Unsupervised Learning: Explicit and Approximate Models

Earl Wong

Unsupervised Learning Models (Generative Models)

- Autoregressive (AR) (aka Fully Visible Belief Networks)
- Flow (aka Nonlinear Independent Component Analysis)
- Variational AutoEncoder (VAE)
- Generative Adversarial Network (GAN)

Regarding Generative Models

- We would like our generative model to produce output samples from a learned density: $p_{model}(x)$
- With AR and Flow, the models are explicit.
- With VAEs, the model is explicit, but approximate.
- With GANs, the model is implicit.

Warmup: A Simple Parametric Model

Consider the following parametric model: $p_{true}(x) = N\left(u_{true}, \sigma^2_{true}\right)$

We want to obtain the best estimates for u_{true} **and** σ^2_{true} **given samples** $x_1, x_2, \ldots, x_N$ **drawn from** $p_{true}(x)$

From maximum likelihood estimation theory, we can derive the following equations (via differentiation of the log likelihood function) to estimate the model parameters.

$$u_{estimate} = \frac{x_1 + x_2 + \ldots + x_N}{N}$$

$$\sigma^2_{estimate} = \left(\frac{1}{N-1}\right) \cdot \sum_{n=1}^{N} \left(x_n - u_{estimate}\right)^2$$

Warmup: A Simple Parametric Model

Now, let $p_{model}(x) = N\left(u_{estimate}, \sigma^2_{estimate}\right)$ **be our learned model.**

In the best case scenario, we would like $u_{estimate} = u_{true} \quad and \quad \sigma^2_{estimate} = \sigma^2_{true}$

We can then write:

$$\max_\theta \sum_{n=1}^{N} \log\ p_\theta\left(x^{(i)}\right), where\ p_\theta\ is\ N\left(u, \sigma^2\right)$$

Or, equivalently:

$$\theta^* = \arg\ \max_\theta E_{x \sim p_{true}} \log p_{model}(x|\theta)$$

Non Statistical Problem Formulation

- Sometimes, a good parametric statistical model may not be possible.

- Instead, we can also approximate $p_{true}(x)$ using a function f(x, W).

- Example: $$f(x, W) = w_0 + w_1 \cdot x + w_2 \cdot x^2 + w_3 \cdot x^3$$

$$where\ W = \{w_1, w_2, w_3\}$$

- Here, we wish to determine the weights W associated with the nonlinear function f.

An Extension to Images

- As humans, we can visualize a N=3 dimensional vector x = (x, y, z).

- For images, x is now a vector with dimension N = image_width * image_height.

- For images, we wish to approximate p_true(x) with f(x, W), where p_true(x) is a joint density function.

- One way to accomplish this, is to let f(x, W*) be a neural network. (W* denotes the weights that yield the best approximation to p_true(x))

Model: AutoRegressive (AR)

- Auto-regressive models originate from signal processing / time series analysis.
- An auto-regressive (AR) model computes the next output at time N using previously computed time outputs:

$$y_N = y_{N-1} + y_{N-2} + \ldots$$

- A moving average (MA) model computes the next time output at time N using previous inputs:

$$y_N = x_{N-1} + x_{N-2} + \ldots$$

- An auto-regressive moving average model (ARMA) computes the next time output at time N using previously computed time outputs (AR) and previous inputs (MA).

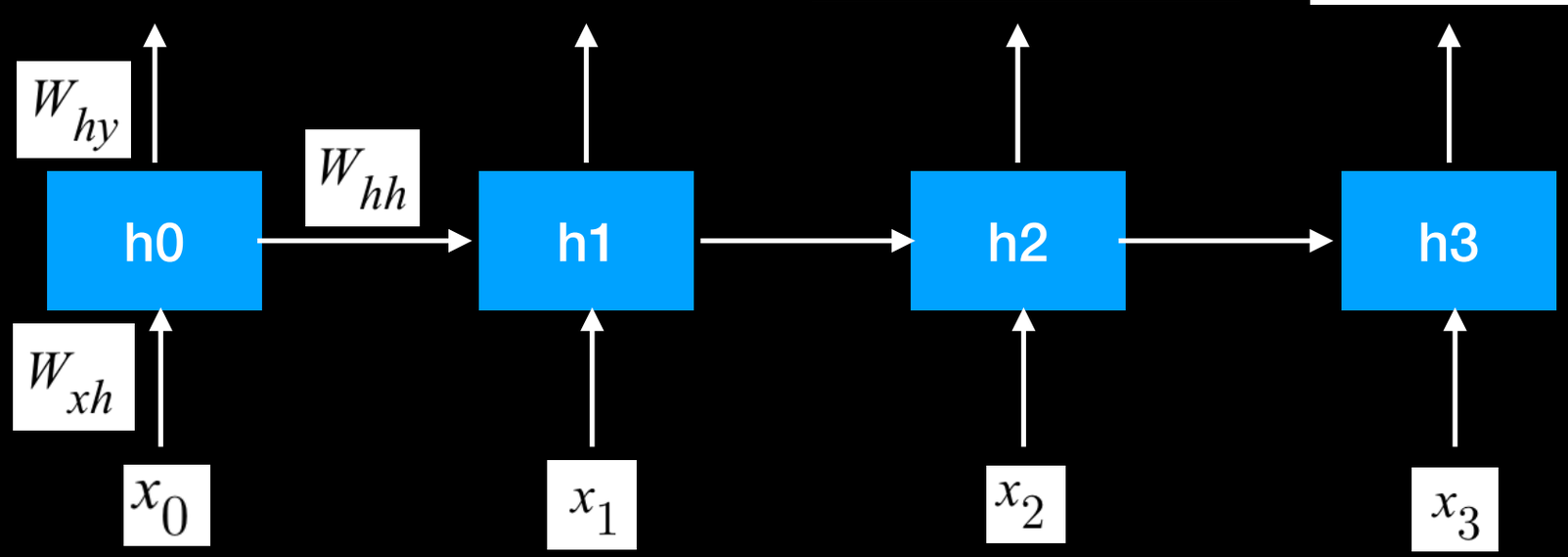
AR Model For Imaging

- A natural image can be modeled by a joint distribution p(x).
- Using the chain rule, a joint distribution can be rewritten as a product of conditionals.
- $p(x_{1}, x_{2}, \dots x_{N}) = p(x_{1}) * p(x_{2} | x_{1}) * p(x_{3} | x_{2}, x_{1}) * \dots * p(x_{N} | x_{1}, x_{2}, \dots x_{N-1})$
- A term such as $p(x_{3} | x_{2}, x_{1})$ indicates that future density estimate is dependent on the past inputs.
- This fact naturally leads to a recurrent neural network (RNN) formulation.

