2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)
October 25-29, 2020, Las Vegas, NV, USA (Virtual)

THE UNIVERSITY OF TOKYO
松尾研究室
Matsuo Lab., the University of Tokyo

Pixyz: a framework for developing deep generative models

Masahiro Suzuki

Project assistant professor, the University of Tokyo

978-1-7281-6211-9/20/$31.00 ©2020 IEEE

Outline

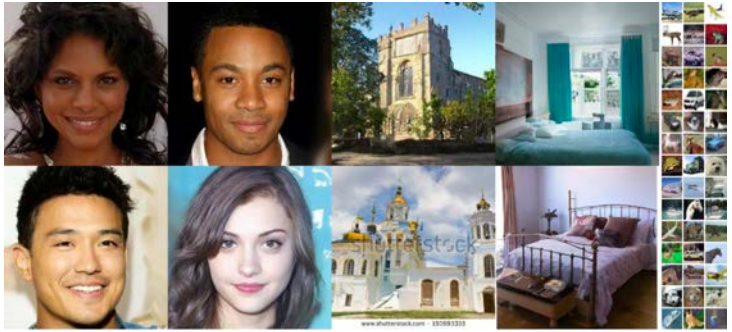
- A brief description of deep generative models
- Pixyz: a framework for developing deep generative models
- Tutorial with Jupyter notebook

Outline

- A brief description of deep generative models
- Pixyz: a framework for developing deep generative models
- Tutorial with Jupyter notebook

782

What you can do with deep generative models



[Ho+ 20]

A stop sign is flying in blue skies.

[Mansimov+ 15]

[Zhu + 17]

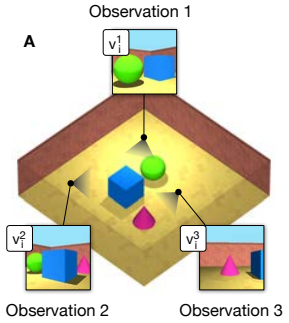
Figure 1: Class-conditional samples generated by our model.

[Brock+ 18]

i went to the store to buy some groceries .
i store to buy some groceries .
i were to buy any groceries .
horses are to buy any groceries .
horses are to buy any animal .
horses the favorite any animal .
horses the favorite favorite animal .
horses are my favorite animal .

783

[Bowman+ 15]



[Eslami+ 18]

Generative Models

- (Probabilistic) generative models :
  - Assume that observed variables (data) are generated from some stochastic models, and that the generating processes are modeled by probability distributions.
    - Latent variables are often assumed to be factors behind the observed variables.
  - Explicitly show "how the data are created".

Observed data

generation

Observed variable

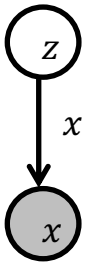
Latent variable

$$p_\theta(x) = \int p_\theta(x|z)p(z)dz$$

Parameter

$z \sim p(z)$

$x \sim p_\theta(x|z)$

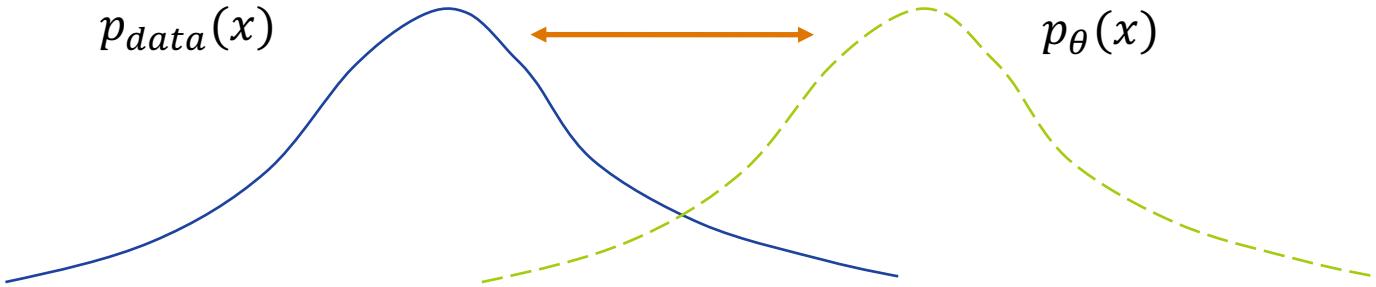


Generative model

784

Training of Generative Models

- Suppose there is a true distribution of training data: $p_{data}(x)$.
  - Note: this distribution is not actually obtained.
- Train the generative model $p_\theta(x)$ to approximate the true distribution $p_{data}(x)$.
  - = finding the parameter θ for the closest approximation.



