

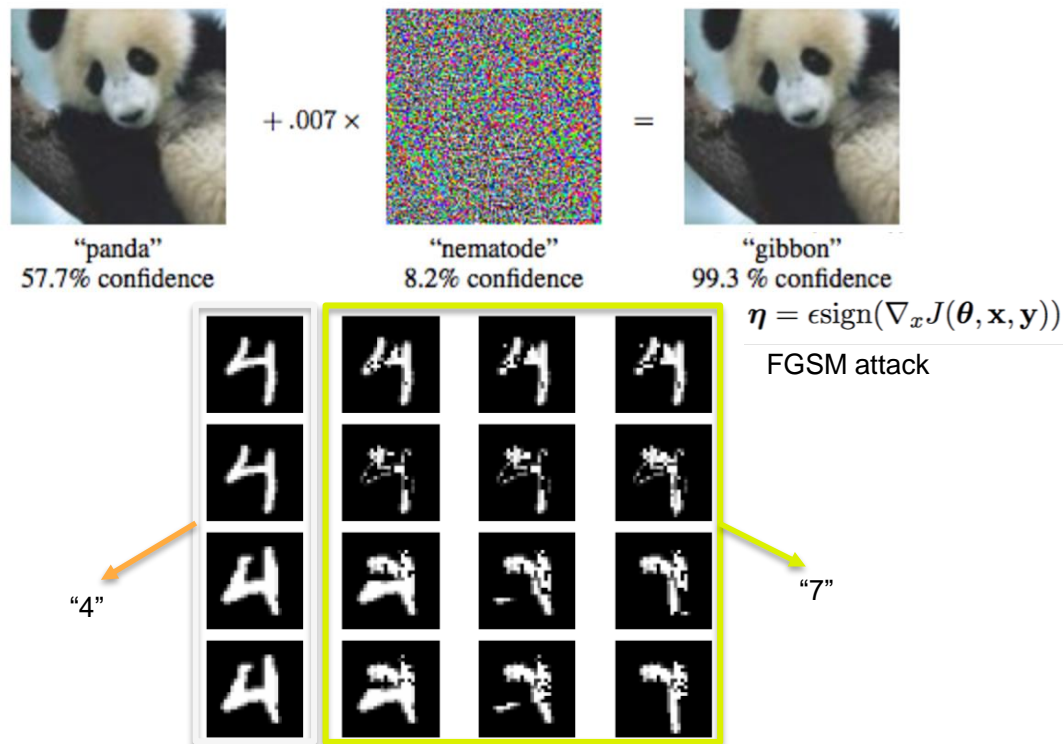
CS182/282A: Designing, Visualizing and Understanding Deep Neural Networks

John Canny

Spring 2019

Lecture 17: Generative Models

Last Time: Adversarial Examples



Goodfellow, Ian J., Jonathon Shlens, and Christian Szegedy. "Explaining and harnessing adversarial examples." *ICLR 2015*.

Bo Li, Yevgeniy Vorobeychik, and Xinyun Chen. "A General Retraining Framework for Scalable Adversarial Classification." *ICLR*. (2016).

Last Time: Optimizing an Adversarial Objective

Fast approaches

- Fast gradient sign ($d = \|\cdot\|_\infty$): $x^* = x + B \text{sgn}(\nabla_x \ell(f_\theta(x), y))$
- Fast gradient ($d = \|\cdot\|_2$): $x^* = x + B \left(\frac{\nabla_x \ell(f_\theta(x), y)}{\|\nabla_x \ell(f_\theta(x), y)\|_2} \right)$

Iterative approaches

- E.g., use a SGD optimizer, such as Adam, to optimize

$$\max_{x^*} \ell(f_\theta(x^*), y) + \lambda d(x, x^*)$$

Optimization

$$\argmin_{\delta} \lambda \|\delta\|_p + J(f_\theta(x + \delta), y^*)$$

Need to know model f_θ

Last Time: Black-box Attacks

Black-box attacks are possible on deep neural networks with **query access**.

The number of queries needed can be reduced



Original image, classified as
“drug” with a confidence of 0.99



Adversarial example, classified as
“safe” with a confidence of 0.96

The Gradient Estimation black-box attack on Clarifai’s Content Moderation Model

Last Time: Adversarial Examples from Adversarial Nets



(a) Strawberry



(b) Toy poodle

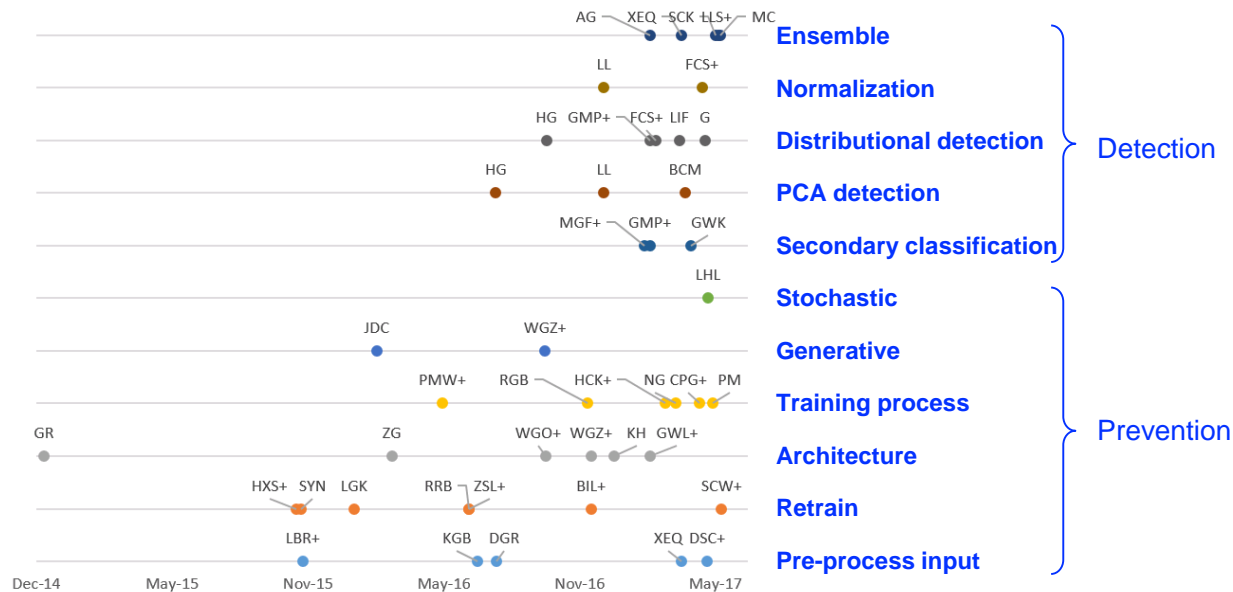


(c) Buckeye



(d) Toy poodle

Last Time: Detecting/Defending Against Adversarial Examples



Updates

- Assignment 3 is due tonight, 11pm.
- Midterm 2 is in-class next Wednesday 4/10 at 8:10am
- 182 midterm here, 282A in 306 Soda
- Review session Thursday night 6-8pm in 4 LeConte.

This Time: Generative Models

- Variational Auto-Encoders (VAEs)
- Auto-Regressive Models
- Transformers (!)
- Generative Adversarial Networks (next time)

Generative Models

Generative models “represent” the full joint distribution $p(x, y)$ where x is an input sample and y is an output or label. There are two ways to do this:

Classifier: Given x and y , determine the probability of the pair (x, y) . This is also called a generative classifier.

Generator: Given y , generate a sample from $x \sim p(\cdot, y)$.

In classical machine learning using Bayesian methods, a generative model is usually a generative classifier that realizes a particular assumption about how the data was generated. Its typically easy to tweak this to generate samples.

In deep learning, we are mostly concerned with actually generating samples that generalize a given dataset (X, Y) . This gives efficient generators that cannot be used to compute $p(x, y)$.

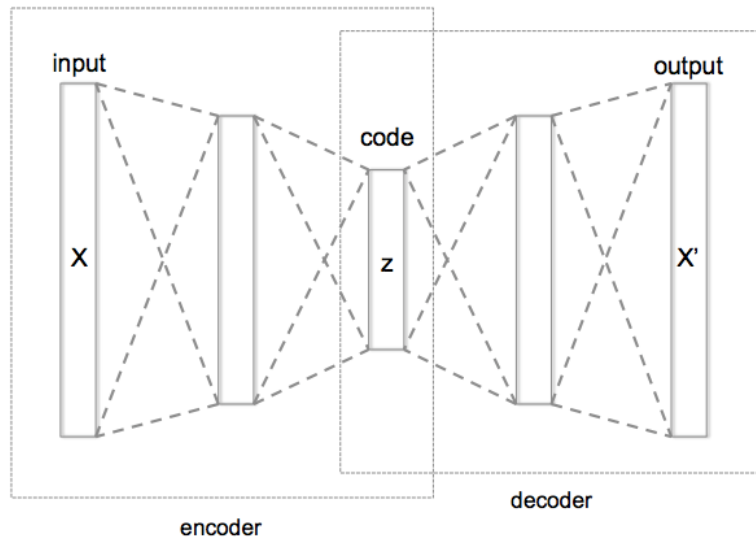
Auto-Encoders

Auto-encoders transform each input x into a “code” variable z , and then back to a synthetic input x' that should be as close as possible to x . The dimension of z is usually much lower than x , and forms an “information bottleneck” to x' .

Encoder: Very similar to, or could be exactly a neural classifier. Can be thought of as a data compressor.

Code: A representation of the input designed to support reconstruction. Could be the set of class labels, but also often encodes within-class variability.

Decoder: Is a conditional synthetic input generator (a generative model).



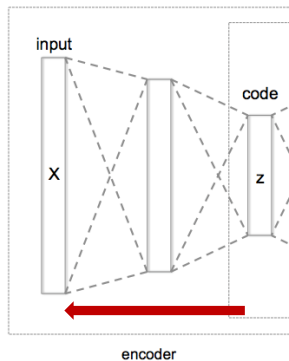
Building Good Decoders (Generators)

First of all, if we have a good encoder $z = f(x)$, do we really need a separate decoder, or can we just “invert” f and compute $x \approx f^{-1}(z)$?

Answer: Yes, in fact we already did this with DNN visualization via activation maximization.

But: There are significant challenges:

- Inversion is an iterative process (SGD or Monte-Carlo) and may be very slow.
- A deep network classifier is not optimized for coding, so we need to train it to minimize the loss $L(x, x')$. But the mapping from $z \rightarrow x'$ is not differentiable, so this requires REINFORCE.



Reconstructions from the representation after last pooling layer (immediately before the first Fully Connected layer)

Implicit Auto-Encoders

Suppose instead we have a decoder only, which is a parametric function (neural network) $x = f(z, \theta)$ for some model parameters θ .

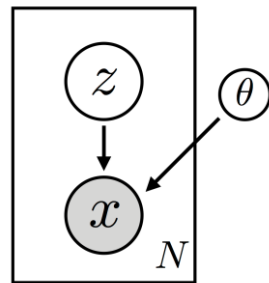
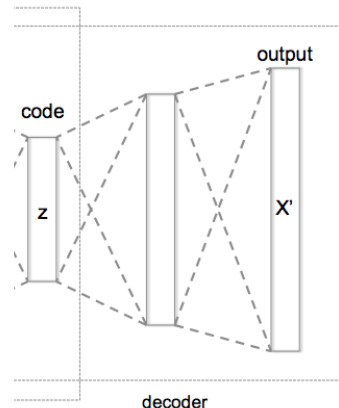
Train: Given a real input x , we can invert $f(\cdot, \theta)$ somehow to get z and then minimize $L(x, x' = f(z, \theta))$ wrt θ .

Difficulties:

- Potentially slow because of iterative inversion of $f(\cdot, \theta)$.
- Potentially slow because of high variance from individual samples x .

VAEs:

- Build the decoder as a probabilistic model $p_\theta(x|z)$.
- Invert using variational inference to get an *approximation* to $p_\theta(z|x)$.



Variational Auto-Encoders

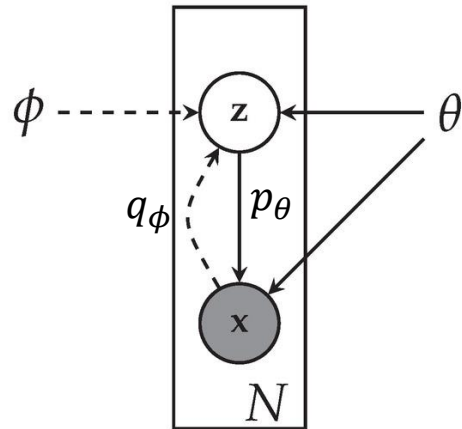
We cant directly compute $p_{\theta}(z|x)$ if $p_{\theta}(x|z)$ is a neural network, i.e. the distribution $p_{\theta}(z|x)$ is intractable.

So instead, we use an *approximate* inverse density $q_{\phi}(z|x)$ which is again implemented with a neural network, this time with parameters ϕ .

This is usually represented with this graphical model:

The solid lines show the actual model $p_{\theta}(x|z)$, where values of x are computed by a θ -parametrized deep network from z .