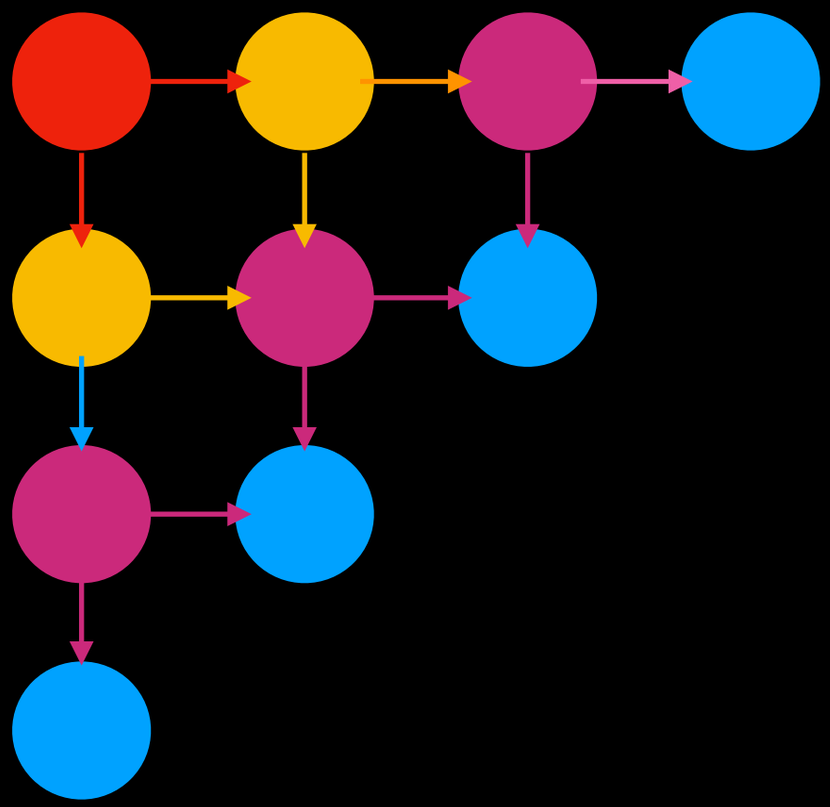
AR Model: Using an RNN

Example: $p\left(x_4 \mid x_0, x_1, x_2, x_3\right)$

$p\left(x_1 \middle| x_0\right)$ $p\left(x_2 \middle| x_0, x_1\right)$ $p\left(x_3 \middle| x_0, x_1, x_2\right)$ $p\left(x_4 \mid x_0, x_1, x_2, x_3\right)$



AR Model for Imaging = PixelRNN



At time t_1, the red pixels are used to generate the peach pixels.

At time t_2, the peach pixels are used to generate the purple pixels.

At time t_3, the purple pixels are used to generate the blue pixels.

Implied, is the fact that at time t_2, the purple pixels also depend on the (previously computed) red pixels.

Implied, is the fact that at time t_3, the blue pixels also depend on the (previously computed) peach and red pixels.

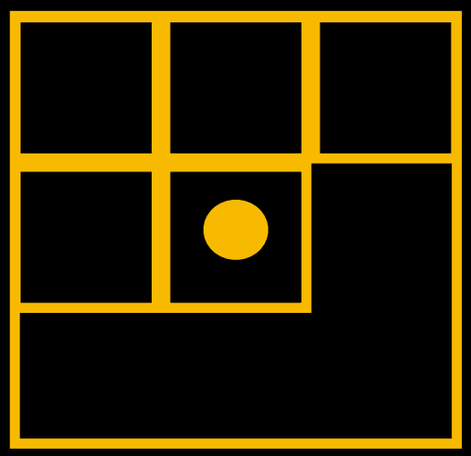
Image Completion Results using PixelRNN

AR Model Using a Masked Convolution

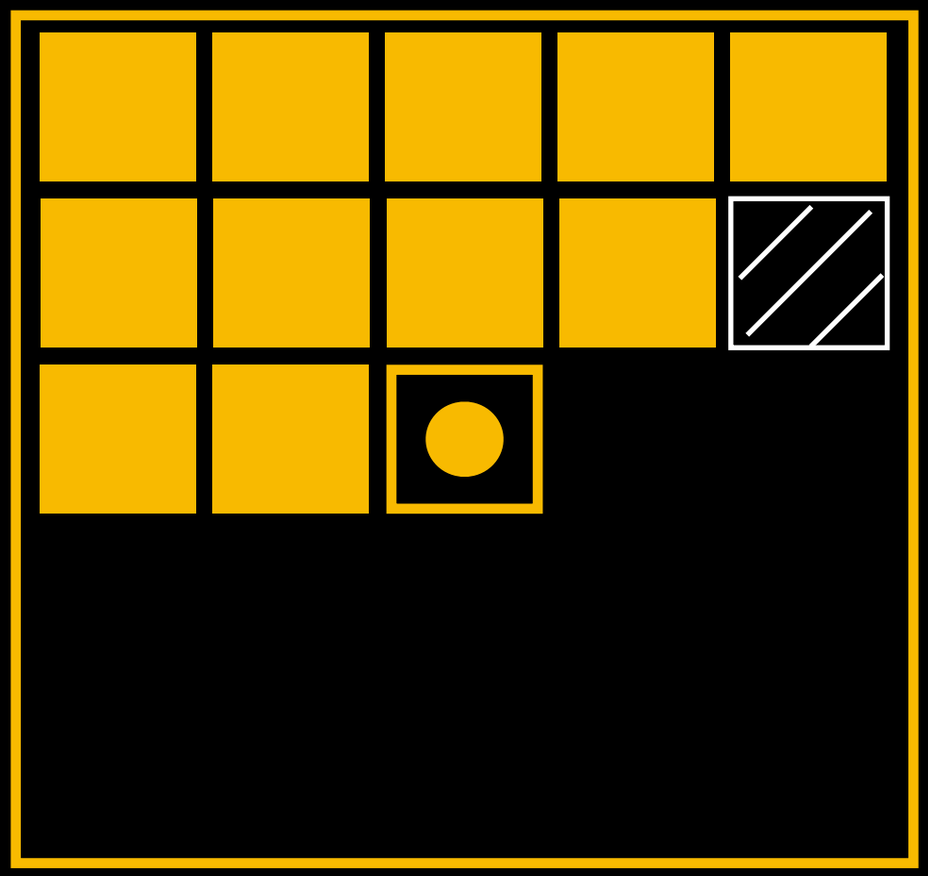
- The previous conditional relationship can also be approximated using a masked convolution.