Training of Generative Models

- Usually, the "distance" is measured with Kullback–Leibler (KL) divergence:

$$\underbrace{D_{KL}[p_{data}(x)||p_\theta(x)]}_{\text{KL divergence}} = -\mathbb{E}_{p_{data}(x)}[\log \frac{p_{data}(x)}{p_\theta(x)}] \geq -\mathbb{E}_{p_{data}(x)}[\log p_\theta(x)]$$

log-likelihood function
= how "good" the model given data

Minimizing KL divergence ⇔ maximizing log-likelihood function

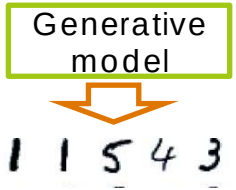
- Therefore, the optimized parameter $\hat{\theta}$ is obtained as follows (maximum likelihood estimation)

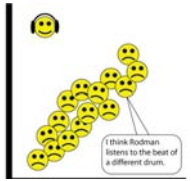
$$\hat{\theta} = \arg\max_\theta \frac{1}{N}\sum_{n=1}^{N} \log p_\theta(x_n), \quad \text{where } x_n \sim p_{data}(x)$$

  - Note: we use sample approximation to calculate the expected value.

What generative models can do

- Generation :
  - If generative models are well trained, they can generate unseen data.
- Density estimation :
  - We can estimate how well the sample fits the model.
  - Outlier detection and anomaly detection
- Imputation :
  - Completing missing values of data.





http://jblomo.github.io/datamining290/slides/2013-04-26-Outliers.html

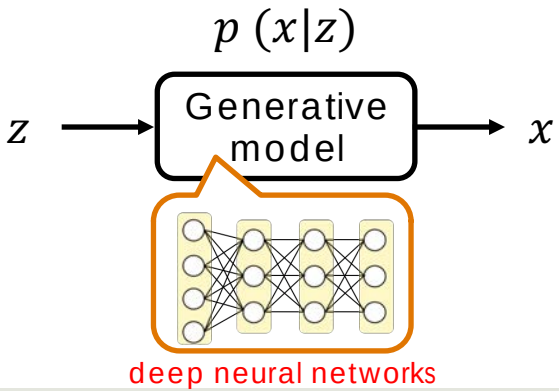
Deep generative models

- If the observed variables are complex, these generating processes cannot be directly expressed with simple probability distributions.
  - When the observed variable is a vector and the dependency between its elements (dimensions) is nonlinear (e.g., high-resolution images).
  - How to represent such nonlinear relationships? -> deep neural networks (DNNs)

- Deep generative models (DGMs)
  - Generative models of which probabilistic distributions are parameterized by DNNs.
  - They can be learned end-to-end from complex inputs.

Generative processes can by represented explicitly (generative models)
+
Non-linear relationships can be captured (DNNs)