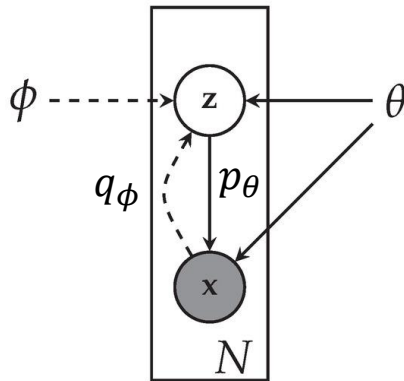
The dotted lines show the *variational approximation* $q_{\phi}(z|x)$ to the inverse model $p(z|x)$. The q_{ϕ} network takes x as input.



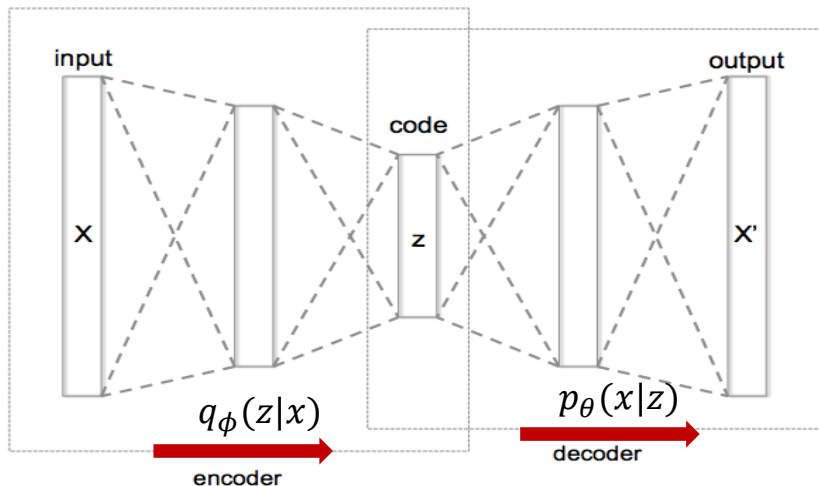
Variational Auto-Encoders

The solid lines show the actual model $p_{\theta}(x|z)$, where values of x are computed by a θ -parametrized deep network from z .

The dotted lines show the *variational approximation* $q_{\phi}(z|x)$ to the inverse model $p(z|x)$. The q_{ϕ} network takes x as input.



This can be viewed as a probabilistic auto-encoder



Representing Densities with Neural Nets (Reparameterization Trick)

We can represent a density $q_\phi(z|x)$ with a neural network by defining

$$z = g_\phi(x, \epsilon)$$

where ϵ is a random or noise variable, and g_ϕ is a regular, feed-forward network. Such a q_ϕ can be used to generate samples, but not necessarily to compute the density given z, x .

e.g. a particularly simple case is additive, scaled gaussian noise

$$g_\phi(x, \epsilon) = g_0(x) + g_1(x) \circ \epsilon$$

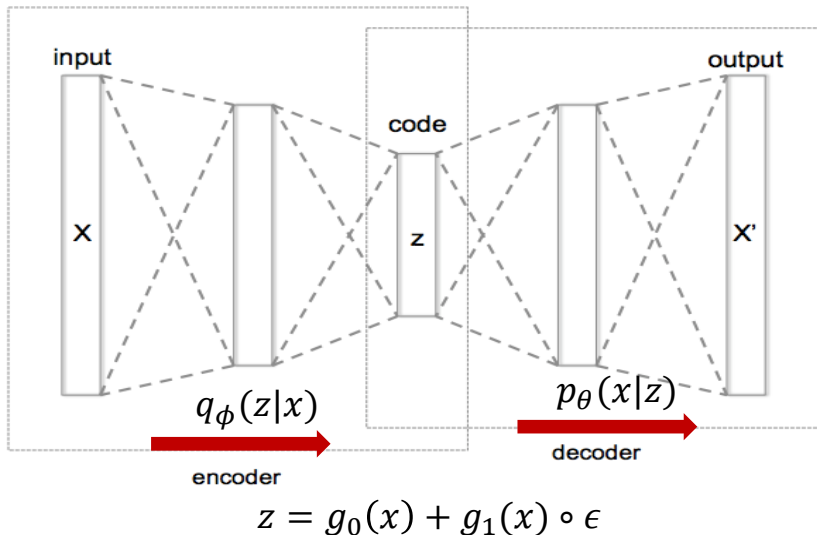
with $\epsilon \sim \mathcal{N}(0, I)$, and \circ denotes element-wise product.

Note: with this choice we *can* compute the density $q_\phi(z|x)$ given x, z .

Variational Auto-Encoders

With that choice we represent $q_\phi(z|x)$ with the function $g_\phi(x, \epsilon)$ with $\epsilon \sim \mathcal{N}(0, I)$.

i.e. we represent a normal distribution of z -values with a mean g_0 and standard deviation (vector) of g_1 .



Loss Functions for VAE

Since this is a generative model, we typically optimize the marginal likelihood of the data (find parameters that make the data most likely): i.e. maximize:

$$\sum_{i=1}^N \log p(x^i)$$

Our model depends on a hidden parameter z which we have to integrate over to get the likelihood above:

$$\begin{aligned} \log p(x^i) &= \log \int_z p_\theta(x^i, z) dz && \text{un-marginalize} \\ &= \log \int_z p_\theta(x^i, z) \frac{q_\phi(z|x^i)}{q_\phi(z|x^i)} dz && \times \text{an identity expression} \\ &= \log E_{z \sim q_\phi(z|x^i)} \left[\frac{p_\theta(x^i, z)}{q_\phi(z|x^i)} \right] && \text{re-write as an expectation over } q_\phi \end{aligned}$$

Loss Functions for VAE

The log likelihood for a point x is:

$$\log p(x) = \log E_{z \sim q_\phi(z|x)} \left[\frac{p_\theta(x, z)}{q_\phi(z|x)} \right]$$

Jensen's inequality (log is a concave function):

$$\geq E_{z \sim q_\phi(z|x)} \left[\log \frac{p_\theta(x, z)}{q_\phi(z|x)} \right]$$

$$\log E(\dots) \geq E \log(\dots)$$


Instead of maximizing $\log p(x)$, we maximize this *variational lower bound*. It has the form:

$$\mathcal{L}(\theta, \phi; x) = E_{z \sim q_\phi(z|x)} [\mathcal{L}(\theta, \phi; x, z)]$$

where

$$\mathcal{L}(\theta, \phi; x, z) = \log \frac{p_\theta(x, z)}{q_\phi(z|x)} = \log p_\theta(x, z) - \log q_\phi(z|x)$$

Aside: Jensen's Inequality

For concave $f(x)$ i.e. $f'' < 0$, (here its $\log x$) construct a linear function $l(x)$ tangent to $f(x)$ at $\bar{x} = E_p[x]$. By construction of the tangent line: $f(\bar{x}) = l(\bar{x})$ and $f(x) \leq l(x)$.

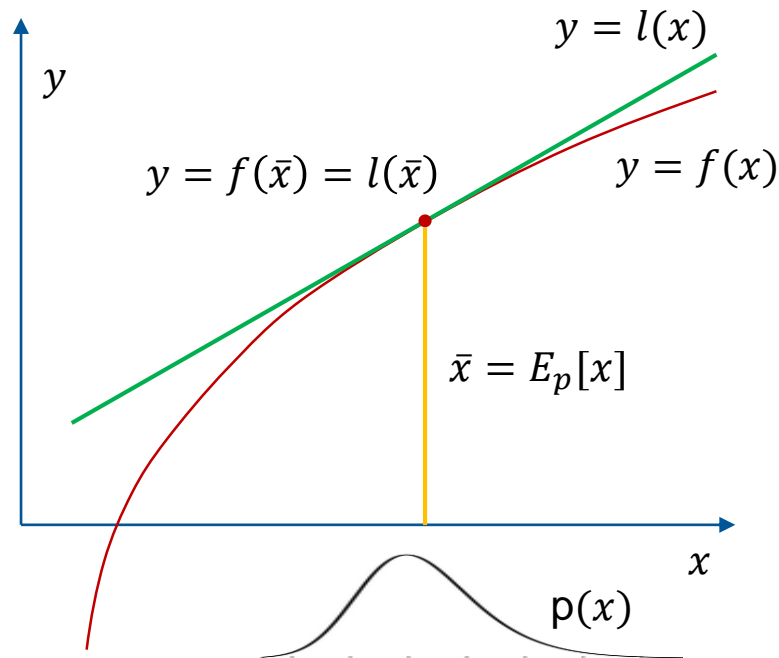
For a linear function $l(E_p[x]) = E_p[l(x)]$,
so $f(E_p[x]) = l(E_p[x]) = E_p[l(x)]$

Since $f(x) \leq l(x)$ it follows

$$f(E_p[x]) = E_p[l(x)] \geq E_p[f(x)]$$

Similarly if $f(x)$ is convex, $f(E_p[x]) \leq E_p[f(x)]$

Finally, for the multivariate case $f(E_{z \sim q(z)}[h(z)])$
simply take $x = h(z)$ and for $p(x)$ the
distribution on x induced by sampling $z \sim q(z)$.



Optimizing Reparametrized Models

It's easy to optimize an objective that's an expected value over a reparametrized distribution. Let's assume we have a loss function in that form:

$$\mathcal{L}(\theta, \phi; x) = E_{z \sim q_{\phi}(z|x)}[\mathcal{L}(\theta, \phi; x, z)]$$


expressed in terms of a “pointwise” loss $\mathcal{L}(\theta, \phi; x, z)$. We can use gradient descent by computing

$$\nabla_{\theta, \phi} \mathcal{L}(\theta, \phi; x)$$

and using an SGD optimizer.


Optimizing Reparametrized Models

We're doing gradient descent with:

$$\begin{aligned} & \nabla_{\theta, \phi} E_{z \sim q_{\phi}(z|x)} [\mathcal{L}(\theta, \phi; x, z)] \\ &= \nabla_{\theta, \phi} \int_z q_{\phi}(z|x) \mathcal{L}(\theta, \phi; x, z) dz \end{aligned}$$


Definition of expected value

Now apply reparameterization $z = g(x, \epsilon)$:

$$\begin{aligned} &= \nabla_{\theta, \phi} \int_{\epsilon} p(\epsilon) \mathcal{L}(\theta, \phi; x, g(x, \epsilon)) d\epsilon \\ &= \nabla_{\theta, \phi} E_{\epsilon \sim p(\epsilon)} [\mathcal{L}(\theta, \phi; x, g(x, \epsilon))] \end{aligned}$$


Definition of expected value

Since gradient and expected value are both linear operations, we can swap them:

$$= E_{\epsilon \sim p(\epsilon)} [\nabla_{\theta, \phi} \mathcal{L}(\theta, \phi; x, g(x, \epsilon))]$$

Now this is easy to estimate!

Optimizing Reparametrized Models

We want to estimate

$$E_{\epsilon \sim p(\epsilon)} [\nabla_{\theta, \phi} \mathcal{L}(\theta, \phi; x, g(x, \epsilon))]$$

which we can approximate with L random samples:

$$\approx \frac{1}{L} \sum_{i=1}^L \nabla_{\theta, \phi} \mathcal{L}(\theta, \phi; x, g(x, \epsilon^i))$$

where $\epsilon^i \sim p(\epsilon)$

e.g. a typical model might use $\epsilon^i \sim \mathcal{N}(0, I)$.

Minibatch VAE optimization

Algorithm 1 Minibatch version of the Auto-Encoding VB (AEVB) algorithm. Either of the two SGVB estimators in section 2.3 can be used. We use settings $M = 100$ and $L = 1$ in experiments.

$\theta, \phi \leftarrow$ Initialize parameters

repeat

$\mathbf{X}^M \leftarrow$ Random minibatch of M datapoints (drawn from full dataset)

$\epsilon \leftarrow$ Random samples from noise distribution $p(\epsilon)$

$\mathbf{g} \leftarrow \nabla_{\theta, \phi} \tilde{\mathcal{L}}^M(\theta, \phi; \mathbf{X}^M, \epsilon)$ (Gradients of minibatch estimator (8))

$\theta, \phi \leftarrow$ Update parameters using gradients \mathbf{g} (e.g. SGD or Adagrad [DHS10])

until convergence of parameters (θ, ϕ)

return θ, ϕ

L is the number of samples of ϵ for each data sample.

Practical Issues

The simple loss estimator with $z^i = g(x, \epsilon^i)$ and $\epsilon^i \sim p(\epsilon)$ has the form:

$$\hat{\mathcal{L}}(\theta, \phi; x) \approx \frac{1}{L} \sum_{i=1}^L \log p_{\theta}(x, z^i) - \log q_{\phi}(z^i | x)$$

but can have high variance (??).