- Recall the main condition: the present is calculated using the past.

- If we employ the following mask in a raster scan fashion, this condition is satisfied.



AR Model for Imaging = PixelCNN



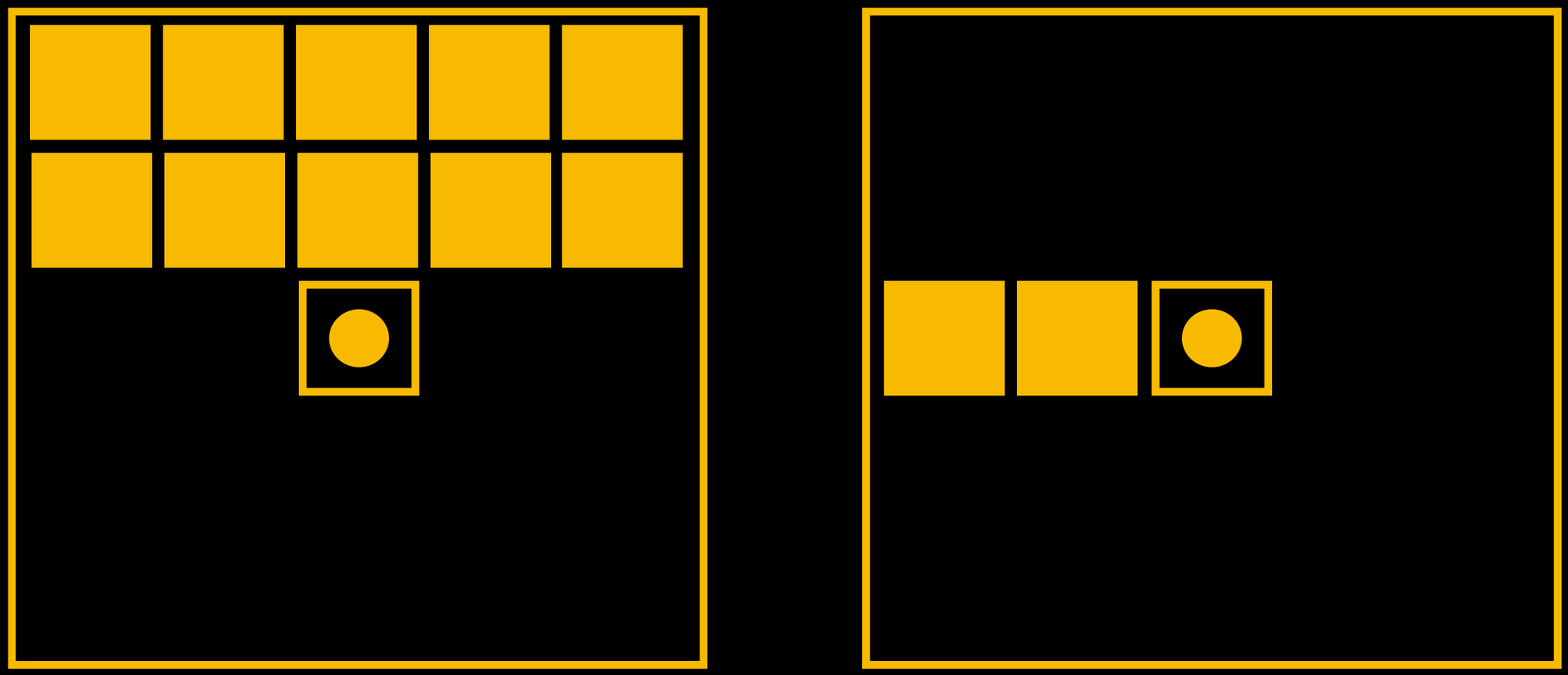
Blind Spot

Unfortunately, a blind spot results from this process.

In general, the size of the blind spot is a function of spatial location in the image.

The blind spot adversely affects the quality of the resulting output.

AR Model for Imaging = Gated PixelCNN

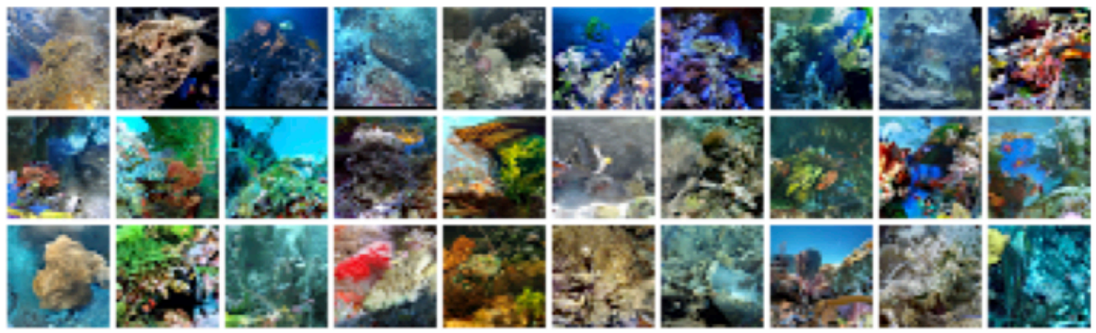


One possible workaround is to perform two separate convolutional passes.