788

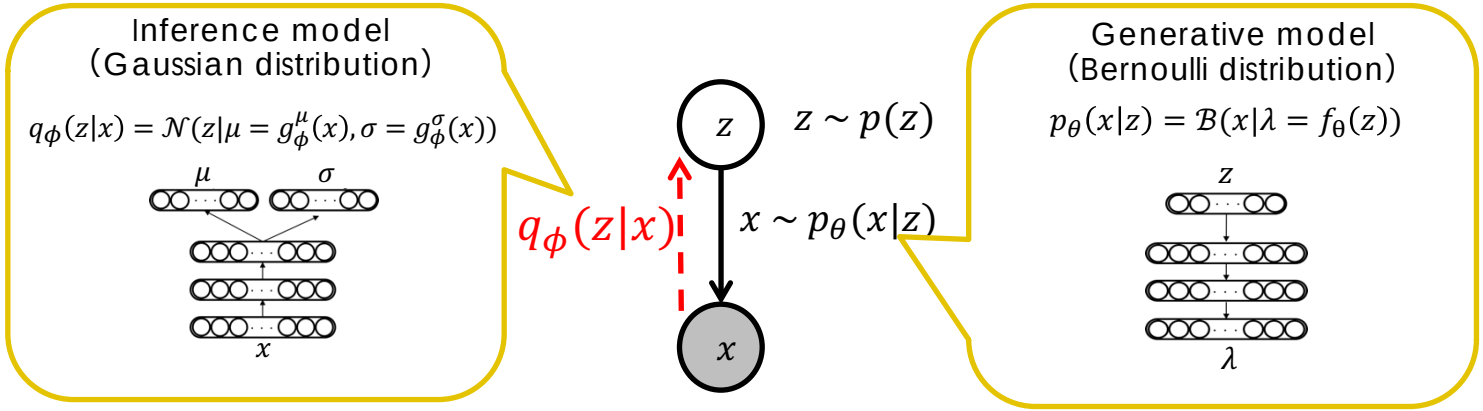


Various deep generation models

	Learning model	Likelihood estimation of generative models	Generation	Inference
VAEs	Generative model: $p(\mathbf{x},\mathbf{z}) = \int p(\mathbf{x}\|\mathbf{z})p(\mathbf{z})d\mathbf{z}$ Inference model; $q(\mathbf{z}\|\mathbf{x})$	Not directly possible (ELBO can be estimated)	Low cost	Possible (inference model)
GANs	Generator: $G(\mathbf{z})$ Discriminator: $D(\mathbf{x})$	Impossible (the discriminator estimates the density ratio)	Low cost	Impossible (or possible with an encoder)
Autoregressive models	Conditional model: $\prod_d p(x_d\|x_1,\dots,x_{d-1})$	Possible	High cost	There are no latent variables
Flow-based models	Flow (invertible function): $\mathbf{x} = f(\mathbf{z})$	Possible	Low cost	Possible (inverse transformation)
Diffusion models	Inverse process: $p(\mathbf{x}_T)\prod_t p(\mathbf{x}_{t-1}\|\mathbf{x}_t)$ Diffusion process : $\prod_t q(\mathbf{x}_t\|\mathbf{x}_{t-1})$	Not directly possible (ELBO can be estimated)	High cost (iterative)	Possible (diffusion process)
Score-based models	Score network: $s(\mathbf{x})$	Not directly possible (log-likelihood gradient can be estimated)	High cost (iterative)	There are no latent variables
Energy-based models	Energy function: $E(\mathbf{x})$	Interactable (because of the partition function)	High cost (iterative)	Depends on the model design.

Variational Autoencoder

- Variational autoencoders [Kingma+ 13, Rezende+ 14]
  - Deep generative model with a latent variable
  - Both inference and generative models are parameterized by DNNs.

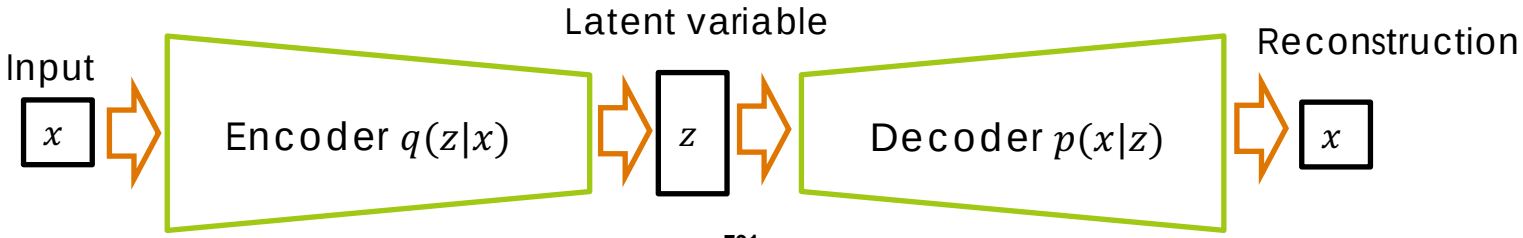


Variational Autoencoder

- Objective function of VAEs
  - Lower bound of the log-likelihood function

$$\log p_\theta(x) \geq \underbrace{\mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)]}_{\text{Negative reconstruction loss}} - \underbrace{D_{KL}[q_\phi(z|x) \parallel p_\theta(z)]}_{\text{Regularization for inference}}$$

- VAE encodes input x into z with an inference model and decodes (reconstructs) x from z with a generative model.
  - The inference model and the generative model can be regarded as an encoder and a decoder in autoencoders.