Expand: $\mathcal{L}(\theta, \phi; x) = E_{z \sim q_{\phi}(z|x)} [\log p_{\theta}(x|z) + \log p_{\theta}(z) - \log q_{\phi}(z|x)]$

An alternative loss estimator is

$$\hat{\mathcal{L}}^B(\theta, \phi; x) \approx -D_{KL}(q_{\phi}(z|x) || p_{\theta}(z)) + \frac{1}{L} \sum_{i=1}^L \log p_{\theta}(x|z^i)$$

where the KL-divergence $D_{KL}(q_{\phi}(z|x) || p_{\theta}(z)) = -E_{z \sim q_{\phi}(z|x)} \left[\log \frac{p_{\theta}(z)}{q_{\phi}(z|x)} \right]$ is computed symbolically. This estimator assumes that $q_{\phi}(z|x)$ allows density estimation such as $g(x, \epsilon) = g_0(x) + g_1(x) \circ \epsilon$

Summary: Practical Issues

The simple loss estimator

$$\hat{\mathcal{L}}(\theta, \phi; x) \approx \frac{1}{L} \sum_{i=1}^L \log p_{\theta}(x, z^i) - \log q_{\phi}(z^i | x)$$

still requires $q_{\phi}(z|x)$ to provide a density value so it must have a special form like $g_{\phi}(x, \epsilon) = g_0(x) + g_1(x) \circ \epsilon$. Both θ and ϕ gradients are discrete approximations, but correlated.

The alternative loss estimator:

$$\hat{\mathcal{L}}^B(\theta, \phi; x) \approx -D_{KL}(q_{\phi}(z|x) || p_{\theta}(z)) + \frac{1}{L} \sum_{i=1}^L \log p_{\theta}(x|z^i)$$

requires $q_{\phi}(z|x)$ to provide a density and to be integrable in closed form. This requires $z = g(x, \epsilon)$ to have a simple form such as $g_{\phi}(x, \epsilon) = g_0(x) + g_1(x) \circ \epsilon$. The θ gradient is approximate, the ϕ gradient is exact.

Variational Auto-Encoders

VAE models train efficiently but have some issues with blurriness, especially if the simple reparameterization $z = g_\phi(x, \epsilon) = g_0(x) + g_1(x) \circ \epsilon$ is used.

VAEs also say nothing about the form of the encoder and decoder functions $q_\phi(z|x)$ and $p_\theta(x|z)$.

Follow-up work uses invertible stages and auto-regressive feature dependencies, which considerably improved performance:

[Variational Inference with Normalizing Flows, Rezende and Mohamed 2015]
[Improved Variational Inference with Inverse Autoregressive Flow, Kingma et al 2016]

But to make sense of these, we first need to cover auto-regressive models.

**Conv. net as encoder/decoder,
trained on faces**

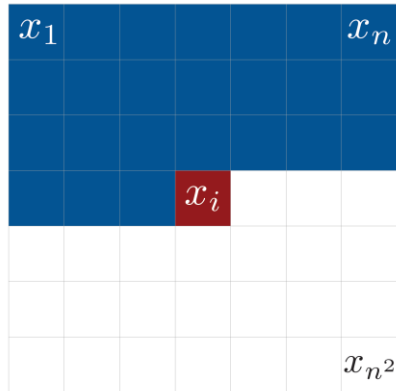


Autoregressive Models

- Key ideas: pixels are highly correlated with each other over various scales.
- Generate one pixel at a time, conditioned on the values of other pixels (i.e. predict or regress on your own predictions – the auto-regressive part).
- For an image with n^2 pixel colors x_1, \dots, x_{n^2} listed in some order, define the probability for an image x :

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

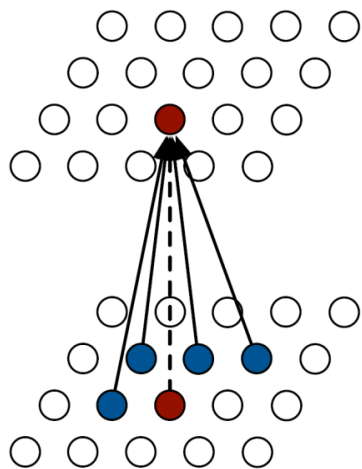
- The current pixel (red) is conditioned on all the previously computed pixels (blue).



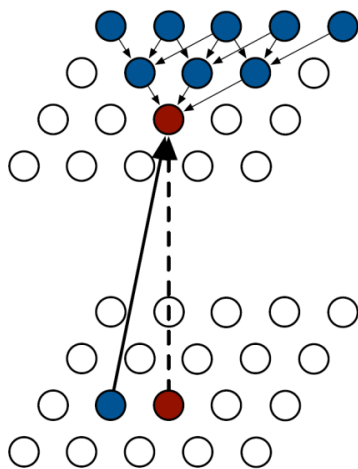
Local Autoregressive Models

In practice it would be rather expensive to condition on all previous x 's for $p(x_i|x_1, \dots, x_{i-1})$.

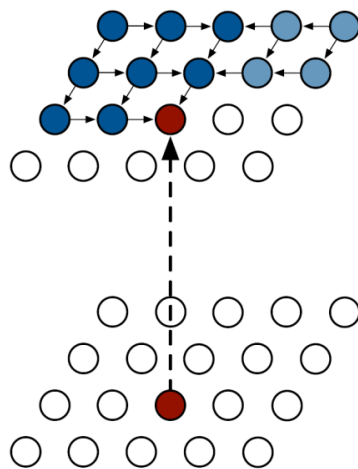
The autoregressive models in the paper use RNNs or CNNs which only examine a few nearby pixels at each step. The RNN models also “remember” the state from more distant pixels via their recurrent state.



PixelCNN



Row LSTM



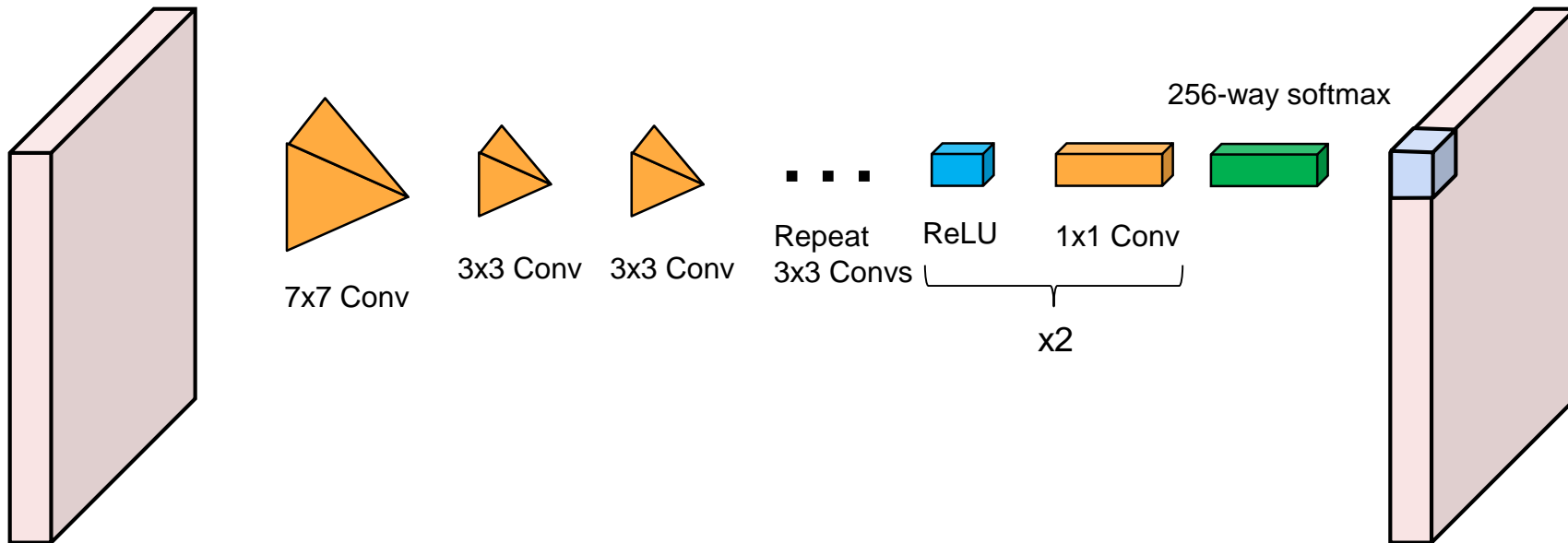
Diagonal BiLSTM

Output layer with recurrent states (blue) and new state (red)

Input layer with past pixels (blue) and new pixel (red)

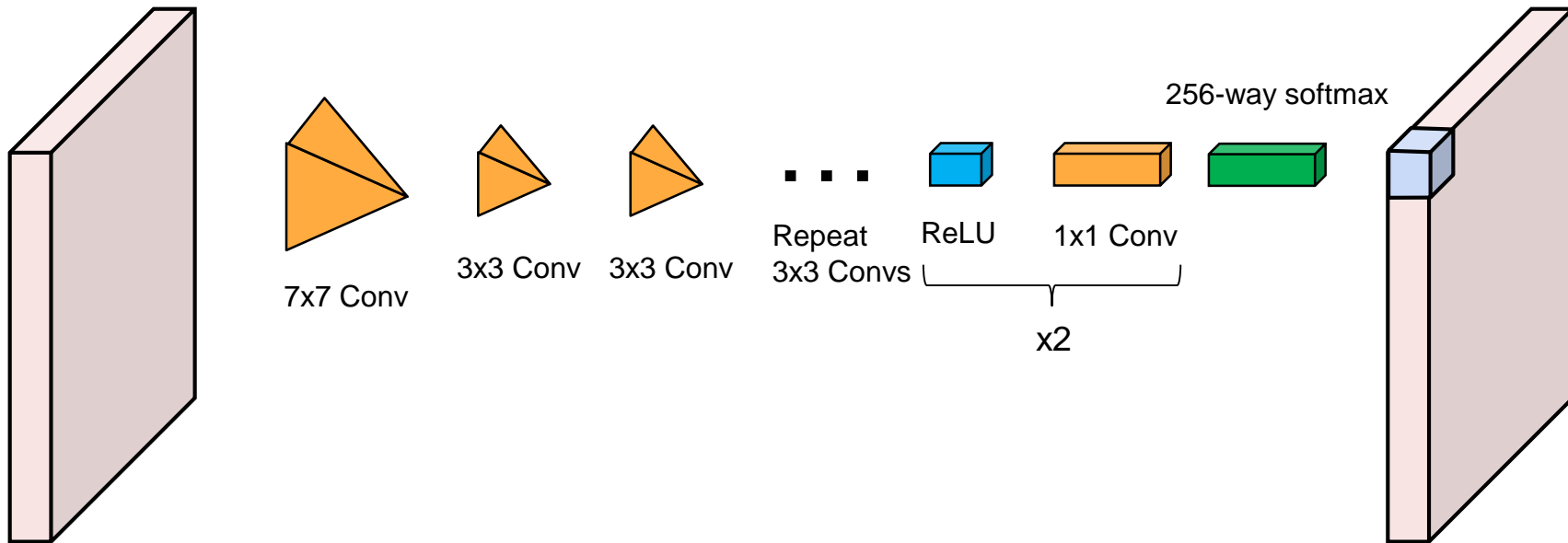
PixelCNN

For PixelCNN, we start with a blank image as input and iteratively generate pixels in it.



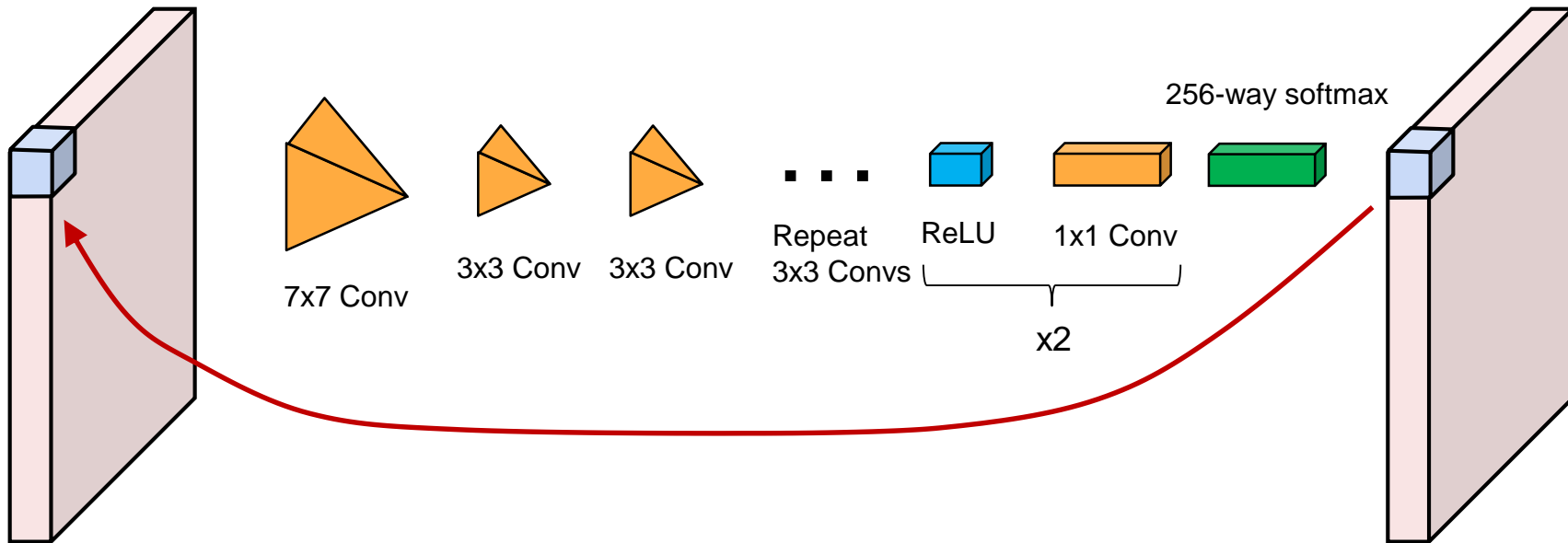
PixelCNN

Aside: PixelCNN actually generates each pixel's *color* one at a time, but we'll ignore that step.