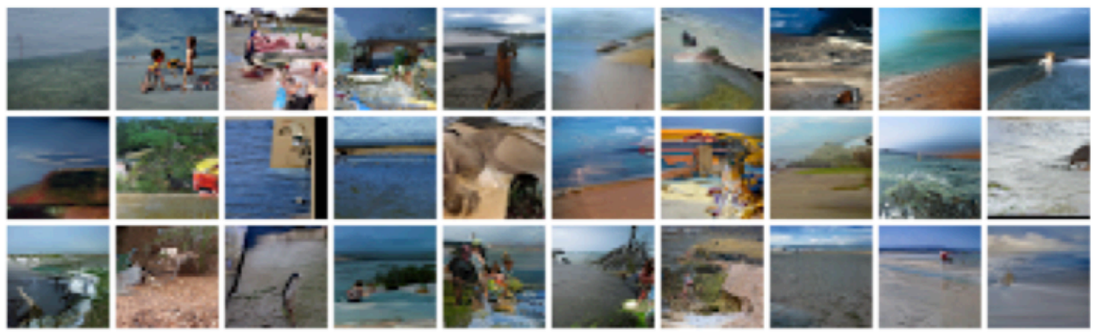
Both passes satisfy the requirement: “the present is computed using the past”

For each location, the two results combined using a weighted sum.

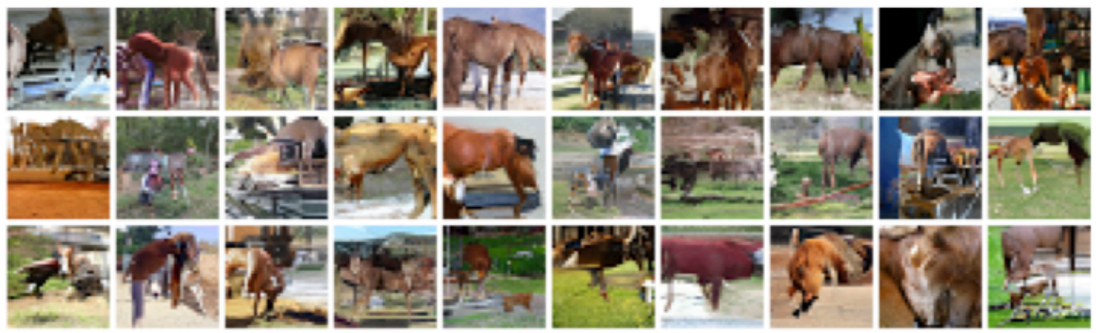
African elephant



Coral Reef



Sandbar



Sorrel horse

Lhasa Apso (dog)

Lawn mower

Brown bear

Robin (bird)

New images generated using PixelCNN.

Notes

- The one major downside of the AR approach is the output is generated sequentially = Very, Very slow.
- By definition and construction, this is the definition of “autoregressive”.
- PixelSNAIL adds causal convolutions and self attention to improve performance. (TODO: Include writeup.)

Model: Flow

Recall the following parametric model considered earlier: $p_{true}(x) = N\left(u_{true}, \sigma^2_{true}\right)$

We want to obtain the best estimates for u_{true} **and** σ^2_{true} **given samples** $x_1, x_2, \cdots, x_N$ **drawn from** $p_{true}(x)$

Now, suppose we wanted to transform this normal density function into a different density function. How would we do this?

From statistics, we know that we can transform the random variable X associated with a density function f(x) into a new random variable Y associated with a new density function g(y) using a function h, where Y = h(X).

h cannot be any arbitrary function though. Instead, h must be continuous, differentiable and invertible (=monotonic increasing or decreasin).

Model: Flow

Next, recall the cumulative distribution function (CDF). The CDF is just the integral of the probability distribution function (PDF).

From f(x) and g(y), we can compute the corresponding CDF's F(x) and G(y) by integrating the PDF's f(x) and g(y).

By definition, we know: $\frac{dF(x)}{dx} = f(x)$ *and* $\frac{dG(y)}{dy} = g(y)$

Now: $G(y) = \Pr(Y \leq y) = \Pr\big(h(X) \leq y\big)$

$$\Pr\Big(X \leq h^{-1}(y)\Big) = F\Big(h^{-1}(y)\Big)$$

And: $\frac{d\big(G(y)\big)}{dy} = \frac{dF\big(h^{-1}(y)\big)}{dy}$ **yields**

$$g(y) = f\Big(h^{-1}(y)\Big) \times \frac{d\big(h^{-1}(y)\big)}{dy}$$

Model: Flow

When applied to monotonically increasing and decreasing h functions, we can then write:

$$g(y) = f\left(h^{-1}(y)\right) \times \left| \frac{d\left(h^{-1}(y)\right)}{dy} \right|$$

$$g(y) = f(x) \times \left| \frac{dx}{dy} \right|$$

In general, let $X_1, X_2, ..., X_n$ **be random variables with joint pdf** $f\left(x_1, x_2, ..., x_n\right)$

Let $Y_i = t_i\left(X_1, X_2, ..., X_n\right), for\ i = 1, ..., N$

where the t's are one to one (invertible and differentiable).

We can then define a new joint density function $g\left(y_1, y_2, ..., y_n\right)$

Model: Flow

We can then write:

$$g(y_1, y_2, ..., y_n) = f(x_1, x_2, ..., x_n) \times |\det J|$$

Where J is an NxN matrix defined as:

$$J = \begin{bmatrix} \frac{\partial t_1^{-1}}{\partial y_1} & \cdots & \frac{\partial t_1^{-1}}{\partial y_n} \\ \vdots & & \vdots \\ \frac{\partial t_n^{-1}}{\partial y_1} & \cdots & \frac{\partial t_n^{-1}}{\partial y_n} \end{bmatrix}.$$

Swapping y's for x's, we can also write:

$$f(x_1, x_2, ..., x_n) = g(y_1, y_2, ..., y_n) \times \left|\det J_{reverse}\right| \quad \textbf{where}$$

∂t_i^{-1} *becomes* ∂t_i

∂y_i *becomes* ∂x_i

Model: Flow

- When we apply a flow to images, we transform the original joint density function into a new joint density function - usually uniformly or normally distributed.

- We denote this new joint density as $p(z)$

- The purpose of this transformation is to find a meaningful latent space representation, defined by $p(z)$

- Recall, the objective for maximum likelihood estimation:

$$\theta^* = \arg\max_\theta E_{x \sim p_{true}} \log p_{model}(x|\theta)$$

Model: Flow

If we substitute the recently derived result, we then have:

$$\arg \max_{\theta} E \left[\left[\log g\left(y_1, y_2, ..., y_n\right) + \log \left| \det J_{reverse} \right| \right] \right]$$

We observe that the “heavy lifting” resides in computing the determinant of the Jacobian.

In practice, we design our process so that the Jacobian matrix is diagonal or upper/lower triangular.

In addition, the t transformations also need to be differentiable and invertible.