Generated images from VAEs

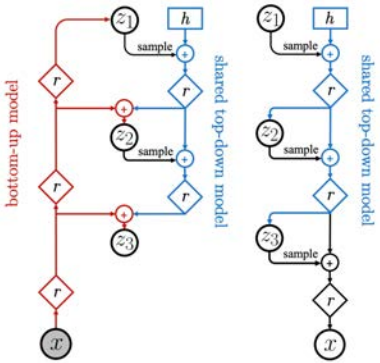
- Sampling images from random z
  - These tend to be blurred.

[Kingma+ 13]

@AlecRad

Generated images from VAEs

- Nouveau VAE（NVAE）[Vahdat+ 20]
  - Hierarchy of latent variables in VAE.
  - Advantages:
    - Acquire hierarchical representations
    - Improve the expressive power of the model
    - More flexible inference



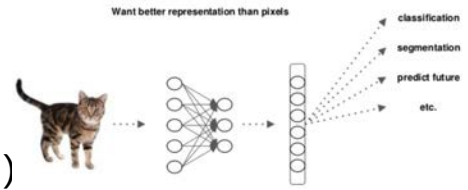
(d) CelebA HQ ($t = 0.6$)

(e) FFHQ ($t = 0.5$)

793

VAEs and representation learning

- VAEs learn not only the reconstruction but also the representation $\mathbf{z} \sim q_\phi(\mathbf{z}|\mathbf{x})$.
  - In DGMs, <u>representation learning is equivalent to inference</u>.
  - VAEs are an excellent representation learning method.



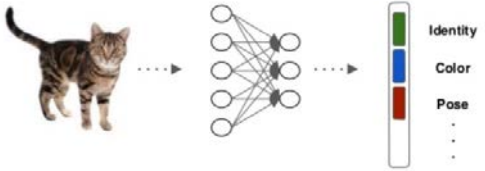
https://www.slideshare.net/lubaelliott/emily-denton-unsupervised-learning-of-disentangled-representations-from-video-creative-ai-meetup

- Representation learning:
  - Obtain "good representation" from the data (preferably unsupervised)
  - What is "good representation"?
    - A representation that not only preserves the properties of the original data but can also be used for other tasks.
  - Meta-Prior [Bengio+ 13, Goodfellow+ 16]
    - Assumptions about the properties of representation that can be used for many tasks
    - e.g., manifold, disentanglement, hierarchy of concepts, semi-supervised learning, clustering, etc.

794

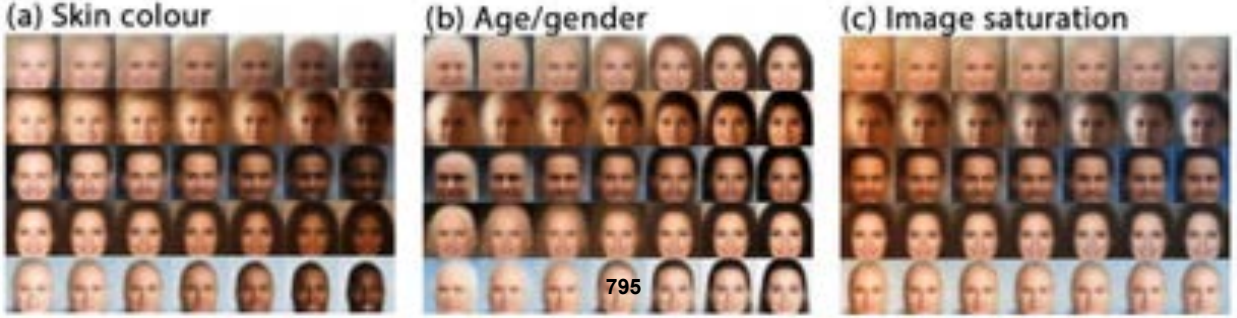
Disentangled representation

- The assumption that data are generated from factors that vary independently
  - Advantage:
    - Easy for humans to interpret (acquisition of "concepts")
    - Potential for use in a variety of tasks



https://www.slideshare.net/lubaelliott/emily-denton-unsupervised-learning-of-disentangled-representations-from-video-creative-ai-meetup

- Disentangled representation can be obtained by regularizing the inference[Higgins+ 17]



795

Multimodal learning

- We use multimodal information to perform more reliable information processing than single modality information.
- Robots also acquire various types of information from multiple sensors.
  - Video, audio, angle, acceleration, etc.