PixelCNN

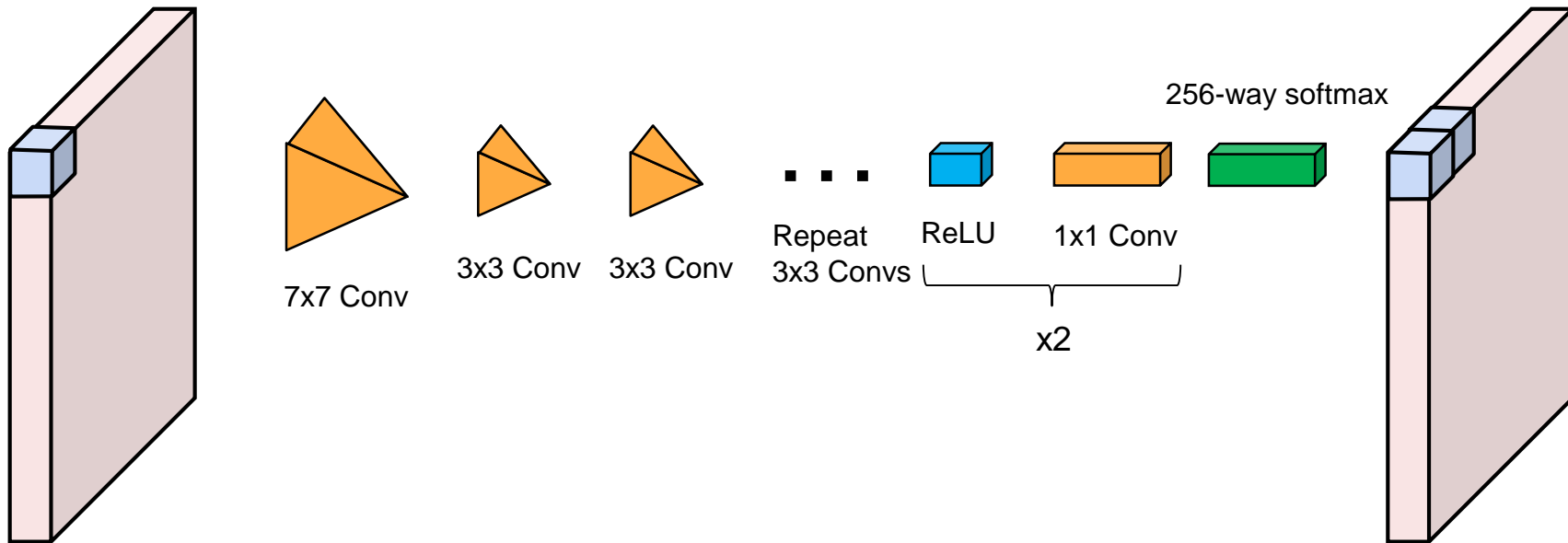
For PixelCNN, we start with a blank image as input and iteratively generate pixels in it.



Then copy the first output pixel back to the input

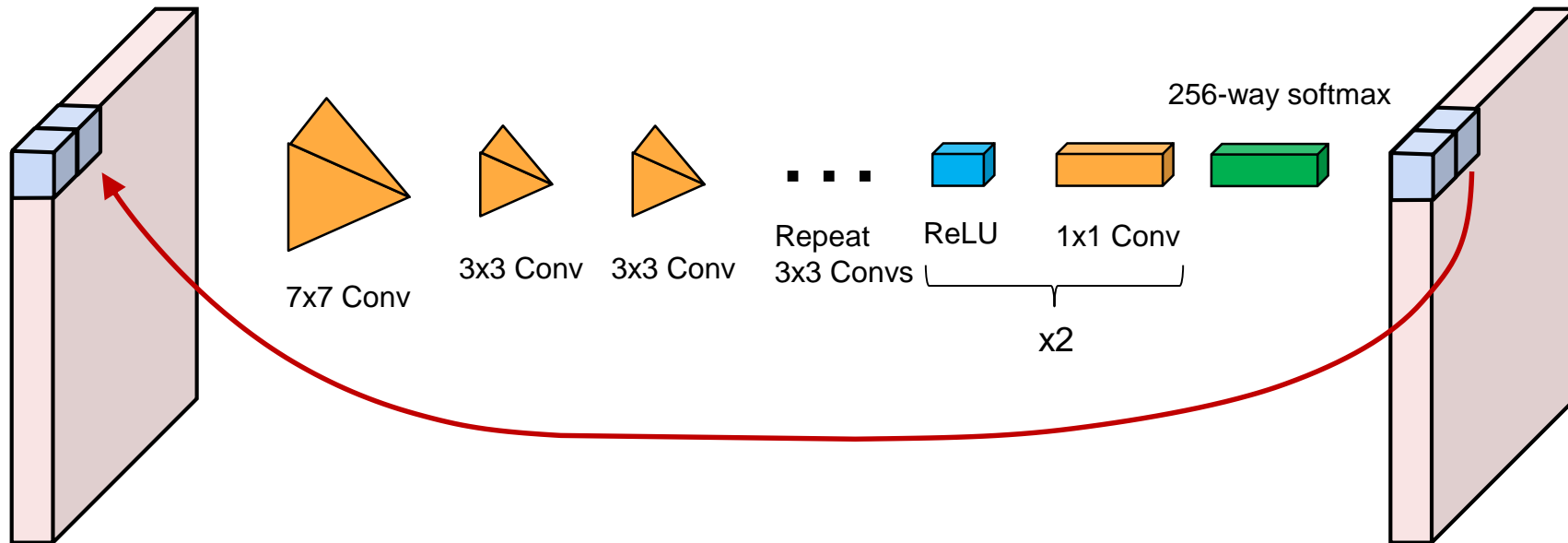
PixelCNN

And then compute the next output pixel



PixelCNN

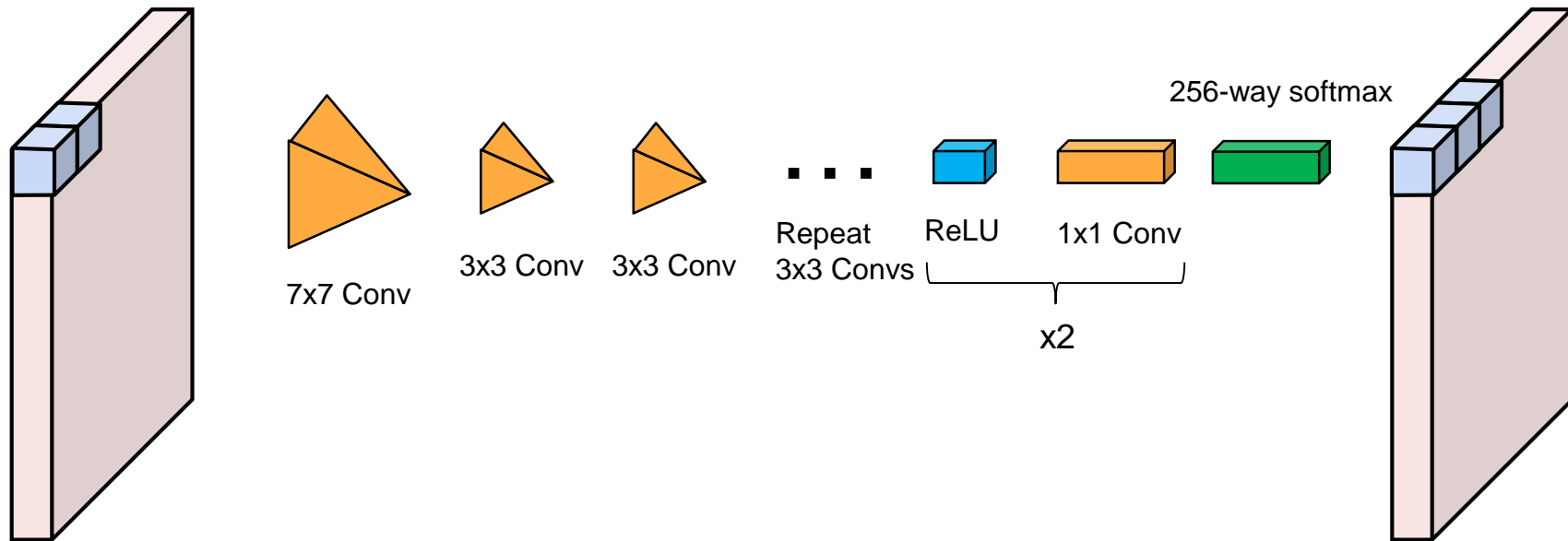
And then compute the next output pixel



and copy it back to the input

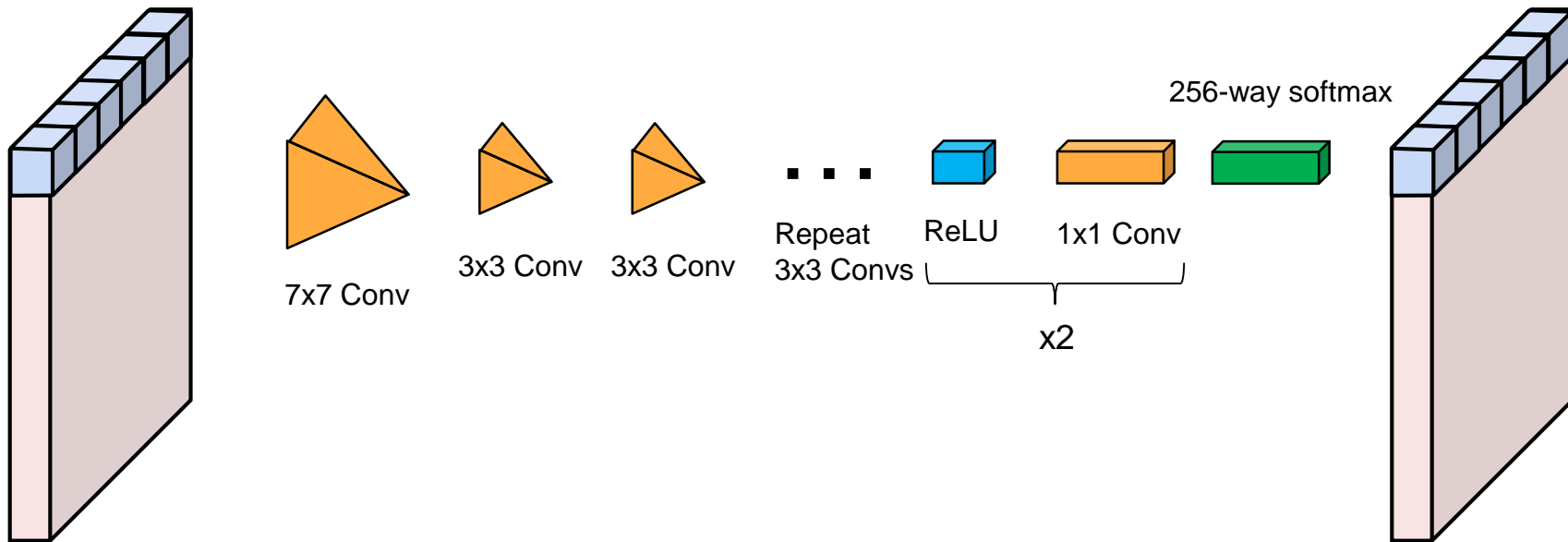
PixelCNN

Etc.



