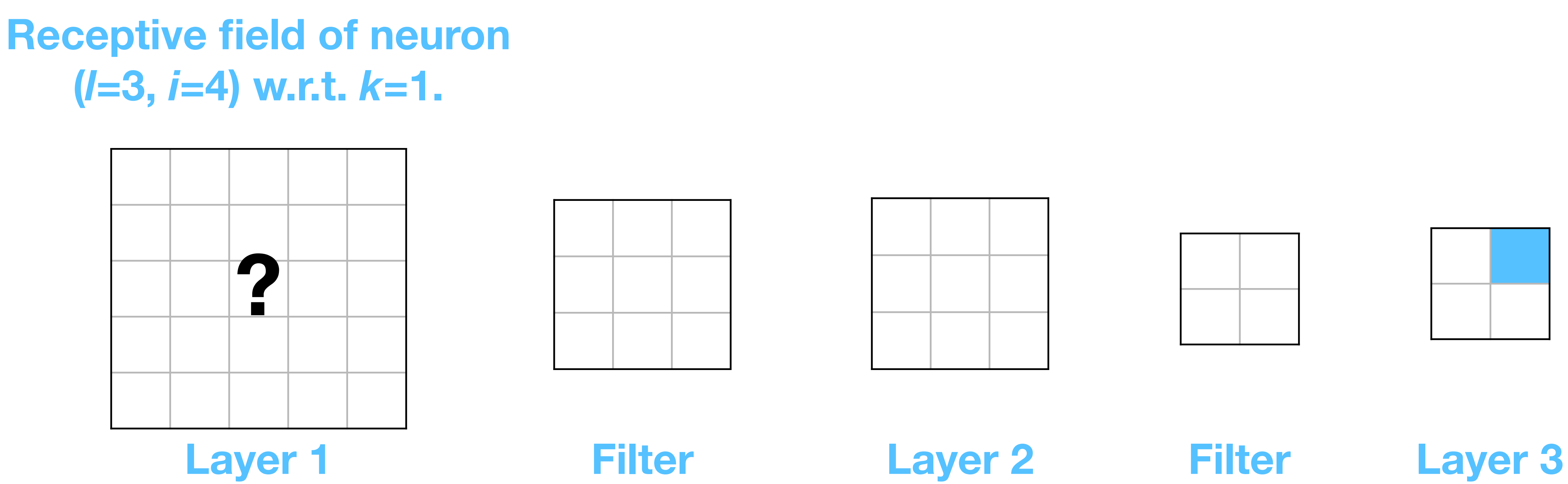
- In general, we define the **receptive field** of neuron *i* in layer *l* w.r.t. layer *k* as the set of neurons in *k* that influence *i* in *l*.

Receptive field of neuron (l=2, i=9) w.r.t. k=1.



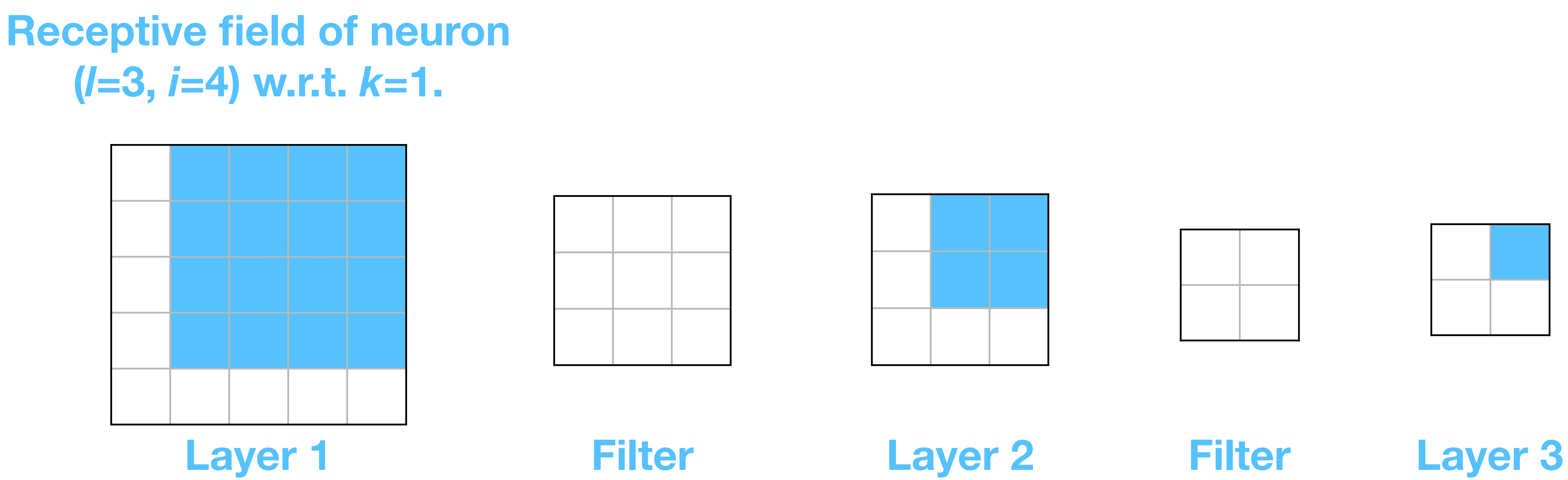
Exercise

- What is the receptive field of neuron (l=3, i=2) w.r.t. k=1?



Solution

- What is the receptive field of neuron (l=3, i=2) w.r.t. k=1?



PixelCNN (van den Oord et al. 2016)

- PixelCNN harnesses **masked convolution** to enforce translation-invariant features in the autoregression: pixel x_i is predicted from neighboring pixels in RF(*i*) using a fixed convolution filter.
  - RF(*i*) is the **receptive field** of pixel *i*.
  - The mask prevents information leakage from future timesteps.

x_1	x_2	x_3	x_4	x_5
x_6	x_7	x_8	x_9	x_{10}
x_{11}	x_{12}	x_{13}	x_{14}	x_{15}
x_{16}	x_{17}	x_{18}	x_{19}	x_{20}
x_{21}	x_{22}	x_{23}	x_{24}	x_{25}

- PixelCNN approximates $p(x_i \mid x_1, \ldots, x_{i-1})$ as $p(x_i \mid \{x_j\}_{j \in \mathrm{RF}(i)})$

- Through multiple convolution layers, the receptive field is enlarged, thus providing more global context.

PixelCNN
(van den Oord et al. 2016)

- PixelCNN harnesses **masked convolution** to enforce translation-invariant features in the autoregression: pixel x_i is predicted from neighboring pixels in RF(i) using a fixed convolution filter.
  - RF(i) is the **receptive field** of pixel i.
  - The mask prevents information leakage from future timesteps.