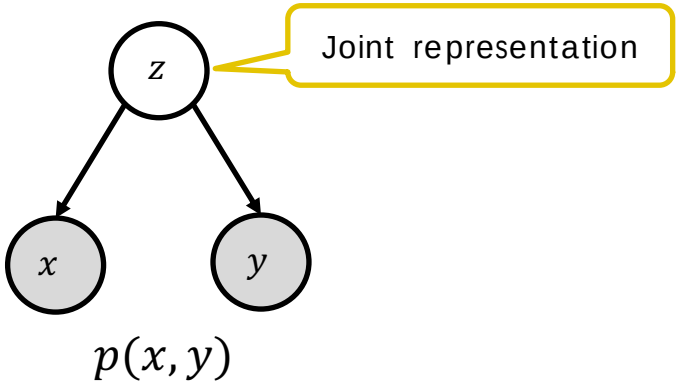
https://www.youtube.com/watch?v=SQSmaVZEXso （HSR）

- We want to make decisions and predictions using multi-modal data in machine learning.

⇒ Multimodal learning

Joint DGMs

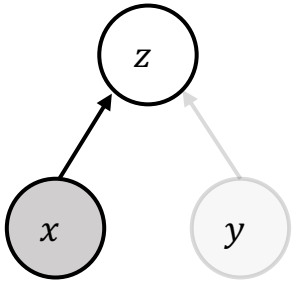
- Joint DGMs: DGMs modeling the joint distribution of modalities $p(x, y)$
  - After learning, this might perform generation with arbitrary conditioning (bidirectional transition): $p(x|y), p(y|x)$
  - This latent variable obtains a joint representation of all modalities.



797

Missing modality problem

- To infer representation from one modality, the other modality needs to be missing.



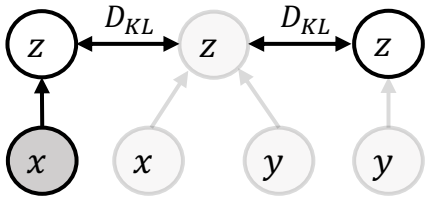
encoder

- If the amount of information on the missing modality y is large, it may not be possible to properly infer z from only x, resulting in a collapsed representation (missing modality problem).
  - This prevents arbitrary conditioning.

JMVAE

- We prepare encoders for each modality, $q(z|x)$ and $q(z|y)$, and learn them to approximate the original VAE encoder, $q(z|x,y)$.

$$\underbrace{\mathrm{E}_{q_\varphi(z|x,y)}[\log p_\theta(x,y|z)] - D_{KL}[q_\varphi(z|x,y) \parallel p_\theta(z)]}_{\text{original objective}} \underbrace{-D_{KL}[q_\varphi(z|x,y) \parallel q_\lambda(z|x)] - D_{KL}[q_\varphi(z|x,y) \parallel q_\lambda(z|y)]}_{\text{approximate the encoder of each modality to the original encoder}}$$



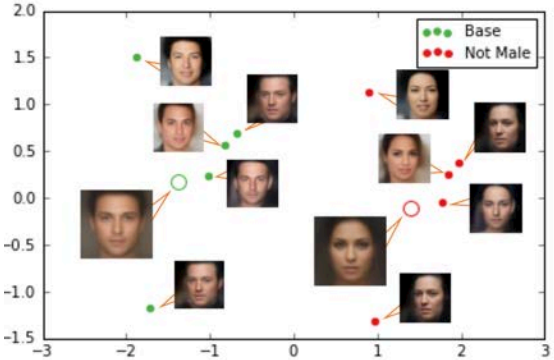
encoder

- After training, we can use each trained encoder, $q(z|x)$ and $q(z|y)$, to infer properly from a single modality.

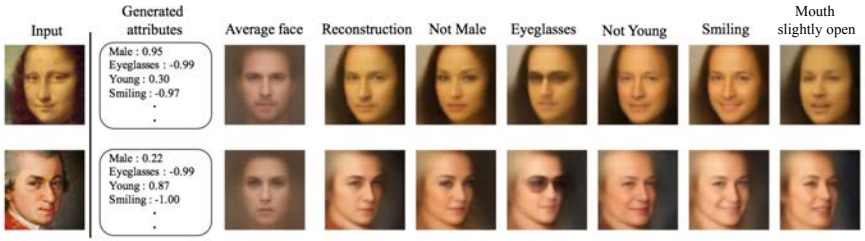
=> JMVAE [Suzuki+ 17]