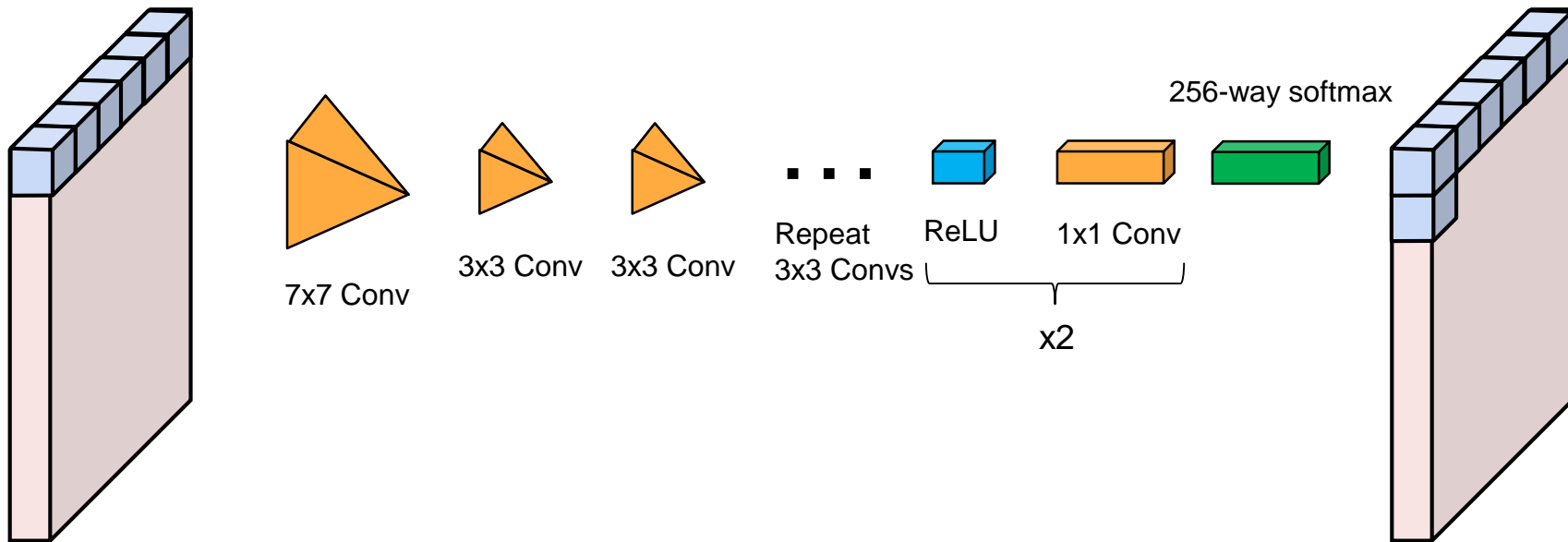
PixelCNN

After completing a row, we start on the next



PixelCNN

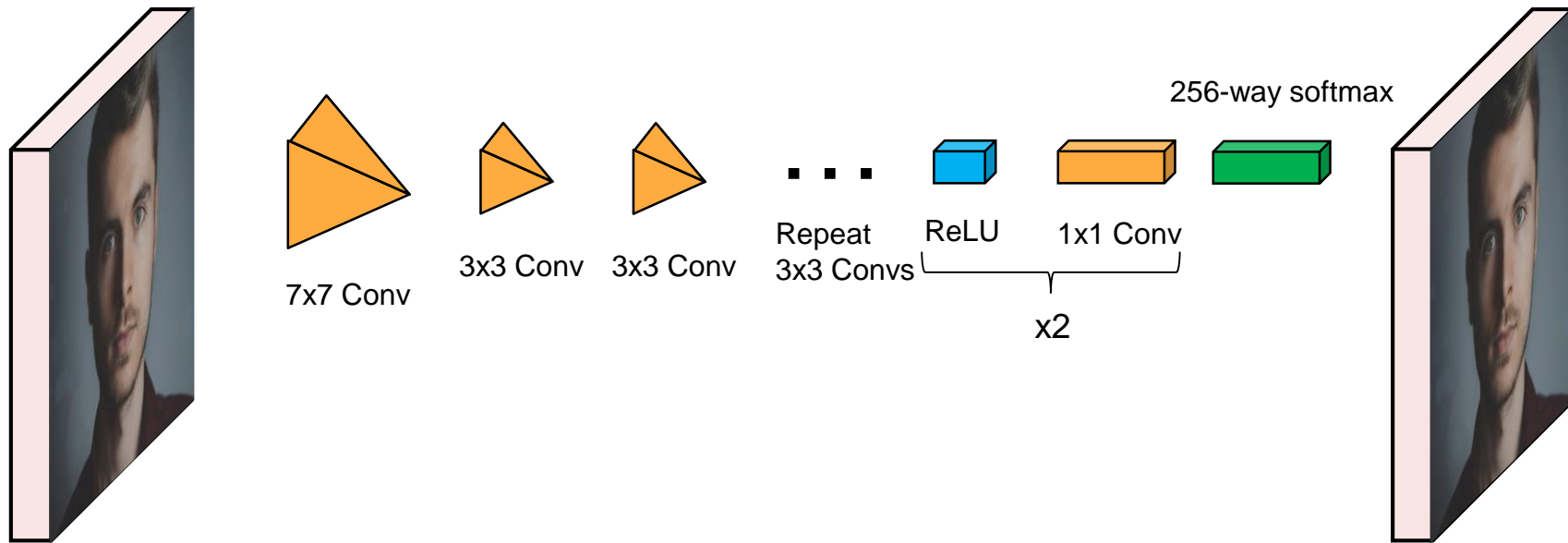
After completing a row, we start on the next



PixelCNN

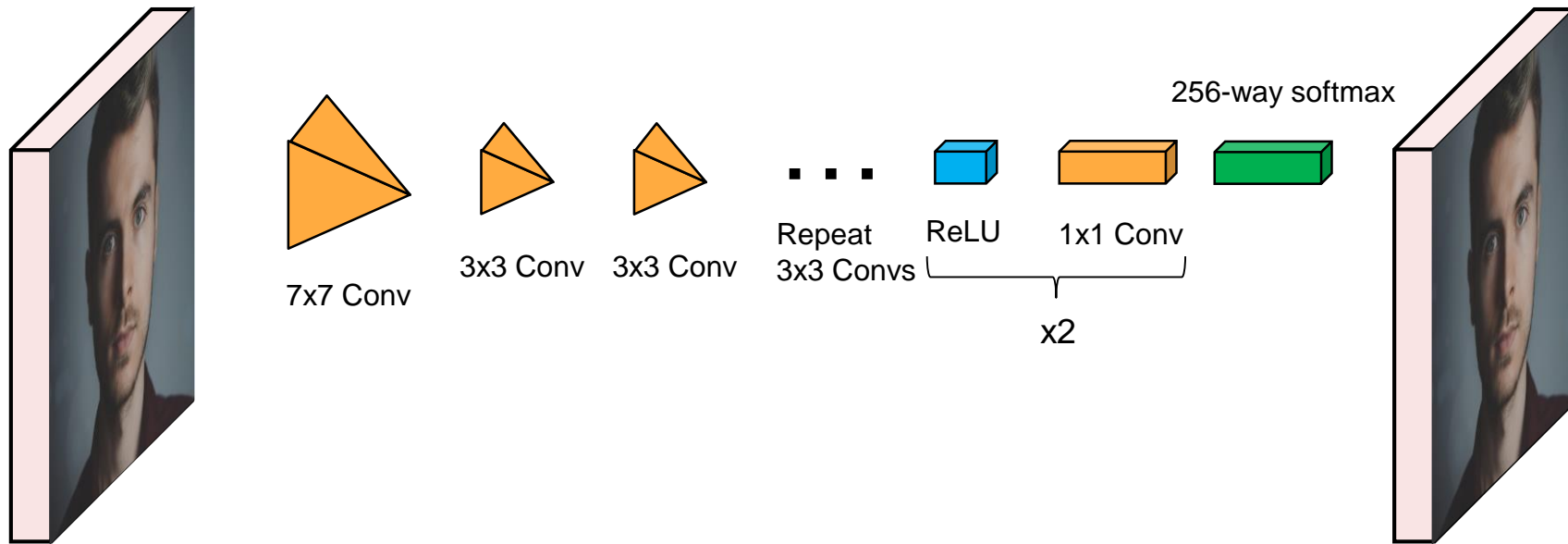
PixelCNN is very *slow at generating* images, because there is a pass through the entire network for each pixel.

But its very *fast to train* because there is no recurrence (only a single pass for the image).



PixelCNN

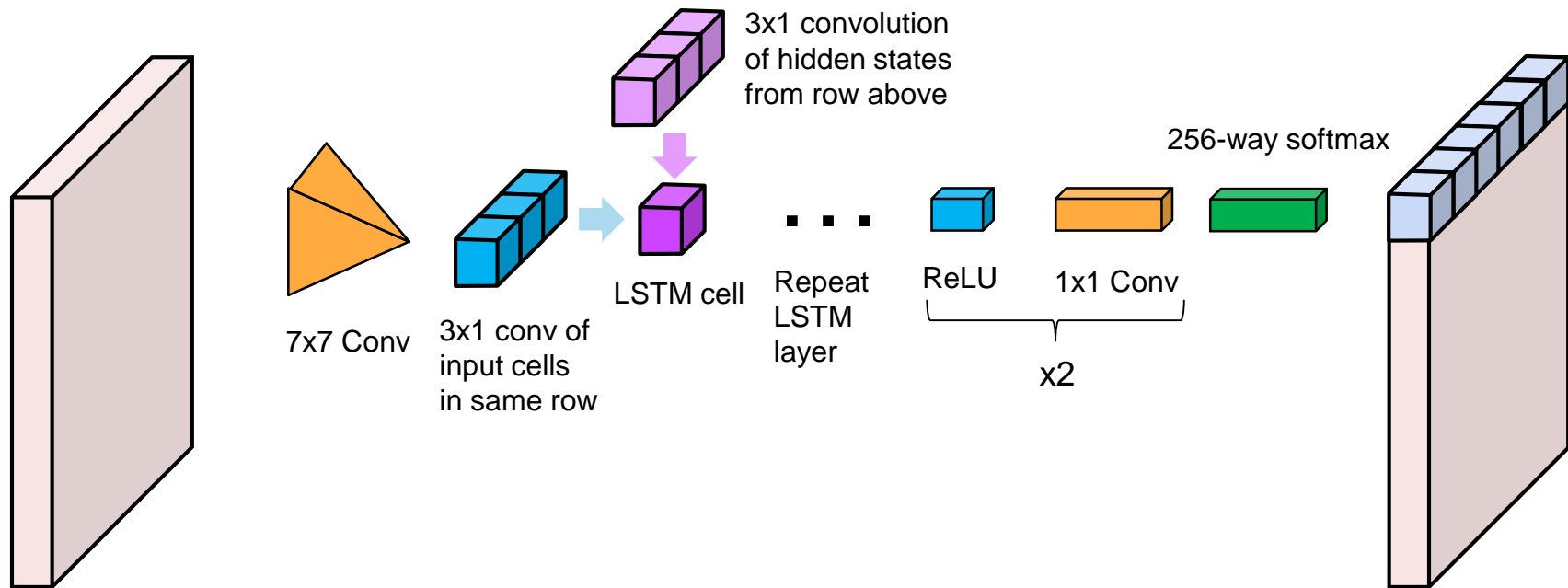
Note: during training the convolutions must be causally masked, i.e. they must ignore pixels at the same or later positions in the image generation order.



PixelRNN – Row LSTM

The PixelRNN uses recurrence instead of the 3x3 convolutions to allow long-range dependencies.

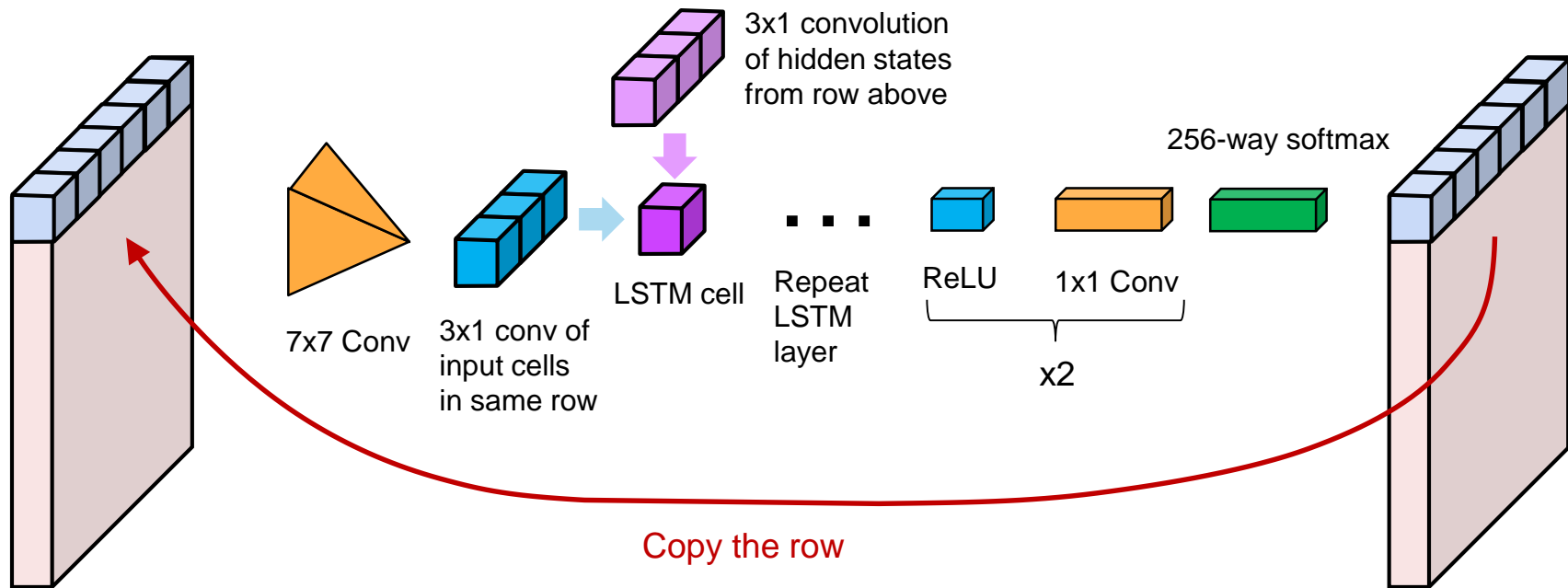
Its able to generate a full row of pixels in one pass.



PixelRNN – Row LSTM

The PixelRNN uses recurrence instead of the 3x3 convolutions to allow long-range dependencies.