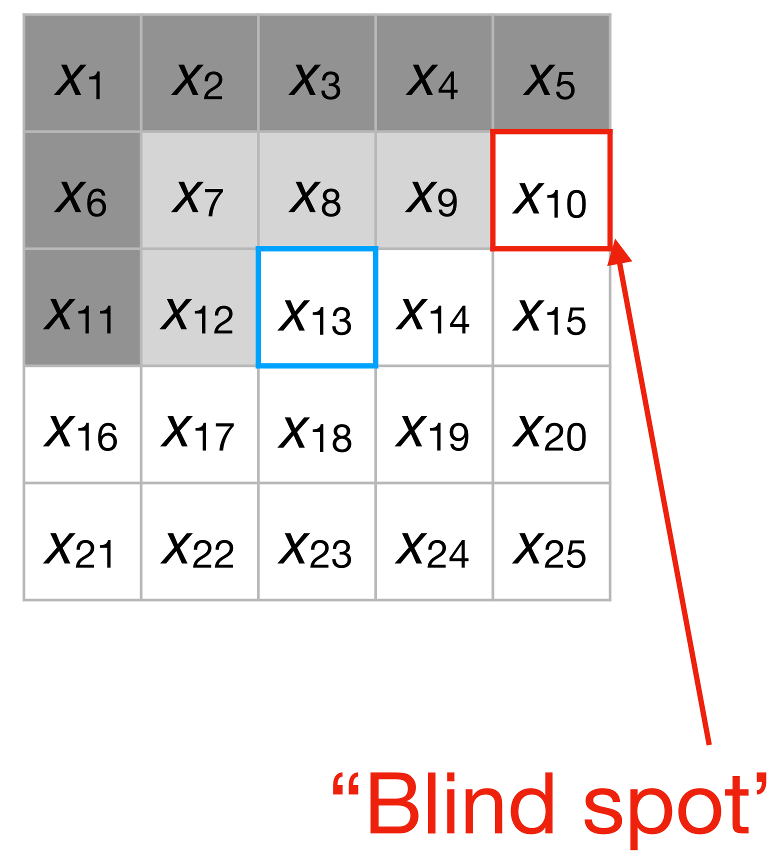
x_1	x_2	x_3	x_4	x_5
x_6	x_7	x_8	x_9	x_{10}
x_{11}	x_{12}	x_{13}	x_{14}	x_{15}
x_{16}	x_{17}	x_{18}	x_{19}	x_{20}
x_{21}	x_{22}	x_{23}	x_{24}	x_{25}

- PixelCNN approximates $p(x_i \mid x_1, \ldots, x_{i-1})$ as $p(x_i \mid \{x_j\}_{j \in \mathrm{RF}(i)})$

- Through multiple convolution layers, the receptive field is enlarged, thus providing more global context.

PixelCNN
(van den Oord et al. 2016)

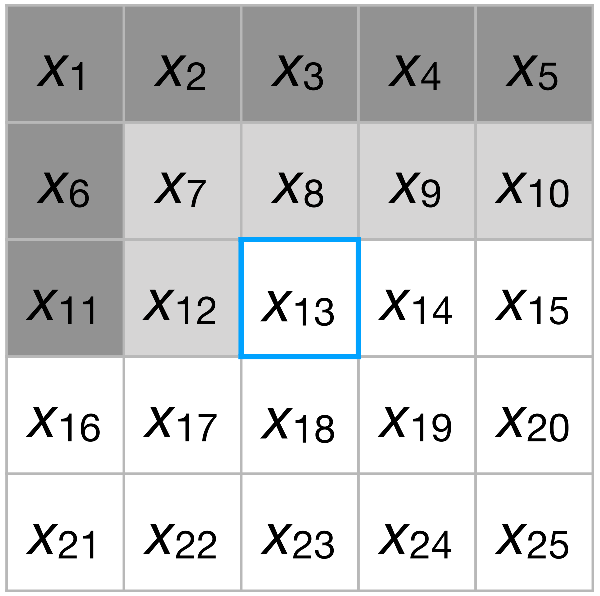
- The original PixelCNN method had “blind spots”.



PixelCNN (van den Oord et al. 2016)

- The original PixelCNN method had “blind spots”.

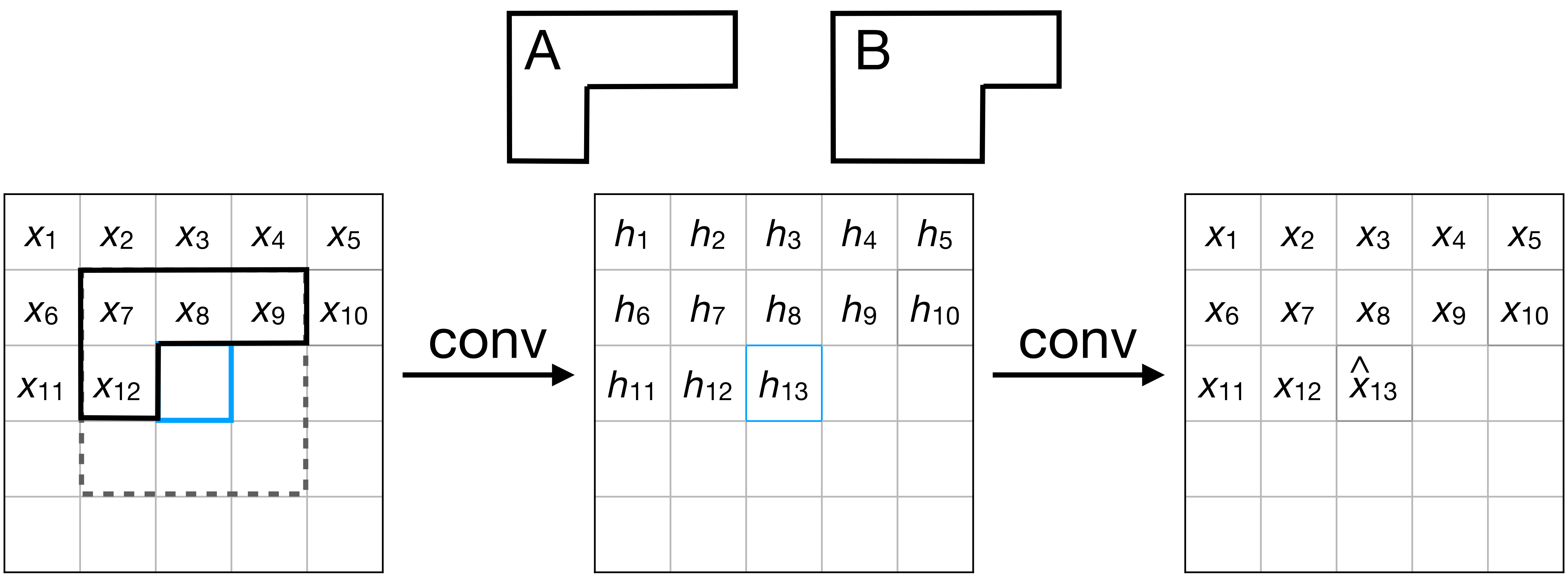
- In a modified PixelCNN, these were fixed using a more elaborate multi-stage autoregression process (unmasked information from previous row).



- With the modified scheme, $\text{RF}(i) \rightarrow \{x_1, \ldots, x_{i-1}\}$ as *d* grows.