JMVAE

- We can obtain the joint representation and perform bidirectional generation.
  - e.g., images (x) and attributes (y)



Joint representation



Bi-directional generation (arbitrary conditioning)

- We applied it to semi-supervised learning [Suzuki+ 18] and zero-shot learning [Suzuki+ 18]

- Recent works can deal with more than two modalities.
  - Multimodal variational autoencoder (MVAE) [Wu+ 18] (not our work)

Various deep generation models

	Learning model	Likelihood estimation of generative models	Generation	Inference
VAEs	Generative model: $p(\mathbf{x},\mathbf{z}) = \int p(\mathbf{x}\|\mathbf{z})p(\mathbf{z})d\mathbf{z}$ Inference model; $q(\mathbf{z}\|\mathbf{x})$	Not directly possible (ELBO can be estimated)	Low cost	Possible (inference model)
GANs	Generator: $G(\mathbf{z})$ Discriminator: $D(\mathbf{x})$	Impossible (the discriminator estimates the density ratio)	Low cost	Impossible (or possible with an encoder)
Autoregressive models	Conditional model: $\prod_d p(x_d\|x_1,\ldots,x_{d-1})$	Possible	High cost	There are no latent variables
Flow-based models	Flow (invertible function): $\mathbf{x} = f(\mathbf{z})$	Possible	Low cost	Possible (inverse transformation)
Diffusion models	Inverse process: $p(\mathbf{x}_T)\prod_t p(\mathbf{x}_{t-1}\|\mathbf{x}_t)$ Diffusion process : $\prod_t q(\mathbf{x}_t\|\mathbf{x}_{t-1})$	Not directly possible (ELBO can be estimated)	High cost (iterative)	Possible (diffusion process)
Score-based models	Score network: $s(\mathbf{x})$	Not directly possible (log-likelihood gradient can be estimated)	High cost (iterative)	There are no latent variables
Energy-based models	Energy function: $E(\mathbf{x})$	Interactable (because of the partition function)	High cost (iterative)	Depends on the model design.

Outline

- A brief description of deep generative models
- Pixyz: a framework for developing deep generative models
- Tutorial with Jupyter notebook

802

How to implement deep generative models?

- Libraries for implementing deep neural networks
  - Tensorflow（Keras）, PyTorch, etc.
  - Many of deep generative model studies use one of these libraries.
    - Since these are not treated as probabilistic models, it is difficult to implement complex deep generative models.
    - Their implementation differs from person to person and tends to difficult to read.

- Probabilistic programing languages
  - Libraries for designing and learning (inference) about probabilistic models.
    - Stan[Gelman 12], PRISM, PyMC3[Salvatier 16]（PyMC4）, Edward[Tran+ 17] (Edward 2), Pyro[Bingham+ 18], Tensorflow probability
  - TensorFlow Probability, Edward, and Pyro can be implemented by mixing DNNs and probability distributions.
    - Tensorflow Probability and Edward: based on Tensorflow
    - Pyro: based on PyTorch

803

Implementation example of deep generative model with probabilistic programming language

- Design a generative model by describing the generative processes of variables.
  - Edward (v1)

```
# MODEL
z = Normal(loc=tf.zeros([FLAGS.M, FLAGS.d]),
           scale=tf.ones([FLAGS.M, FLAGS.d]))
hidden = tf.layers.dense(z, 256, activation=tf.nn.relu)
x = Bernoulli(logits=tf.layers.dense(hidden, 28 * 28))

# INFERENCE
x_ph = tf.placeholder(tf.int32, [FLAGS.M, 28 * 28])
hidden = tf.layers.dense(tf.cast(x_ph, tf.float32), 256,
                         activation=tf.nn.relu)
qz = Normal(loc=tf.layers.dense(hidden, FLAGS.d),
            scale=tf.layers.dense(
                hidden, FLAGS.d, activation=tf.nn.softplus))

# Bind p(x, z) and q(z | x) to the same TensorFlow placeholder for x.
inference = ed.KLqp({z: qz}, data={x: x_ph})
```