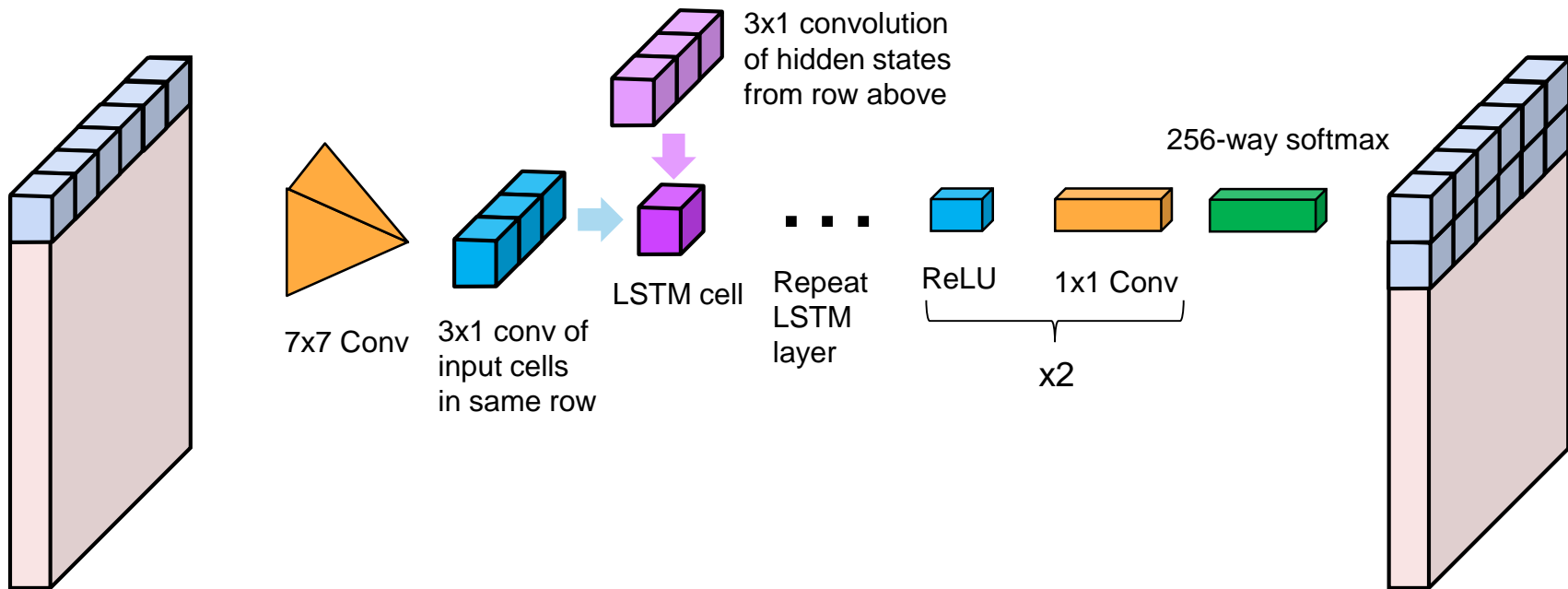
Its able to generate a full row of pixels in one pass.



PixelRNN – Row LSTM

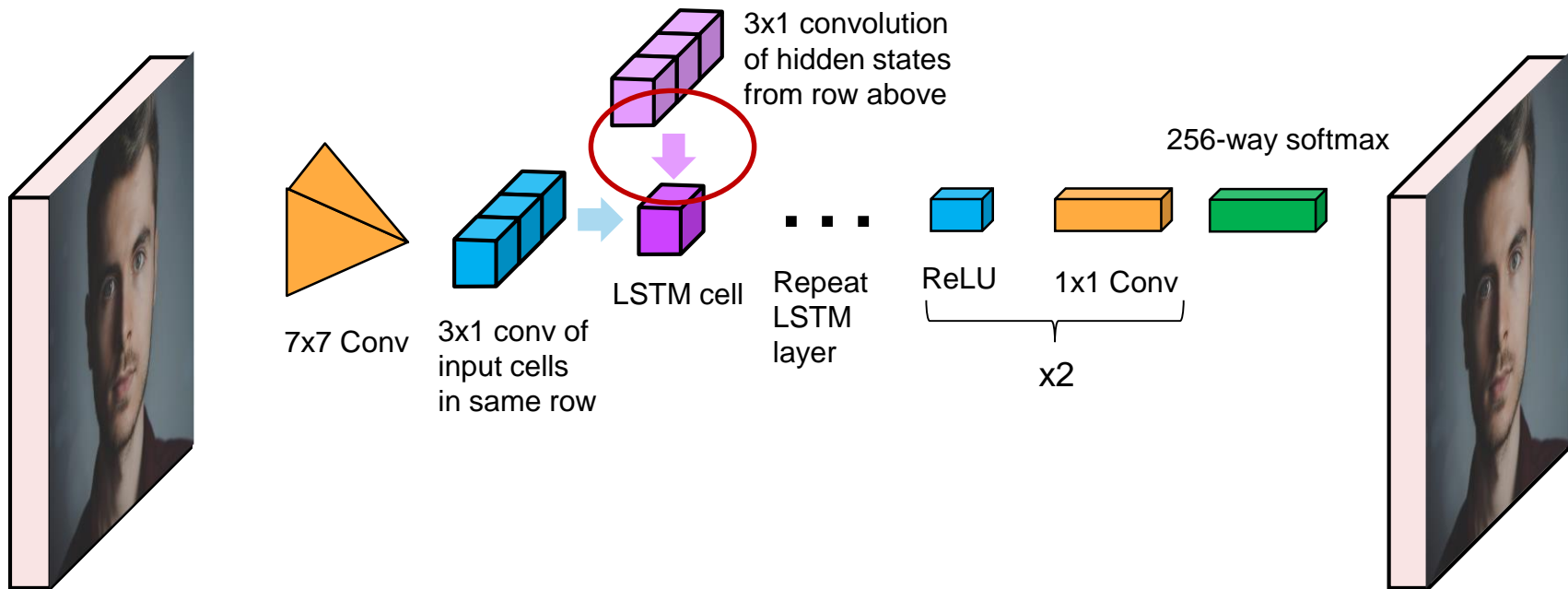
The PixelRNN uses recurrence instead of the 3x3 convolutions to allow long-range dependencies.

Its able to generate a full row of pixels in one pass.



PixelRNN

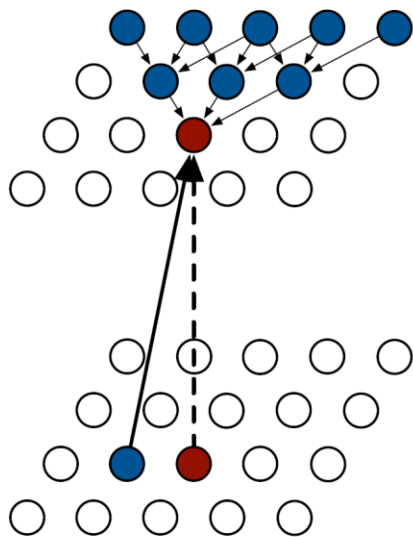
PixelRNN takes longer to train than PixelCNN because of the recurrent states, which must be calculated row by row.



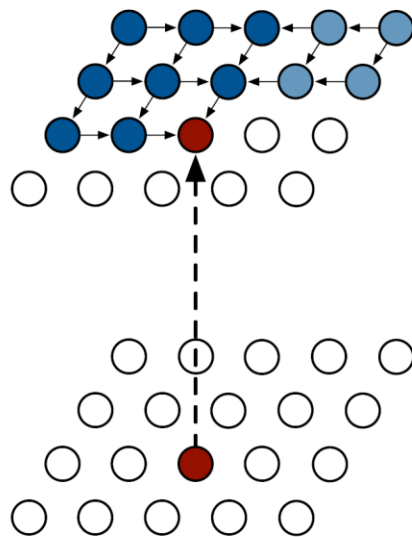
PixelRNN: Row LSTM vs. Diagonal BiLSTM

Row LSTM use recurrent state only from a triangular subset of previously-computed states.

The Diagonal BiLSTM uses all of them.

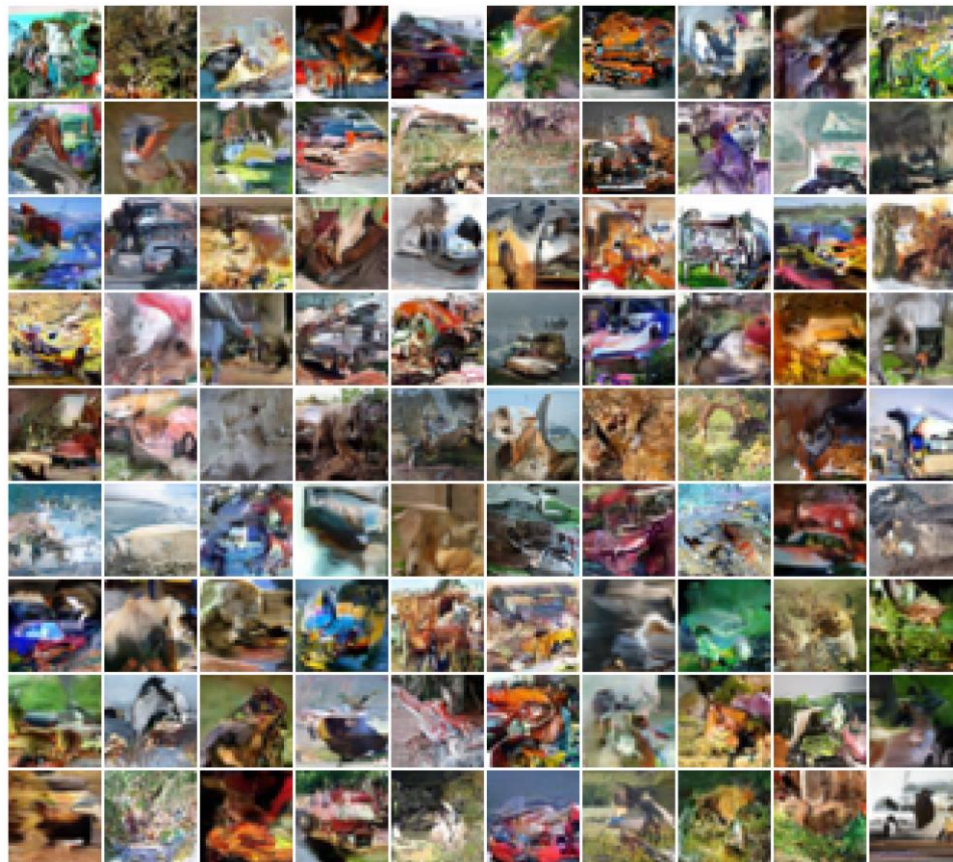


Row LSTM



Diagonal BiLSTM

CIFAR10-trained model (Diagonal BiLSTM)



ImageNet-trained model (Diagonal BiLSTM)