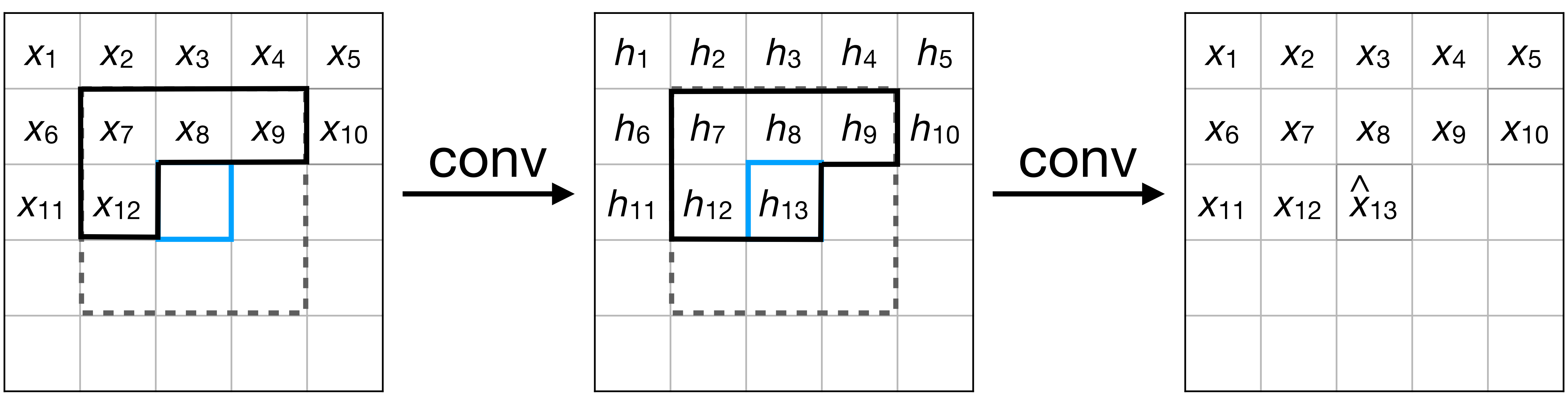
PixelCNN
(van den Oord et al. 2016)

- The causal mask is essential to prevent the CNN from depending on information that is not available at test time.
- In a multi-layer PixelCNN, which is the correct causal mask for layers 2+?



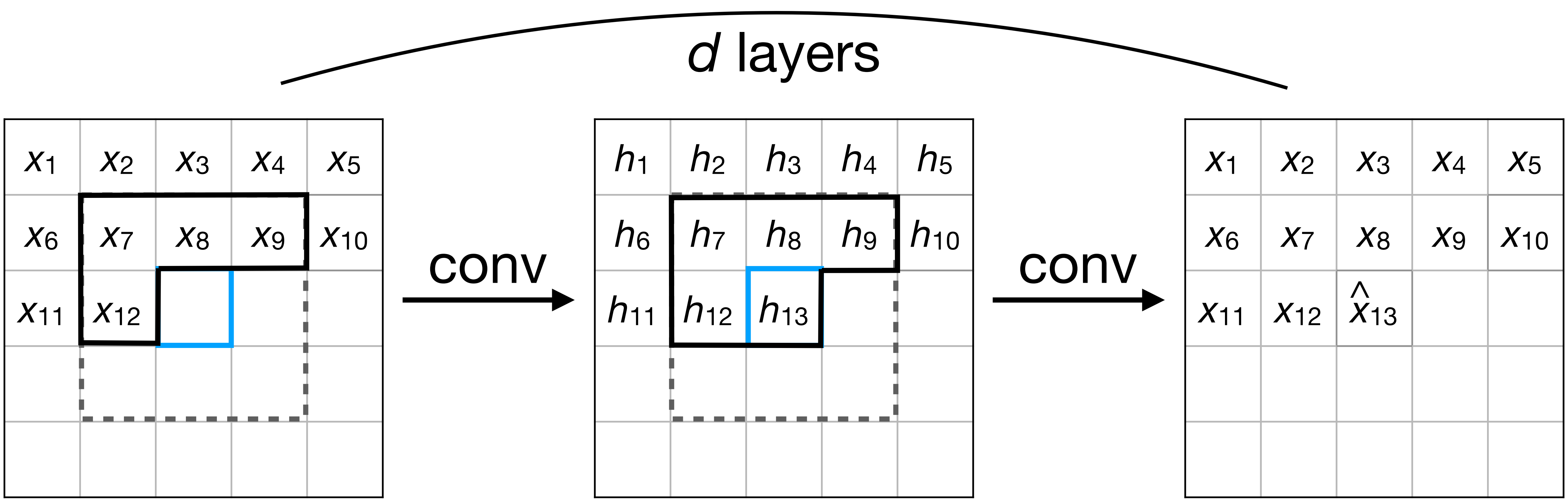
PixelCNN (van den Oord et al. 2016)

- The causal mask is essential to prevent the CNN from depending on information that is not available at test time.
- The first convolutional layer must exclude x_i, but subsequent layers can (and should) include h_i.
- This is harmless because h_i was computed based only on pixels $\{ x_{j<i} \}$.



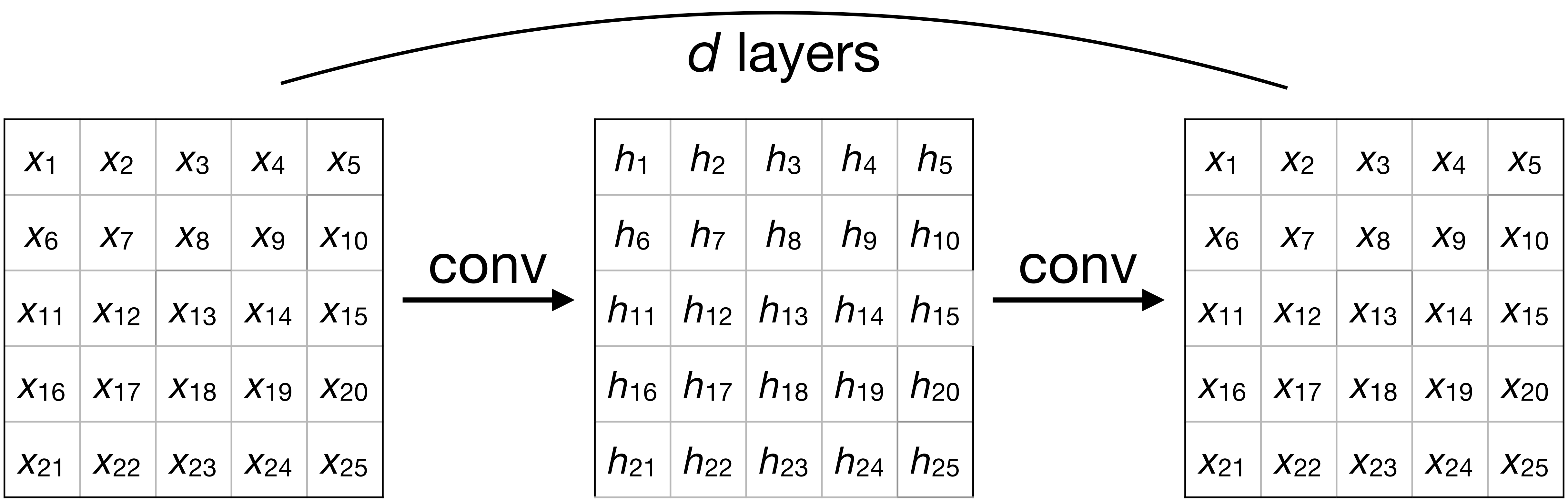
PixelCNN: time costs

- **Inference**: $O(d)$ per pixel, where d is number of layers.



PixelCNN: time costs

- **Inference**: $O(d)$ per pixel, where d is number of layers.
- **Training**: $O(d*m)$ total work, which can be parallelized over pixel locations since the span is only d.



PixelCNN: conditional generation

- PixelCNN can also be used for conditional generation $p(\mathbf{x} \mid \mathbf{y})$, e.g., create an image $\mathbf{x}$ conditional on a class $\mathbf{y}$.