Next, we look at two Flow based approaches: Real NVP and Glow.

Real NVP

Design Criteria

- We want to learn highly non-linear models in high dimensional space for continuous data using maximum likelihood.
- We want to define a powerful class of bijective functions, which allows us to perform exact and tractable density estimation and inference.
- We want to design our bijective functions so that the corresponding Jacobian matrix is diagonal, upper triangular or has a fast LU decomposition.
- We want to be able to compose our simple bijection functions, to create a more complex bijective function, with similar Jacobian matrix properties.
- Real NVP = real valued non-volume preserving transformations meets these design criteria objectives.

Real NVP: The Crux

Apply the following transformation to the data.

Given a D dimensional input x and $d < D$, the output y of an affine coupling layer follows the equations

$$y_{1:d} = x_{1:d} \tag{4}$$

$$y_{d+1:D} = x_{d+1:D} \odot \exp\left(s(x_{1:d})\right) + t(x_{1:d}), \tag{5}$$

where s and t stand for scale and translation. and are functions from $R^d \mapsto R^{D-d}$, and $\odot$ is the Hadamard product or element-v

This transformation produces the corresponding lower triangular Jacobian matrix.

The Jacobian of this transformation is

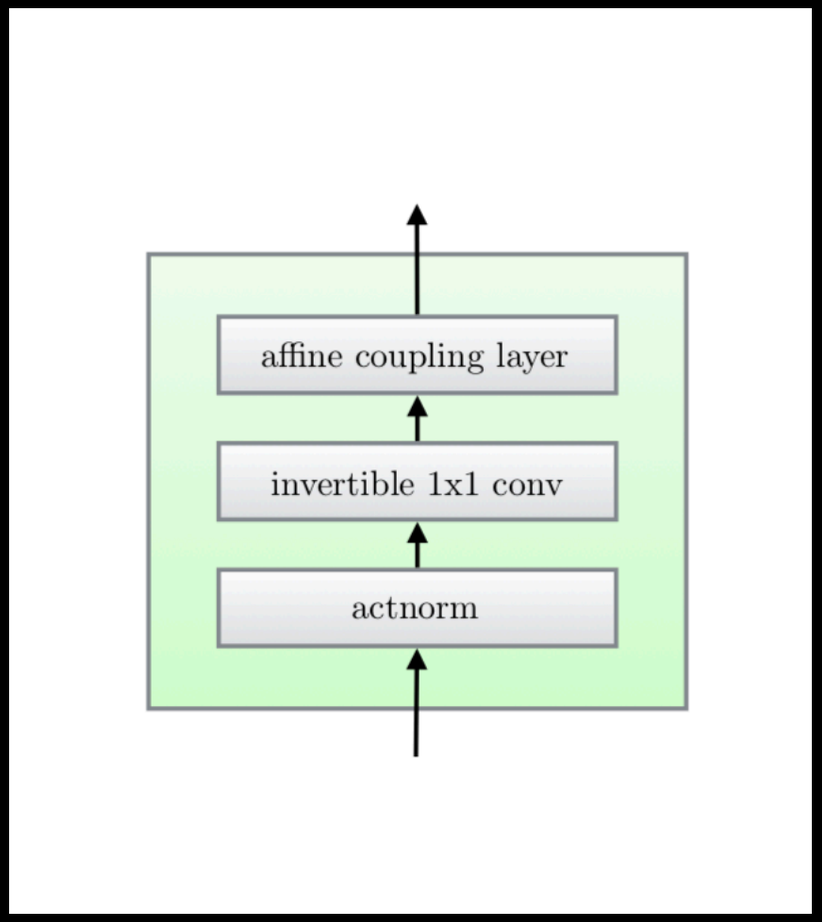
$$\frac{\partial y}{\partial x^T} = \begin{bmatrix} \mathbb{I}_d & 0 \\ \frac{\partial y_{d+1:D}}{\partial x_{1:d}^T} & \operatorname{diag}\left(\exp\left[s\left(x_{1:d}\right)\right]\right) \end{bmatrix}$$

Its determinant is:

$$\exp\left[\sum_j s\left(x_{1:d}\right)_j\right]$$

Sample Flow output of Real NVP, trained using CelebA

Glow



Glow modifies the Real NVP proposal
by adding an invertible 1x1 convolution and an
ActNorm processing.

The 1x1 convolution permutes / reorders
the channels in a given layer of the network.

ActNorm is used as a replacement for batch norm,
since the batch sizes used in Glow are small.

ActNorm takes each activation layer and transforms
the values in that layer to a zero mean, unit variance
distribution.

This fundamental building block is then cascaded
multiple times, and at multiple scales, producing
improved generator results (relative to Real NVP).