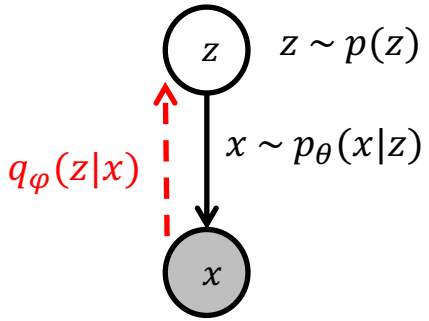
Generative model

$z \sim p(z)$

$x \sim p_\theta(x|z)$

Inference model

$z \sim q_\varphi(z|x)$

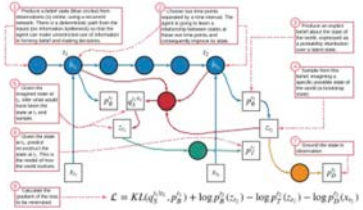


Learning (Inference)

Challenges in implementing deep generative models

- Probabilistic programming languages are rarely used in recent deep generative model studies.
  - Recent deep generative models are very complexed
- TD-VAE[Gregor+ 18]: A sequential deep generative model with multiple inference distributions

$$\mathbb{E}_{q(z_{t_1}, z_{t_2} | b_{t_1}, b_{t_2})} \left[\log p\left(x_{t_2} | z_{t_2}\right) + \log p_B\left(z_{t_1} | b_{t_1}\right) + \log p\left(z_{t_2} | z_{t_1}\right) - \log p_B\left(z_{t_2} | b_{t_2}\right) - \log q\left(z_{t_1} | z_{t_2}, b_{t_1}, b_{t_2}\right) \right]$$

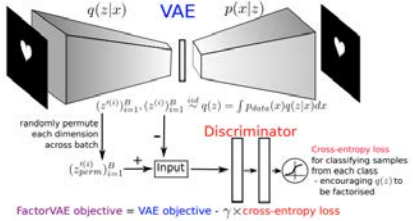


[Gregor+ 18]

- FactorVAE[Kim+ 18]: Adversarial learning is used to estimate the regularization term of VAE inference.

$$\frac{1}{N} \sum_{i=1}^{N} \left[\mathbb{E}_{q(z|x^{(i)})} \left[\log p\left(x^{(i)} | z\right) \right] - KL\left(q\left(z | x^{(i)}\right) \| p(z)\right) \right] - \gamma KL(q(z) \| \overline{q}(z))$$

where $KL(q(z) \| \overline{q}(z)) = \mathbb{E}_{q(z)} \left[\log \frac{q(z)}{\overline{q}(z)} \right] \approx \mathbb{E}_{q(z)} \left[\log \frac{D(z)}{1 - D(z)} \right]$



[Kim+ 18]

Features of deep generative models

We focus on the following features of the deep generative models compared to the normal generation models.

- Feature 1: DNNs that compose DGMs are encapsulated by probability distributions.

- Feature 2: Model structure and regularization are described in the objective function (loss function), which is optimized using gradient methods (e.g., SGD).

806

Feature 1: Encapsulation of DNNs by probability distribution

- The probability distributions on the deep generative models are parameterized by DNNs.
  - -> Each DNN structure is encapsulated by a probability distribution.
- In recent papers of complex deep generative models, the details of DNN are not mentioned in the description of the model.
  - TD-VAE[Gregor+ 18]

$$\mathop{\mathbb{E}}_{q\left(z_{t_1}, z_{t_2} | b_{t_1}, b_{t_2}\right)} \left[\log p\left(x_{t_2} | z_{t_2}\right) + \log p_B\left(z_{t_1} | b_{t_1}\right) + \log p\left(z_{t_2} | z_{t_1}\right) - \log p_B\left(z_{t_2} | b_{t_2}\right) - \log q\left(z_{t_1} | z_{t_2}, b_{t_1}, b_{t_2}\right)\right]$$

-> We need a framework that allows us to implement generative models by manipulating probability distributions without considering the structure of DNNs.

807

Difference in parameterization of probability distribution by DNNs

- The way of representing the probability distribution by DNNs differs depending on the type of deep generation model.

1. Model the conditional distribution $p(x \mid z)$.
   - implicit (GANs) or explicit(VAEs).

$$x \sim p(x|z) \Leftrightarrow x = f(z)$$

2. Model $p(x)$ directly.
   - Auto-regressive models :
     - the product of conditional distributions
   - Flow-based models :
     - change of variables

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

$$p(x) = p(f(x)) \left| \frac{df(x)}{dx} \right|$$

- To handle various deep generation models in a unified manner, <u>differences in parameterization should be concealed with probability distributions</u>.