- One approach is to give each convolution layer a class-dependent bias term $\mathbf{b_y}$ that depends on $\mathbf{y}$ (e.g., 1-hot class label), which is multiplied by a learned weight matrix.

x_1	x_2	x_3	x_4	x_5
x_6	x_7	x_8	x_9	x_{10}
x_{11}	x_{12}	x_{13}	x_{14}	x_{15}
x_{16}	x_{17}	x_{18}	x_{19}	x_{20}
x_{21}	x_{22}	x_{23}	x_{24}	x_{25}

Latent-space autoregression for image generation

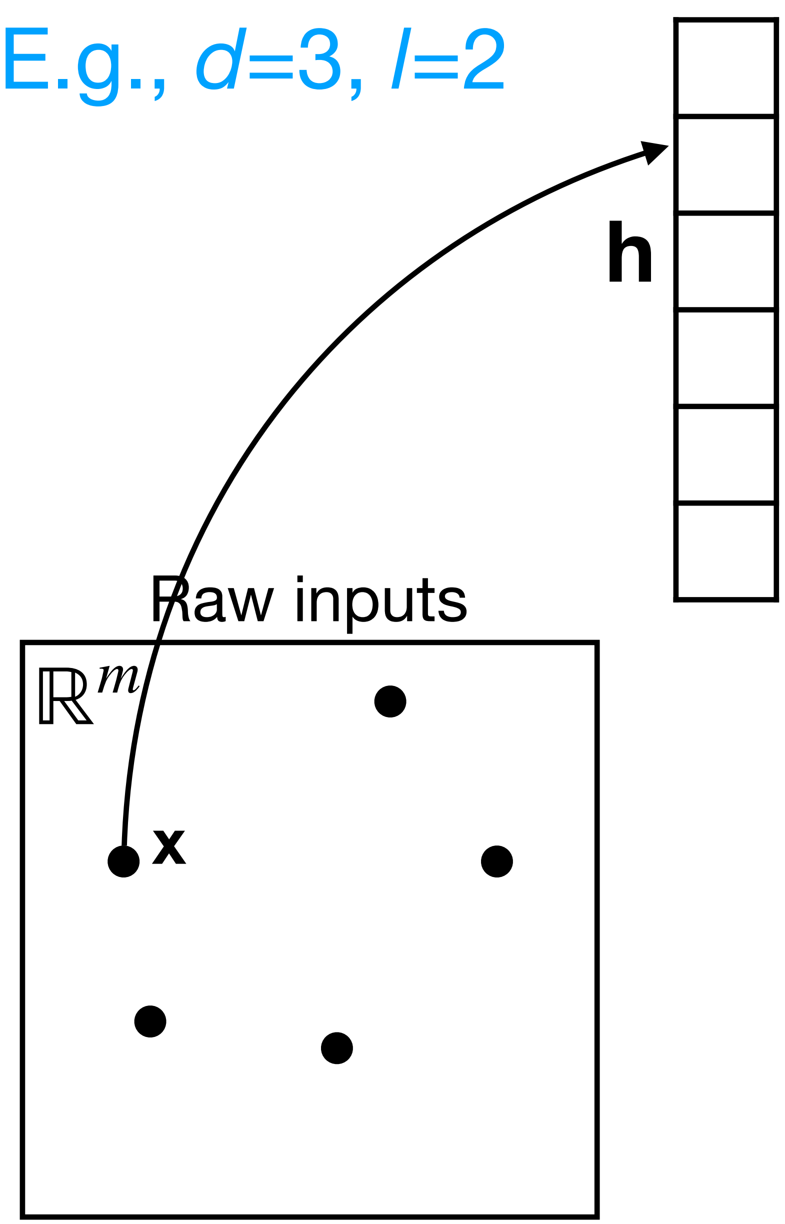
Latent-space autoregression for image generation

- Autoregressing large images pixel-by-pixel is slow.
- Rather than generate pixels directly, we can instead autoregress the (discrete) variables of a latent feature map **z.**
- We can then decode **z** into **x** using a trained autoencoder.
- We typically use a VQ-VAE since it is **discrete** and suitable for **generation**.

(Deep) VQ-VAEs

- We can generalize this idea into a deep VQ-VAE:
  1. Transform each $\mathbf{x} \in \mathbb{R}^m$ into a feature vector $\mathbf{h} \in \mathbb{R}^{l \times d}$.