Random samples from the Glow model

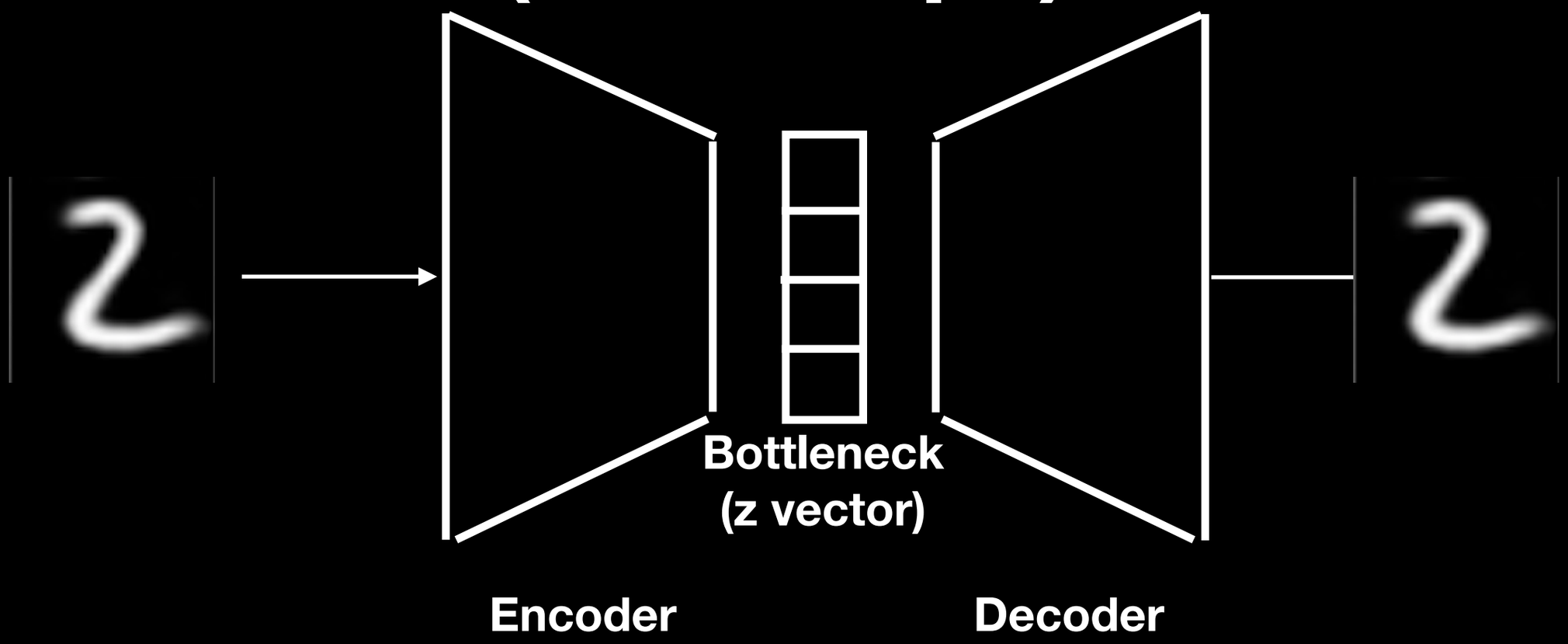
Notes

- Flow++ (TODO: Include writeup.)

Model: VAE

- VAE's define explicit, but intractable density functions.
- Because of this, we work with a lower bound of the density function.
- We first define an auto encoder, and then describe how the auto encoder becomes "variational".

Model: AutoEncoder (Concept)



The objective of the auto encoder is to learn non-trivial representations for identical input/ output pairs.

The auto encoder is comprised of 3 components: the encoder, the bottleneck, the decoder

In the ideal case, the bottleneck (vector) performs “lossless compression” of the input.

The auto encoder is trained using MSE loss and backpropagation.

Variational AutoEncoder

- As shown, the bottleneck = z vector is deterministic.
- In general, the z vector functions in a similar fashion as the “4096” vector in the early image recognition backbones. (i.e. Just throw away the decoder and use the encoder.)
- Unfortunately, there are no known “shipping systems” that utilize this auto encoder and z vector setup, learned from unlabeled data.
- Now, suppose we allowed various components in our auto encoder to become probabilistic (=variational).
- How would we do this? (Auto Encoding Variational Bayes)

Variational AutoEncoder

Consider Bayes Rule:

$$P(Y|X) = \frac{P(X|Y)P(Y)}{P(X)}$$

$$P(X) = \frac{P(X|Y)P(Y)}{P(Y|X)}$$

Converting this to our auto encoder notation, we obtain:

$$p_\theta(x) = \frac{p_\theta(x|z)p(z)}{p_\theta(z|x)}$$

Synonymous with our previous goals, we want to find the best $p_\theta(x)$ **, given our training data.**

Variational AutoEncoder

By construction, we can constrain $p(z)$ to be gaussian.

In addition, we can train a decoder network to produce a statistical output consisting of a mean vector output and a variance vector output to replicate the distribution: $p_\theta(x|z)$

In addition, we also want an encoder network to reproduce: $p_\theta(z|x)$

We accomplish the latter, by training a separate network: $q_\psi(z\ x)$ since the original term in red is intractable.

We can then re-express Bayes equation as:

$$p_\theta(x) = \frac{p_\theta(x|z)p(z)\ q_\psi(z|x)}{p_\theta(z|x)\ q_\psi(z|x)}$$

Which then yields:

$$\log p_\theta(x) = \log p_\theta(x|z) - \log \frac{q_\psi(z|x)}{p(z)} + \log \frac{q_\psi(z|x)}{p_\theta(z|x)}$$

Variational AutoEncoder

Observing that:

$$\log p_\theta(x|z) = E_{z\sim q_\psi(z|x)} \log p_\theta(x)$$

We can write:

$$\log p_\theta(x|z) = E_{z\sim q_\psi(z|x)} \log p_\theta(x) = E_{z\sim q_\psi(z|x)} \log p_\theta(x|z) - E_{z\sim q_\psi(z|x)} \log \frac{q_\psi(z|x)}{p(z)} + E_{z\sim q_\psi(z|x)} \log \frac{q_\psi(z|x)}{p_\theta(z|x)}$$

$$\log p_\theta(x|z) = E_{z\sim q_\psi(z|x)} \log p_\theta(x) = E_{z\sim q_\psi(z|x)} \log p_\theta(x|z) - D_{KL}\left(q_\psi(z|x)||p(z)\right) + D_{KL}\left(q_\psi(z|x)||p_\theta(z|x)\right)$$

Since the KL divergence is always positive, we can write the following Variational lower bound (VLB), eliminating the need to compute the intractable term.

$$\log p_\theta(x\ z) = E_{z\sim q_\psi(z|x)} \log p_\theta(x) >= E_{z\sim q_\psi(z|x)} \log p_\theta(x|z) - D_{KL}\left(q_\psi(z|x)||p(z)\right)$$

Reconstruction Loss Term **Regularization Term**

Variational AutoEncoder

We now have the following two networks. The networks are jointly trained, using the VLB constraint.

Encoder

$$q_{\psi}(z|x) = N\left(u_{z|x}, \Sigma_{z|x}\right)$$

Neural Network

The output from q is used to set the mean and variances for A multivariate gaussian.