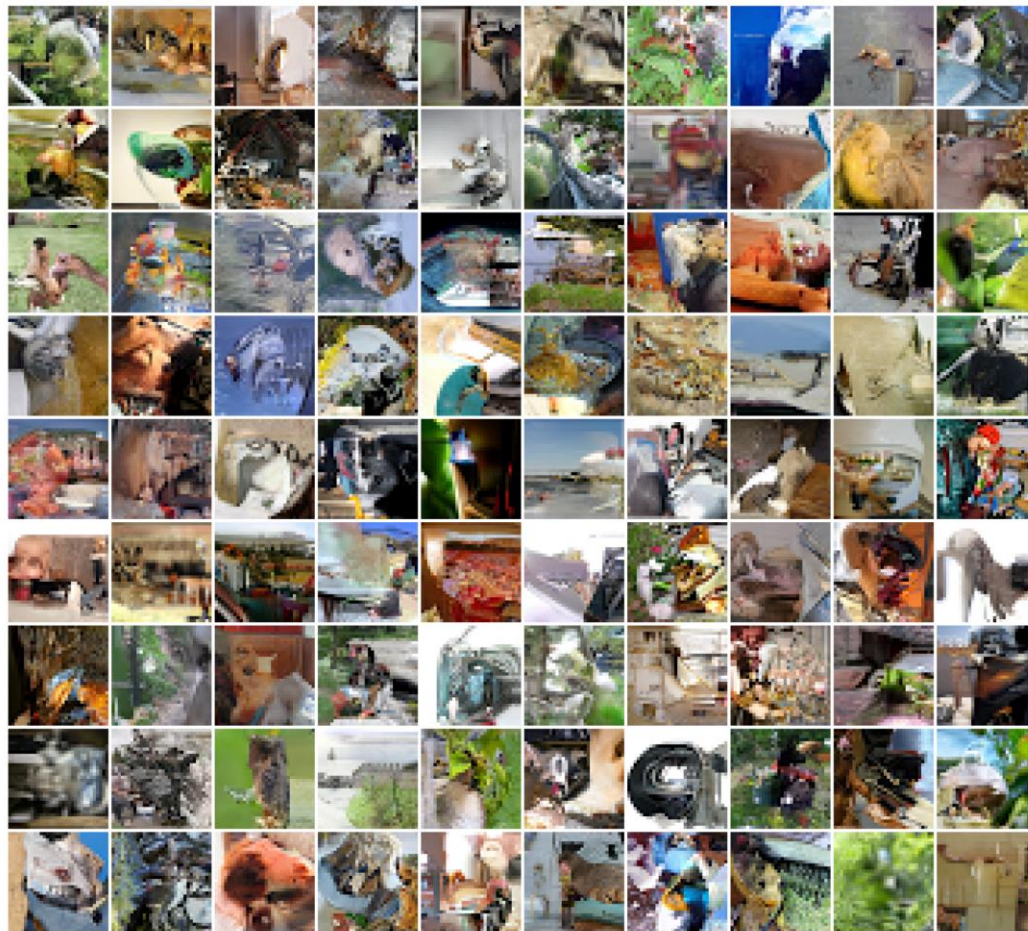
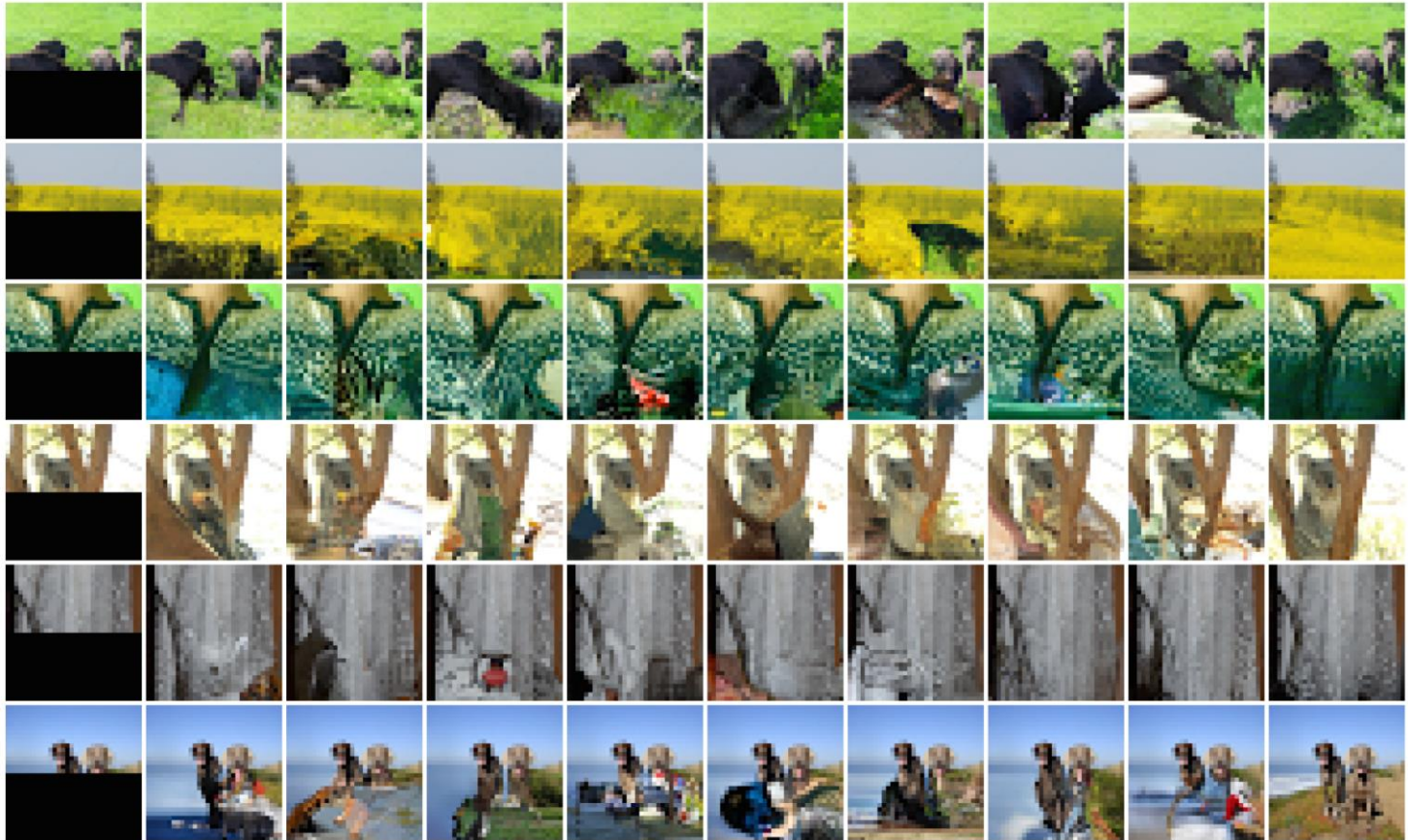


Image Completion Tasks



Performance

NLL is negative log likelihood – this is a full generative model so we can compute the probability density of real images. This table is for CIFAR10.

Model	NLL Test (Train)
Uniform Distribution:	8.00
Multivariate Gaussian:	4.70
NICE [1]:	4.48
Deep Diffusion [2]:	4.20
Deep GMMs [3]:	4.00
RIDE [4]:	3.47
PixelCNN:	3.14 (3.08)
Row LSTM:	3.07 (3.00)
Diagonal BiLSTM:	3.00 (2.93)

Gated PixelCNN

After the original PixelRNN/CNN paper, the PixelCNN design was improved by adding gating (replacing the ReLUs in PixelCNN).

This allowed conditional generation and improved performance over PixelRNN.

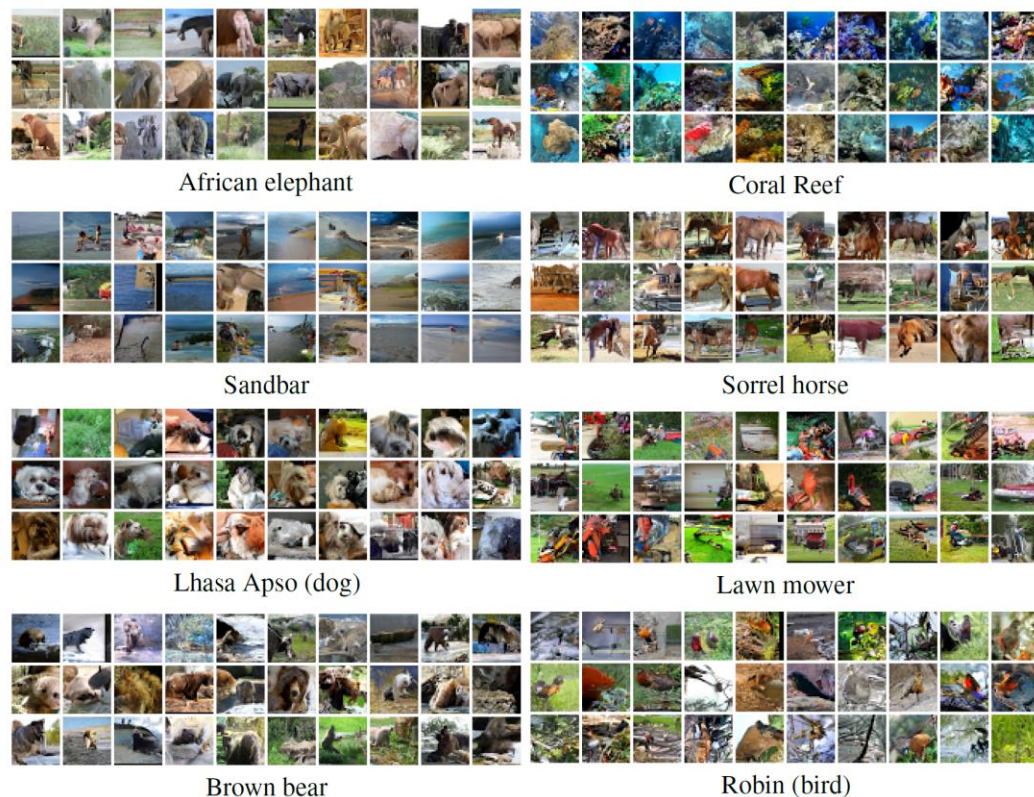


Figure 3: Class-Conditional samples from the Conditional PixelCNN.

Image Transformer

Similar autoregressive design to PixelCNN and PixelRNN:

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

but using a multi-headed attention network instead of a CNN/RNN.

The idea is to use a large influence region of the image (like PixelRNN) but without recurrence.

Like PixelRNN, it computes pixel representations in layers.

Image Transformer

The design is quite similar to other transformer implementations. We note that it uses dropout to help regularize the model.

The figure shows the design between two layers.

Position encodings are now 3-dimensional, for (x,y,c) where c is the color coordinate. Position embeddings are only added in the first layer.

The pixels m_i and q come from the previous data layer (q is the center pixel).

The pixel q' is generated for the next data layer.

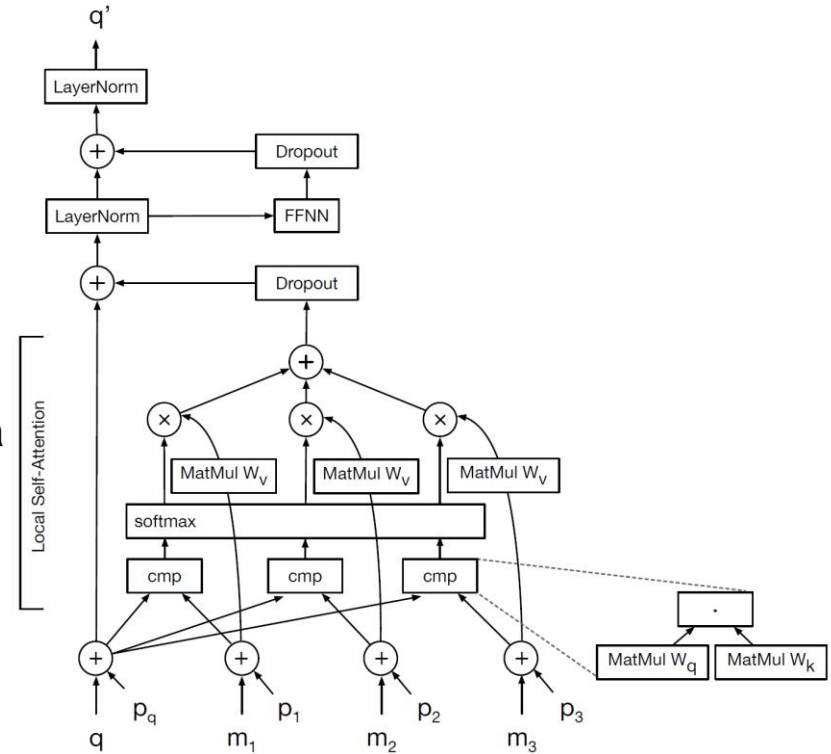
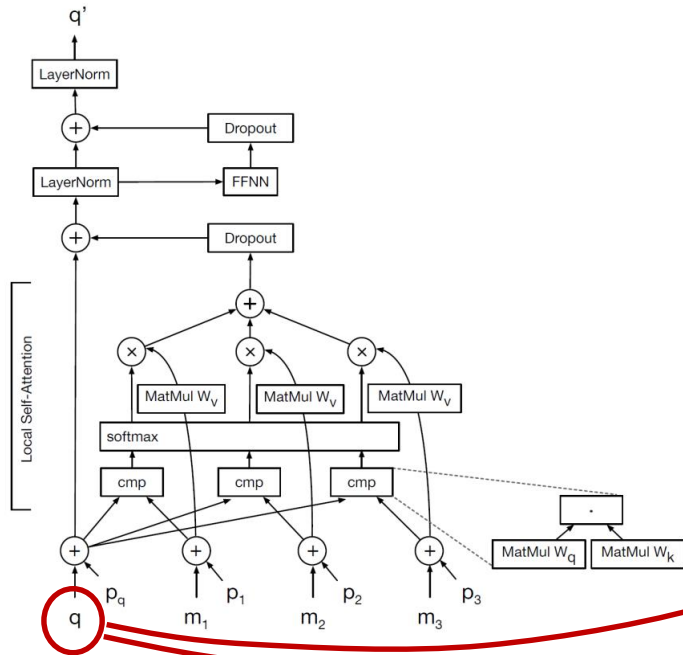


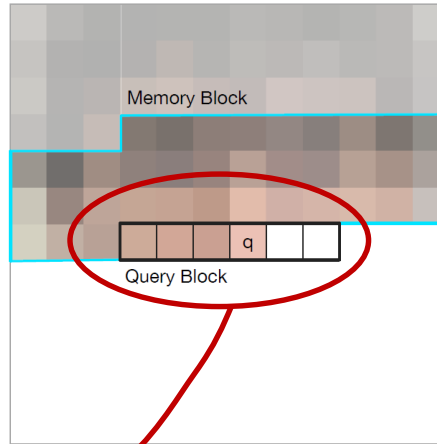
Image Transformer

Like the Row-LSTM, it generates blocks of pixels at a time.

The query blocks show the block of pixels to be generated.



Local 1D Attention



Local 2D Attention

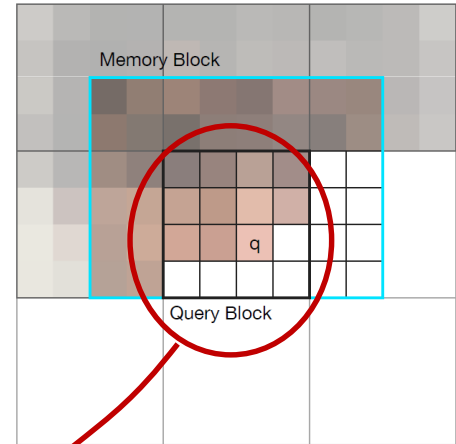


Image Transformer

Like the Row-LSTM, it generates blocks of pixels at a time.