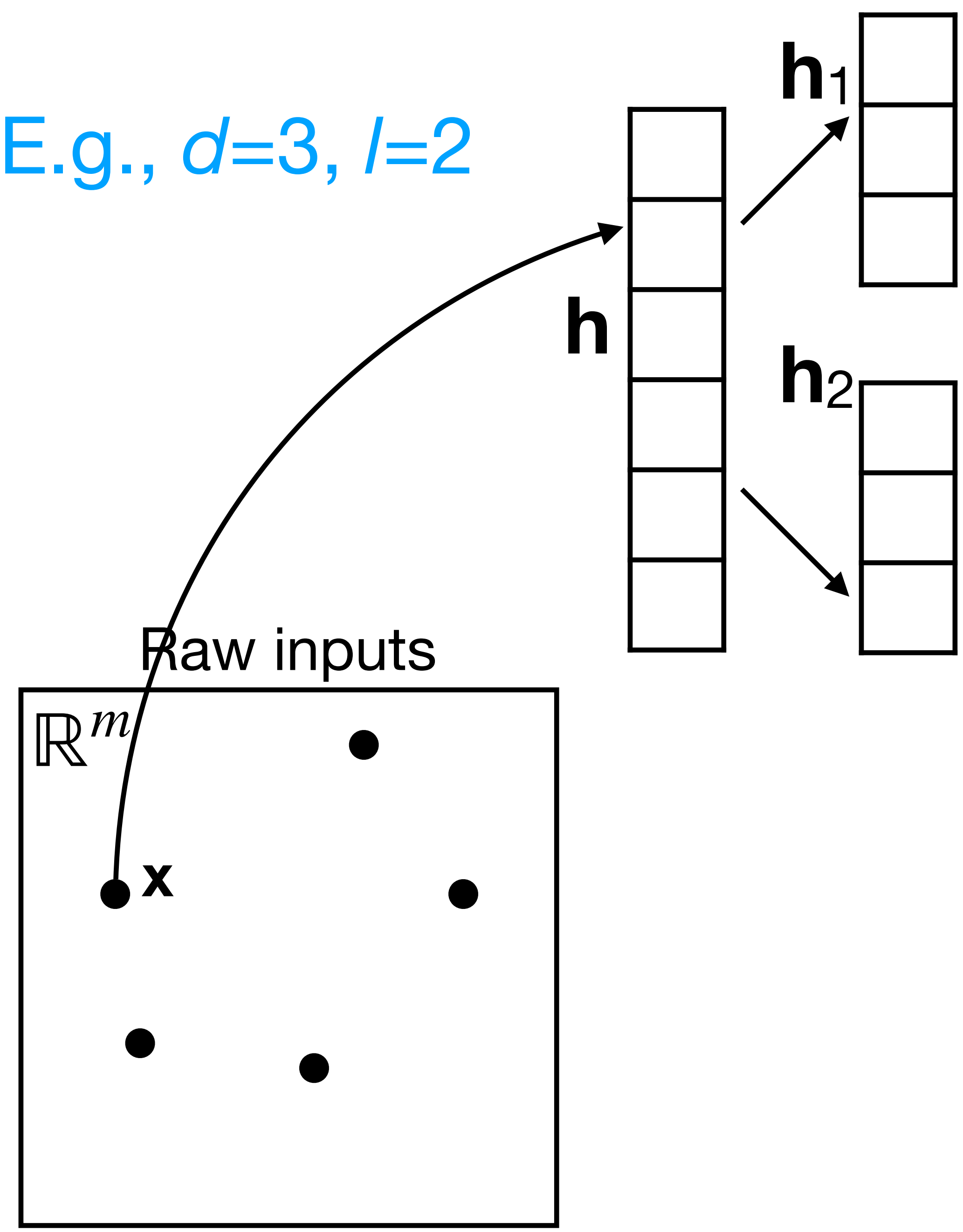
E.g., d=3, l=2



(Deep) VQ-VAEs

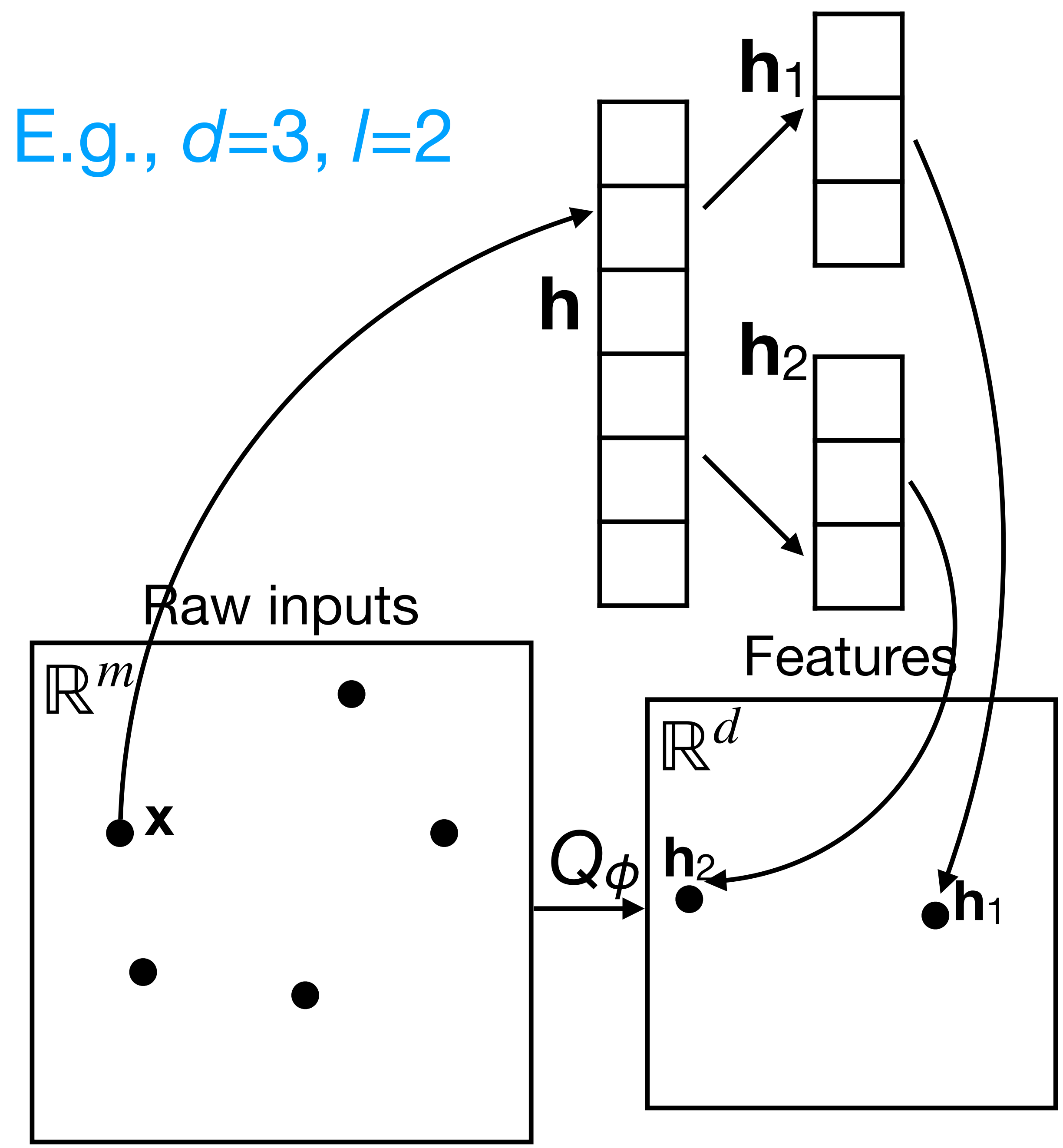
- We can generalize this idea into a deep VQ-VAE:
    1. Transform each $\mathbf{x} \in \mathbb{R}^m$ into a feature vector $\mathbf{h} \in \mathbb{R}^{l \times d}$.
    2. Split $\mathbf{h}$ into multiple (l) vectors $\mathbf{h}_1, \ldots, \mathbf{h}_l \in \mathbb{R}^d$.

E.g., d=3, l=2



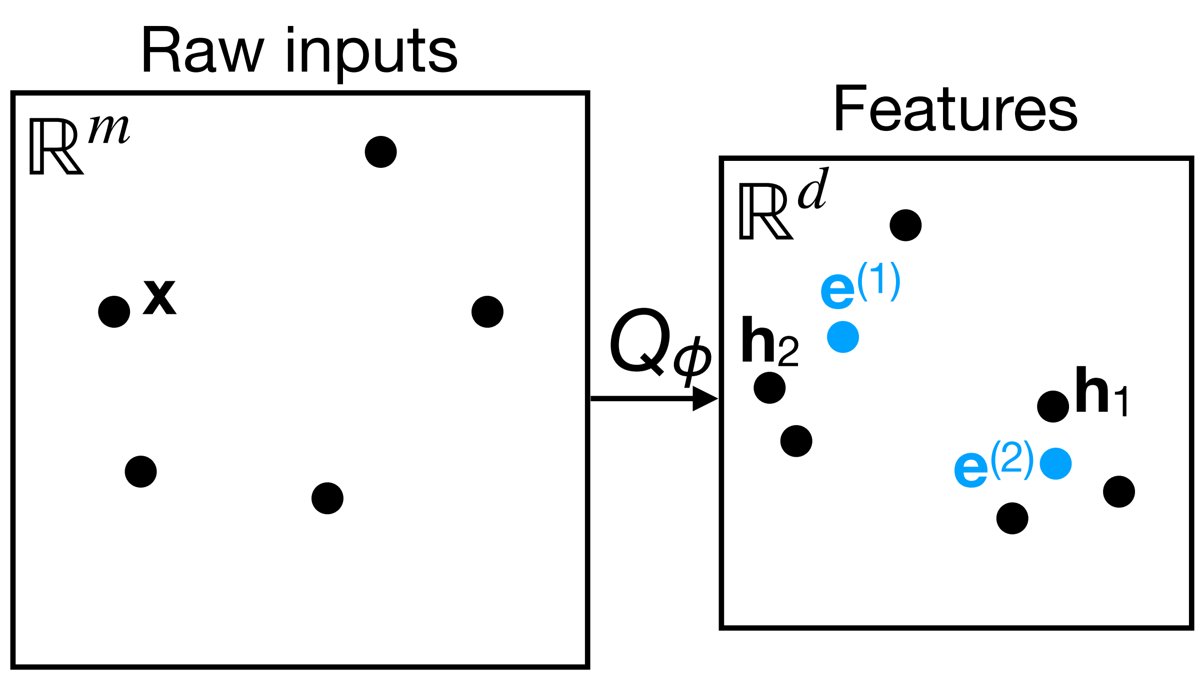
(Deep) VQ-VAEs

- We can generalize this idea into a deep VQ-VAE:
  1. Transform each $\mathbf{x} \in \mathbb{R}^m$ into a feature vector $\mathbf{h} \in \mathbb{R}^{l \times d}$.
  2. Split $\mathbf{h}$ into multiple (l) vectors $\mathbf{h}_1, \ldots, \mathbf{h}_l \in \mathbb{R}^d$.