Decoder = Generative Model

$$p_{\theta}(x|z) = N\left(u_{x|z}, \Sigma_{x|z}\right)$$

Neural Network

1) **We pass the training data through the encoder, enforcing the KL divergence constraint in the VLB (second term of VLB).**
2) **The encoder outputs a mean vector and a variance vector.**
3) **These values are then used to "seed" a multivariate Gaussian, p(z).**
4) **Next, we generate sample codes z from p(z).**
5) **The decoder is then designed to accurately reconstruct images similar to the training input, by enforcing the first term of the VLB.**

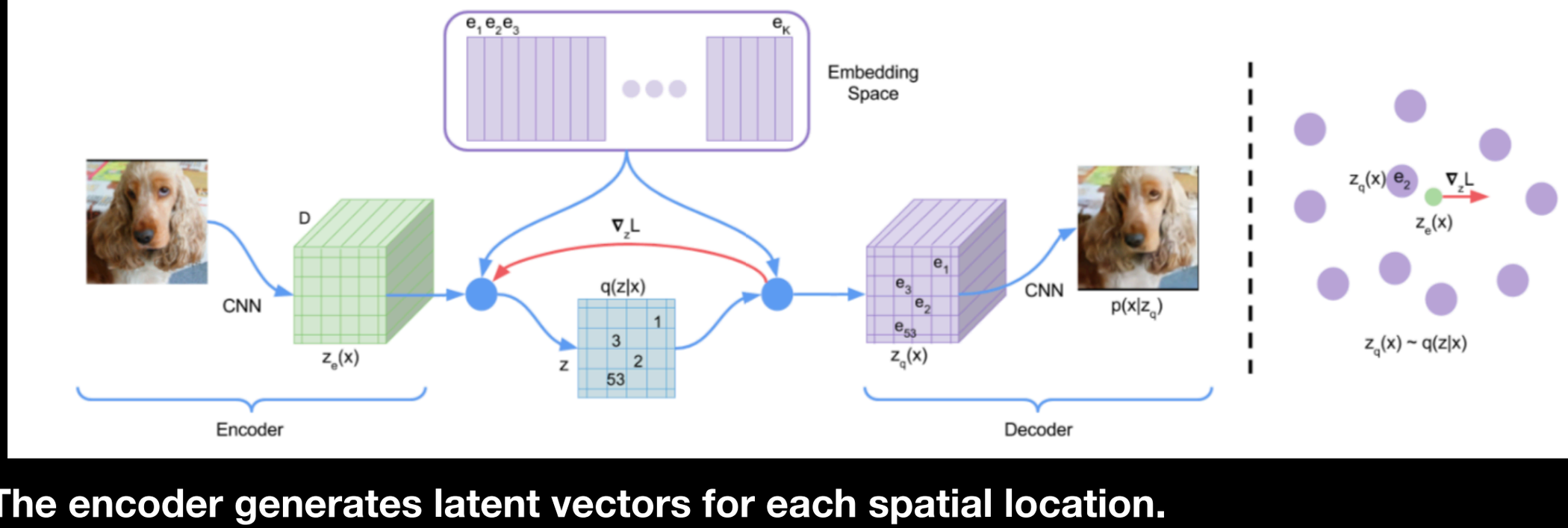
If this is unclear, this is what is being done in the code during the forward pass:

```
# The encoder network outputs two separate vectors,
# one for mean, and one for standard deviation.

q_mean, q_stddev = self.encode(images)

# This vector is used to seed a multi-variate normal
# distribution
q_dist = Normal(q_mean, q_stddev)

# We then extract a sample from this distribution,
# and use the sample as input to our decoder/generator.
z_sample = q_dist.rsample()
```

Extensions

- Standalone VAE results were not great.
- One postulated reason, was attributed to the fact that the resulting z-space was continuous, thereby allowing undesirable mixing.
- Instead, if the space could be constrained to specific output clusters, more meaningful samples could probably be generated. (i.e. A forced disentanglement.)
- This was the motivation behind VQ-VAE.

VQ-VAE



The encoder generates latent vectors for each spatial location.

These latent vectors are then matched to the entries (e_i) in a codebook/VQ table.

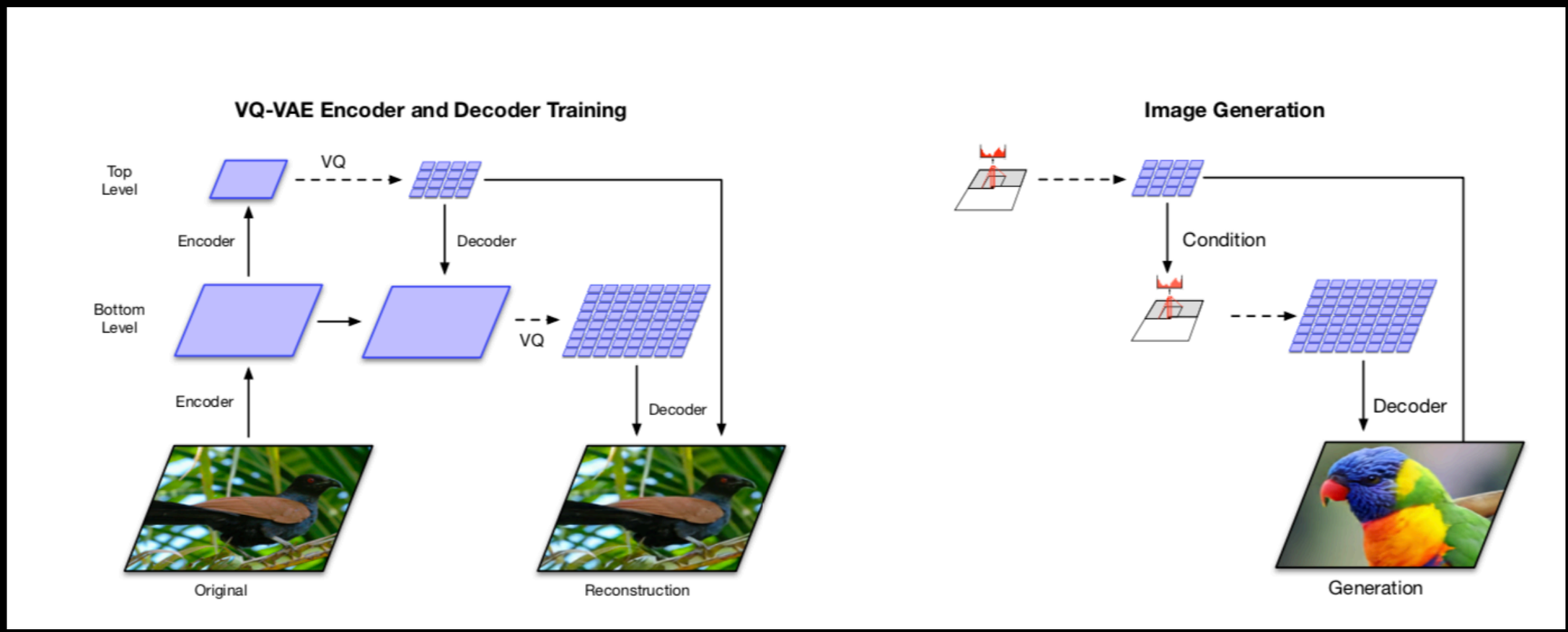
A new output volume is created, consisting of the discrete embedding vectors.