The query blocks show the block of pixels to be generated

The memory blocks show the context from the previous layer used to generate it.

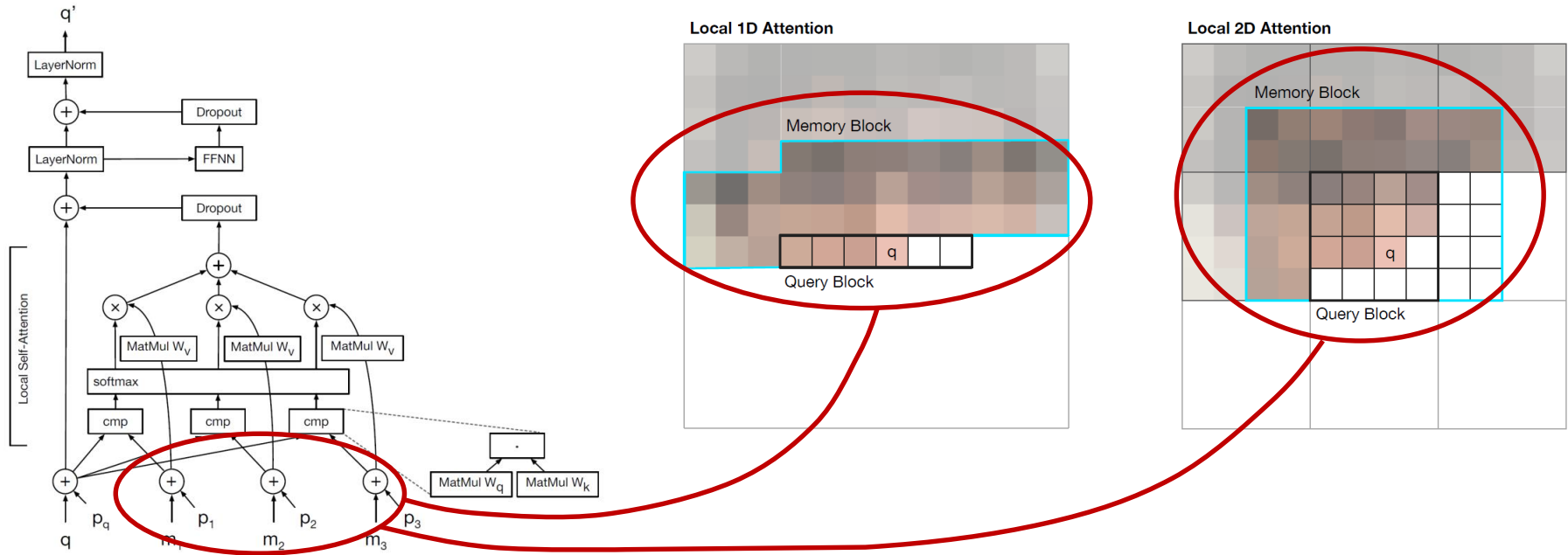
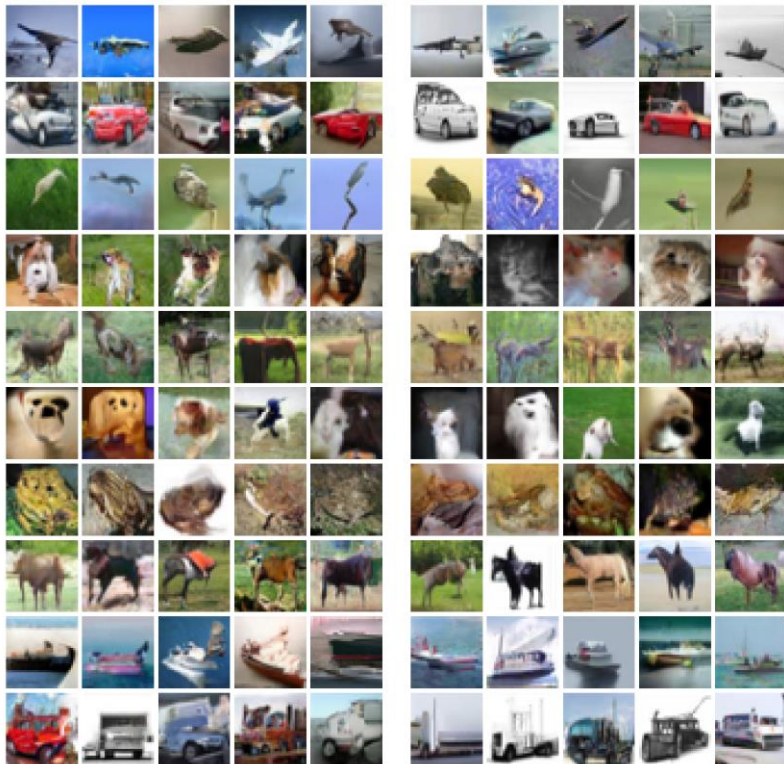


Image Transformer Results

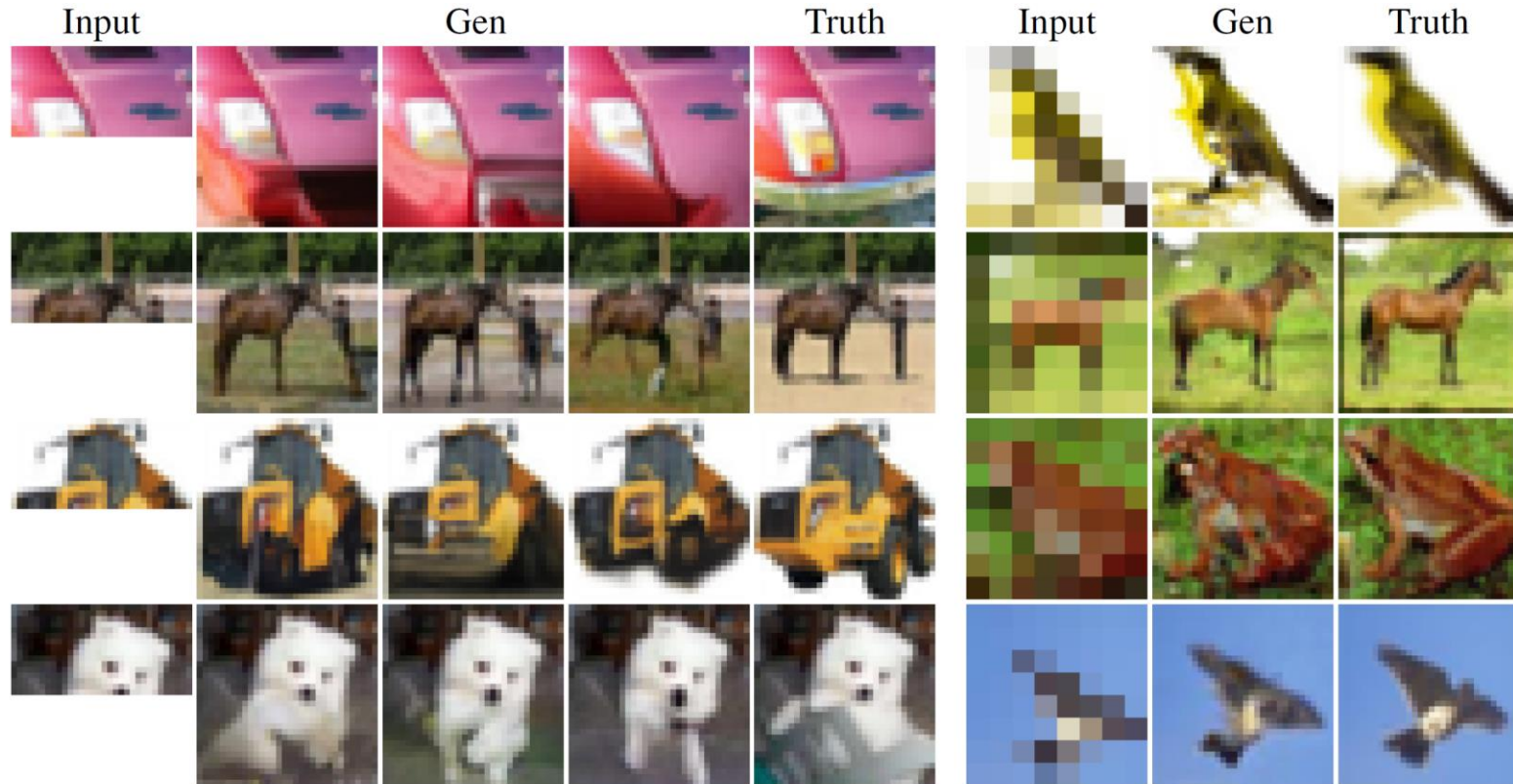


Synthetic CIFAR10 Images

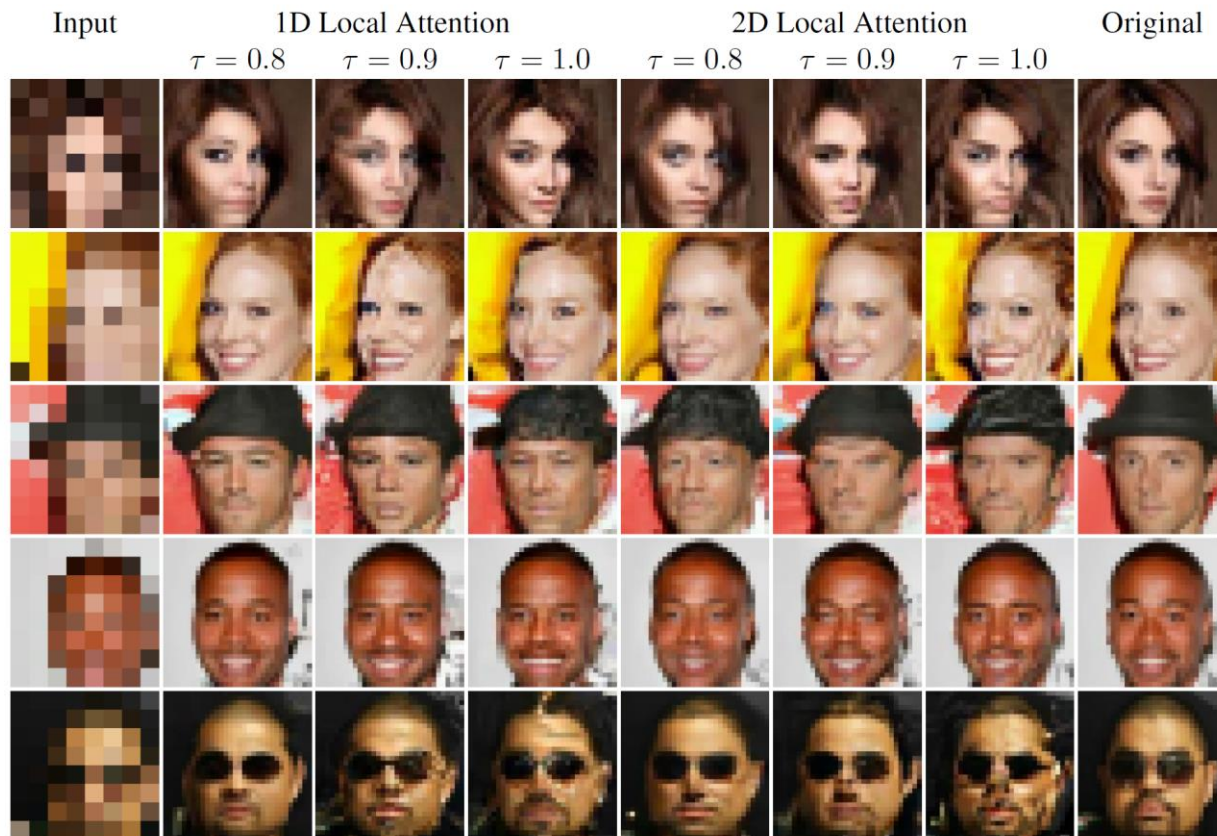
Model Type	<i>b</i> size	NLL	
		CIFAR-10 (Test)	ImageNet (Validation)
Pixel CNN	-	3.14	-
Row Pixel RNN	-	3.00	3.86
Gated Pixel CNN	-	3.03	3.83
Pixel CNN++	-	2.92	-
PixelSNAIL	-	2.85	3.80
Ours 1D local (8l, cat)	8	4.06	-
	16	3.47	-
	64	3.13	-
	256	2.99	-
Ours 1D local (cat)	256	2.90	3.77
Ours 1D local (dmol)	256	2.90	-

Note: PixelSNAIL is also a Transformer model

Results – Superresolution+Completion



Results – Comparing Attention Pattern



Takeaways

- VAEs construct an approximate encoder from a decoder.
- VAEs needs a decoder model that can compute density, and an encoder model that can sample.
- Reparameterization is normally used with VAEs to learn a distribution.
- Autoregressive models are full generative models (can compute density) by using an incremental factorization of the image density.
- Basic autoregressive models (PixelCNN and PixelRNN) use either CNNs or RNNs to influence a center pixel by neighboring pixels.
- The image transformer replaces the CNNs or RNNs with a multi-headed attention network.