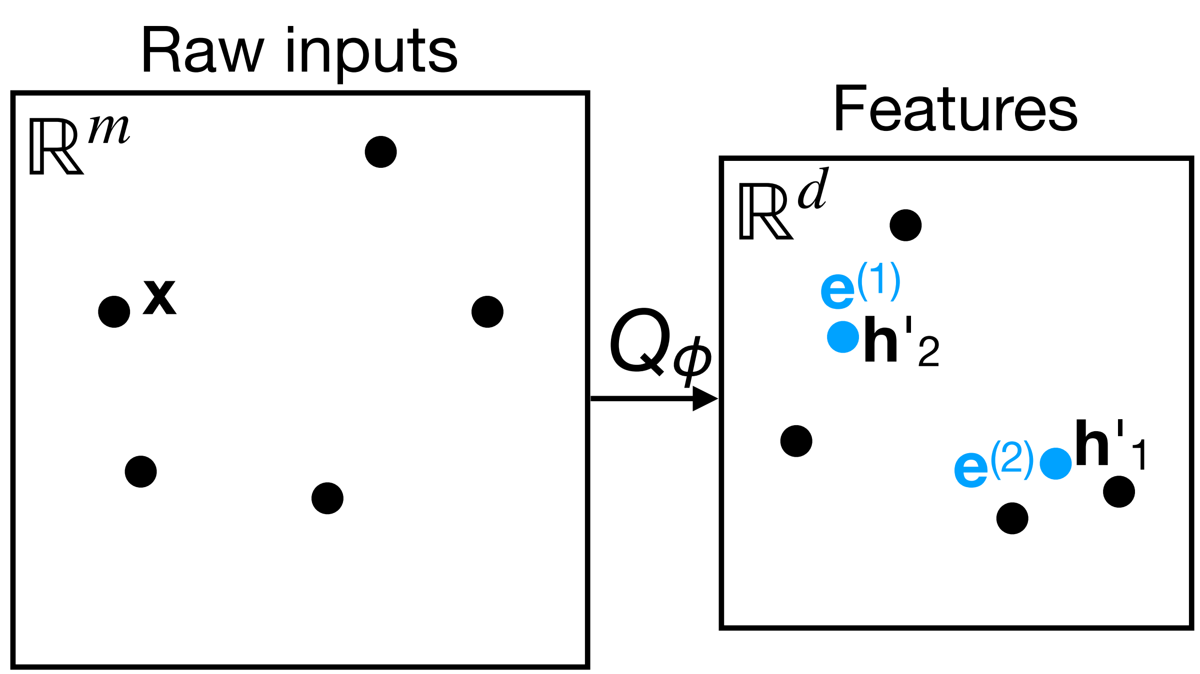
(Deep) VQ-VAEs

- We can generalize this idea into a deep VQ-VAE:
  1. Transform each $\mathbf{x} \in \mathbb{R}^m$ into a feature vector $\mathbf{h} \in \mathbb{R}^{l \times d}$.
  2. Split $\mathbf{h}$ into multiple (l) vectors $\mathbf{h}_1, \ldots, \mathbf{h}_l \in \mathbb{R}^d$.
  3. Using running estimates of K cluster centroids over $\{\mathbf{h}_j^{(i)}\}_{i,j}$,



(Deep) VQ-VAEs

- We can generalize this idea into a deep VQ-VAE:
  1. Transform each $\mathbf{x} \in \mathbb{R}^m$ into a feature vector $\mathbf{h} \in \mathbb{R}^{l \times d}$.
  2. Split $\mathbf{h}$ into multiple (l) vectors $\mathbf{h}_1, \ldots, \mathbf{h}_l \in \mathbb{R}^d$.
  3. Using running estimates of K cluster centroids over $\{\mathbf{h}_j^{(i)}\}_{i,j}$, quantize each $\mathbf{h}_j$ into $\mathbf{h}'_j$ using the nearest centroid $\mathbf{e}^{(z)}$.

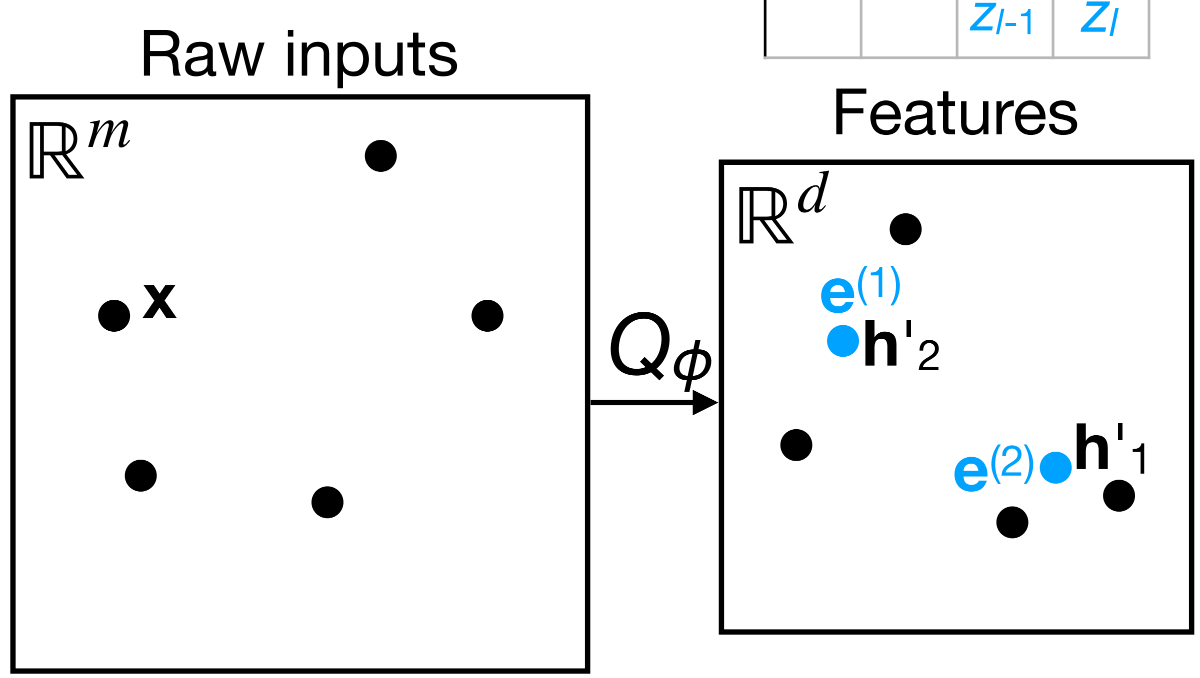


(Deep) VQ-VAEs

- We can generalize this idea into a deep VQ-VAE:
  1. Transform each $\mathbf{x} \in \mathbb{R}^m$ into a feature vector $\mathbf{h} \in \mathbb{R}^{l \times d}$.
  2. Split $\mathbf{h}$ into multiple (l) vectors $\mathbf{h}_1, \ldots, \mathbf{h}_l \in \mathbb{R}^d$.
  3. Using running estimates of K cluster centroids over $\{\mathbf{h}_j^{(i)}\}_{i,j}$, quantize each $\mathbf{h}_j$ into $\mathbf{h}'_j$ using the nearest centroid $\mathbf{e}^{(z)}$.