The output volume is then passed to the decoder.

The decoder then tries to recreate a meaningful output, using the reconstruction loss.

In addition, the output loss is also back propagated to the encoder network, forcing the network to produce outputs corresponding to latent vectors with more meaningful embeddings.

VQ-VAE2



In VQ-VAE2, hierarchical encoders (consisting of deep neural networks) are also employed.

During training, each encoder produces a latent map at a given resolution = hierarchical latent space.

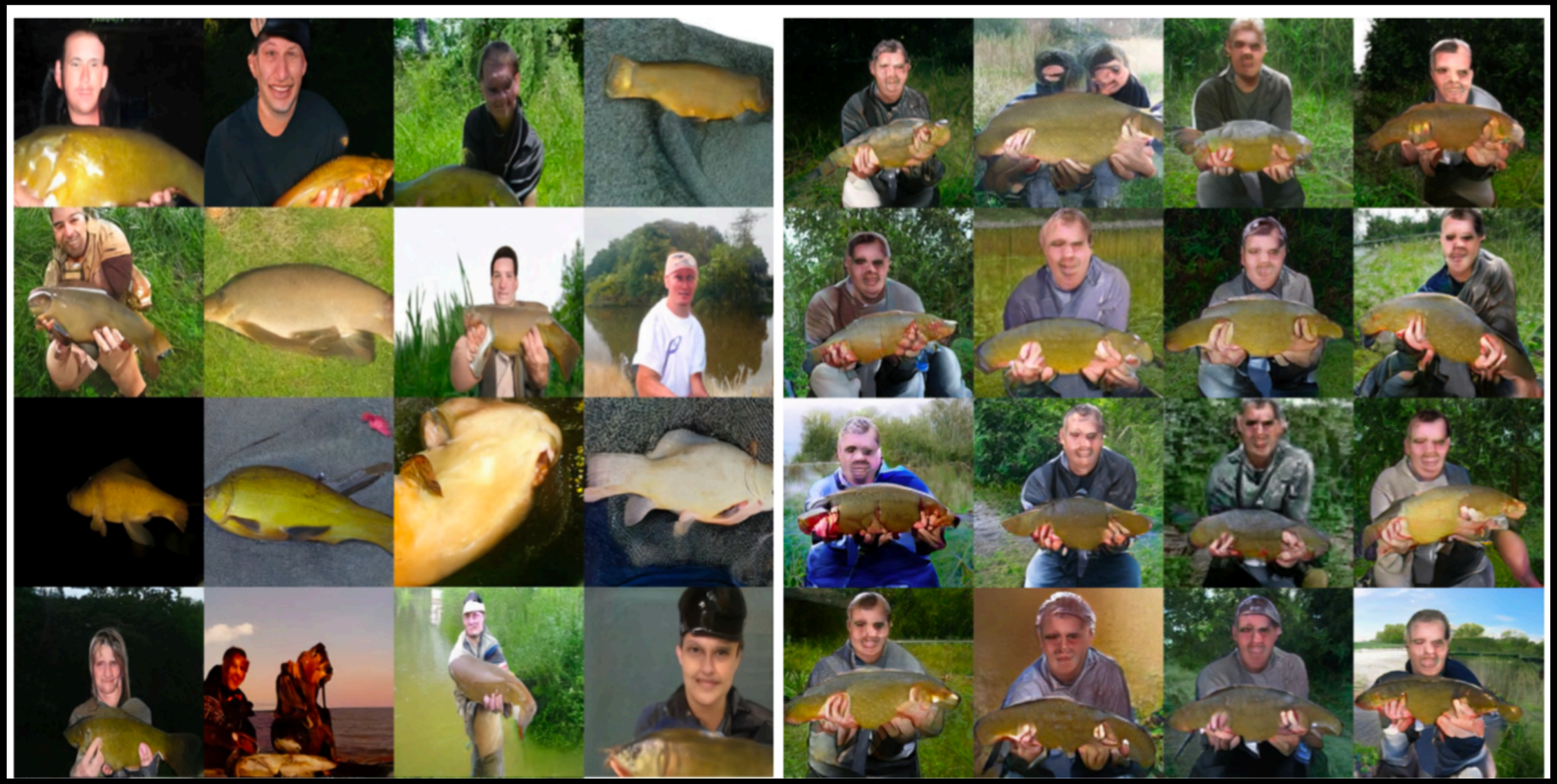
For each latent map, the best VQ vector is then selected from the codebook associated with that resolution.

The decoder then attempts to reconstruct the image, only now, using VQ vectors from different resolutions.

For generation, the embedding volumes are now computed using PixelCNN. The computed embeddings are then sent to the decoder to produce the resulting output image.

VQ-VAE2 (256 x256)

BigGAN



VQ-VAE2 generated class conditional images have greater diversity than BigGAN.

256
x
256
Images

Because of the PixelCNN inference architecture, long range dependencies (such as eye color) are respected using VQ-VAE2.

Because of the discrete embedding space, the class conditional generator consistently produced meaningful samples.

References

- PixelRNN, Oord et. al., arXiv January 2016
- Conditional Image Generation with PixelCNN Decoders, Oord, et. al., arXiv June 2016
- PixelSNAIL: An Improved AutoRegressive Generative Model, Chen, et. al., Arxiv December 2017
- NICE: Nonlinear Independent Component Estimation, Dinh, et. al., arXiv October 2014
- Density Estimation Using Real NVP, Dinh, et. al., arXiv May 2016
- Glow: Generative Flows with Invertible 1x1 Convolutions, Kingma, et. al., arXiv July 2018
- Flow++: Improving Flow-Based Geneative Models with Variational Dequantization and Architecture Design, Ho, et. al., arxiv February 2019
- Auto-Encoding Variational Bayes, Kingma, et. al., arXiv December 2013
- Neural Discrete Representation Learning, Oord, et. al., arXiv November 2017
- Generating Diverse High Fidelity Images with VQ-VAE2, Razavi, et. al., arXiv June 2019