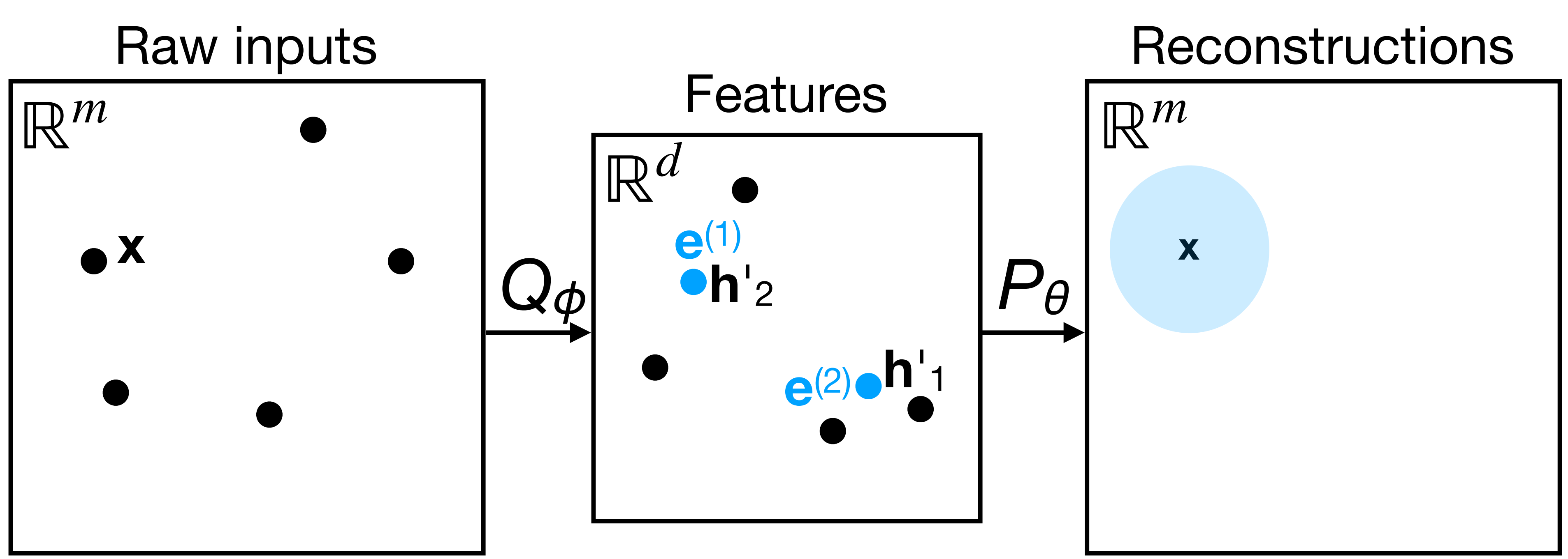
This maps **x** into an array of discrete codes.

$$\mathbf{z} = \begin{array}{|c|c|c|c|} \hline z_1 & z_2 & z_3 & z_4 \\ \hline z_5 & & & \\ \hline & & \ldots & \\ \hline & & z_{l-1} & z_l \\ \hline \end{array} \in \{1, \ldots, K\}^l$$



(Deep) VQ-VAEs

- We can generalize this idea into a deep VQ-VAE:
  1. Transform each $\mathbf{x} \in \mathbb{R}^m$ into a feature vector $\mathbf{h} \in \mathbb{R}^{l \times d}$.
  2. Split $\mathbf{h}$ into multiple (l) vectors $\mathbf{h}_1, \ldots, \mathbf{h}_l \in \mathbb{R}^d$.
  3. Using running estimates of K cluster centroids over $\{\mathbf{h}_j^{(i)}\}_{i,j}$, quantize each $\mathbf{h}_j$ into $\mathbf{h}_j'$ using the nearest centroid $\mathbf{e}^{(z)}$.
  4. Concatenate the $\mathbf{h}_1', \ldots, \mathbf{h}_l'$ into $\mathbf{h}'$, and then transform $\mathbf{h}'$ into $P(\mathbf{x} \mid \mathbf{h}') = P(\mathbf{x} \mid \{z_j\})$.



D_{KL} for VQ-VAEs

- For VQ-VAEs, we want $P(z) = \mathcal{U}(1, \ldots, K) = \frac{1}{K} \; \forall z$.

D_{KL} for VQ-VAEs

- For VQ-VAEs, we want $P(z) = \mathscr{U}(1, \ldots, K) = \dfrac{1}{K}\ \forall z$.

- We have a deterministic encoder:

$$Q(z \mid \mathbf{x}) = \begin{cases} 1 & \text{if } z = \arg\min_k \|\mathbf{x} - \mathbf{e}^{(k)}\|^2 \\ 0 & \text{otherwise} \end{cases}$$

- By definition of KL divergence, we have:

$$\begin{aligned} D_{\mathrm{KL}}(Q_\phi(z \mid \mathbf{x}) \mid P(z)) &= \sum_{z=1}^{K} Q(z \mid \mathbf{x}) \log \frac{Q(z \mid \mathbf{x})}{P(z)} \\ &= 1 \log \frac{1}{\frac{1}{K}} + \sum_{\ldots} 0 \log \frac{0}{\frac{1}{K}} \\ &= \log K \end{aligned}$$

D_{KL} for VQ-VAEs

- Since log K does not depend on any of the VQ-VAE's parameters (ϕ, θ, $\mathbf{E}$), it can be ignored from the ELBO.

- That just leaves:

$$\cancel{-D_{\mathrm{KL}}(Q_\phi(z \mid \mathbf{x}) \mid P(z))} + \mathbb{E}_{Q_\phi}[\log P(\mathbf{x} \mid z)]$$

Autoregression of VQ-VAE latent space

- In practice, this means that $Q(\mathbf{z} \mid \mathbf{x})$ and $P(\mathbf{z})$ in VQ-VAEs tend to be very different from the uniform distribution.
- Hence, sampling from $P(\mathbf{z})$ and then decoding naively tends to produce very bad results.
- Instead, we can train an autoregressor to model $P(\mathbf{z})$.
- Note the same problem can also occur for continuous VAEs but is generally less severe.

Autoregression of VQ-VAE latent space

- Early VQ-VAEs used PixelCNN for latent-space autoregression.

- This yields a 2-stage VQ-VAE training procedure:

  1. Train the VQ-VAE encoder and decoder jointly.

  2. Train a PixelCNN on the latent codes { $\mathbf{z}$ }.

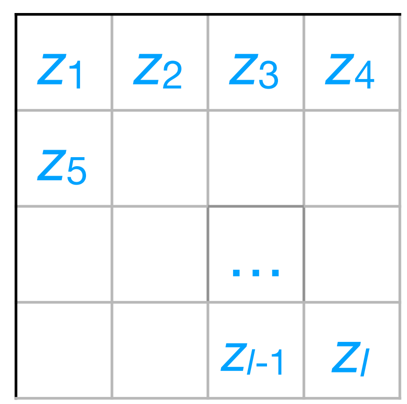
Autoregression of VQ-VAE latent space

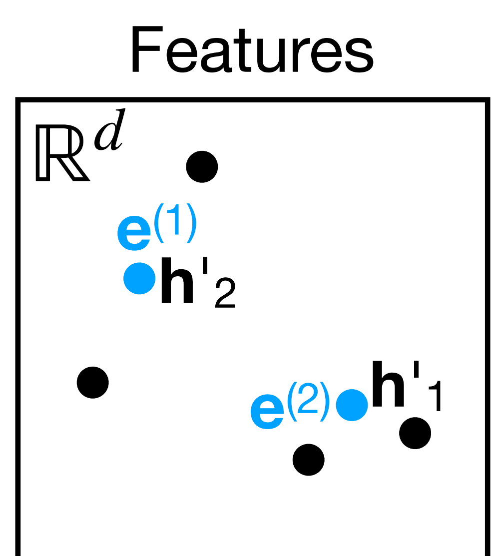
- Inference procedure for generation:
    1. Using the PixelCNN, autoregress the latent code **z**.

z_1	z_2	z_3	z_4
z_5			
		…	
		z_{l-1}	z_l

Autoregression of VQ-VAE latent space

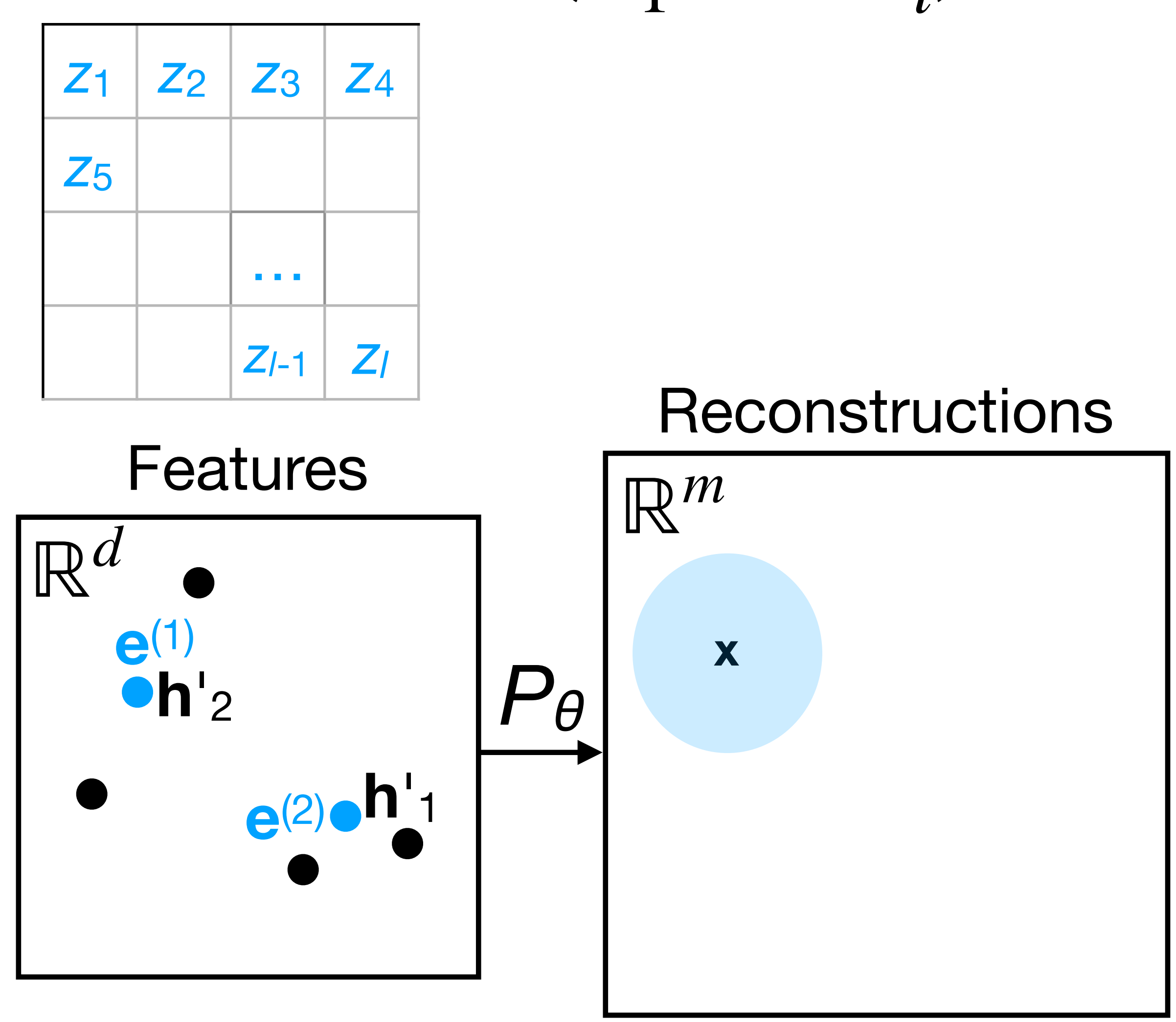
- Inference procedure for generation:
    1. Using the PixelCNN, autoregress the latent code **z**.
    2. Map **z** to their cluster centroids $(\mathbf{h}'_1, \ldots, \mathbf{h}'_l) = \mathbf{h}'$.





Autoregression of VQ-VAE latent space

- Inference procedure for generation:
  1. Using the PixelCNN, autoregress the latent code **z**.
  2. Map **z** to their cluster centroids $(\mathbf{h}'_1, \ldots, \mathbf{h}'_l) = \mathbf{h}'$.
  3. Decode **h'** into **x**.



Autoregression of VQ-VAE latent space

- VQ-VAE + latent-space autoregression (using Transformers) can generate images with state-of-the-art quality (rivaling diffusions).

- In particular, VQ-VAE typically produce sharper images compared to continuous VAEs:
    - The KL regularization term in continuous VAEs is necessary to shape $P(\mathbf{z}) \approx \mathcal{N}(\mathbf{0}, \mathbf{I})$.
    - However, it leads to multiple **x** sharing similar **z**, causing the decoder to "blur" the reconstruction to minimize MSE.
    - In theory, the same could happen to VQ-VAEs, but in practice, without the KL term, the problem is less severe.

Autoregressive text generation

Autoregressive text generation

- Text is an inherently **discrete** domain consisting of a sequence of tokens (e.g., words, word-parts).
- While unconditional generation $p(\mathbf{x})$ is possible, conditional generation $p(\mathbf{x} \mid \mathbf{y})$ is more common, e.g.:
  - Translate a sentence from one language to another.
  - Respond to a prompt.
  - Text captioning of an image

Autoregressive text generation: notation

- Sometimes we define "generation" to be $p(\mathbf{x} \mid \mathbf{y})$:
    - Machine translation: **y** is input sentence, and **x** is a translation.
    - Text captioning: **y** is an image, and **x** is a caption.

- Other times we define it as $p(x_{T+T'}, \ldots, x_{T+1} \mid x_T, \ldots, x_1)$:
    - Respond to a prompt, which consists of first *T* tokens.

- The choice of notation is subjective.

Autoregressive text generation

- Since the generated text is typically variable-length, a natural architecture is an RNN.
  - Keep autoregressing until an end-of-sentence (EOS) symbol is sampled.
- As of 2025, Transformers (rather than RNNs) are dominant, but hybrid attentional-recurrent models (e.g., Mamba) are also gaining traction.