Feature 2: Model definition by the objective function

- In the deep generative models, the objective function to be optimized is explicitly set.
  - All inference and regularization of variables in the model are added to the objective function.
  - FactorVAE:

$$\frac{1}{N}\sum_{i=1}^{N}\left[\underbrace{\mathbb{E}_{q(z|x^{(i)})}\left[\log p\left(x^{(i)}|z\right)\right]}_{\text{Reconstruction}} - \underbrace{KL\left(q\left(z|x^{(i)}\right)\|p(z)\right)}_{\text{Regularization of representation}}\right] - \gamma KL(q(z)\|\overline{q}(z))$$

  - In the context of deep generative models, model design ≒ objective function definition

-> We need a framework that can easily define the objective function from the probability distribution.

809

Features of deep generative models

We focus on the following features of the deep generative models compared to the normal generation models.

- Feature 1: DNNs that composes the deep generative models are encapsulated by probability distributions.

- Feature 2: Model structure and regularization are described into the objective function (loss function), which is optimized using gradient methods (e.g., SGD).

Features of deep generative models

We focus on the following features of the deep generative models compared to the normal generation models.

- Feature 1: DNNs that composes the deep generative models are encapsulated by probability distributions.

  => A framework that can encapsulate DNNs with probability distribution （Distribution API）

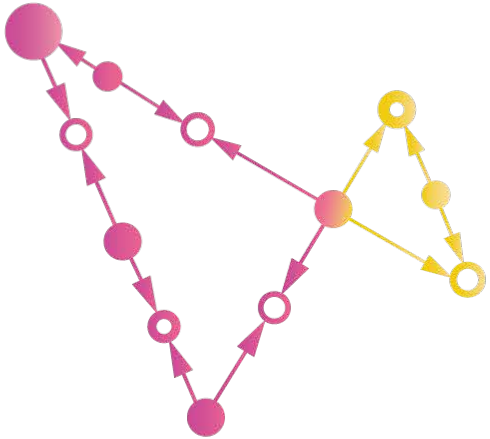
- Feature 2: Model structure and regularization are described into the objective function (loss function), which is optimized using gradient methods (e.g., SGD).

  => A framework that can easily define the objective function from the probability distribution （Loss API） and can train it（Model API）.

We propose to implement deep generative models with a combination of APIs considering each feature.

811

Features of deep generative models

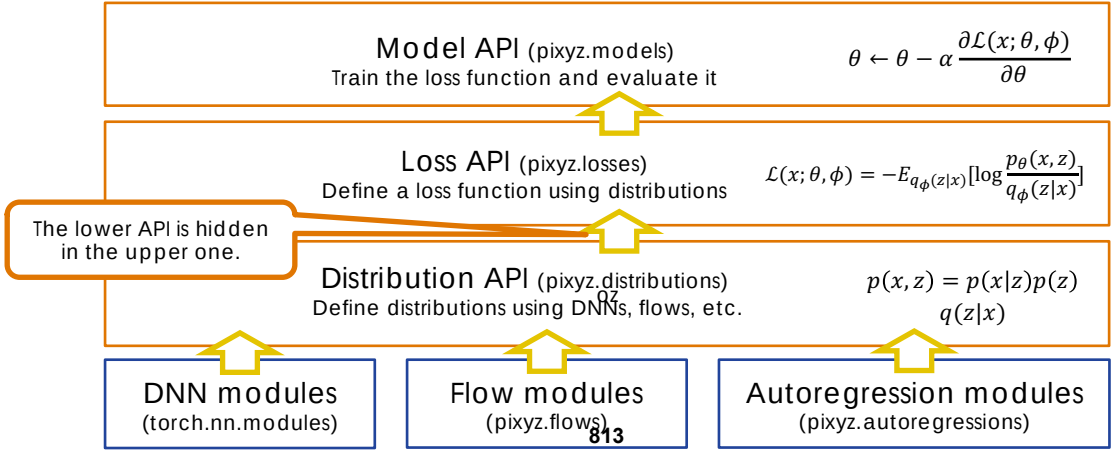


$$P(x,y,z)$$

812

Pixyz

- Pixyz: PyTorch-based library specialized for deep generative models.
  - We focus on making it easier to implement and use complex deep generative models
  - Step-by-step implementation with three APIs.
  - The upper API is independent of the lower API.



Implementation of VAE (1. Distribution API)

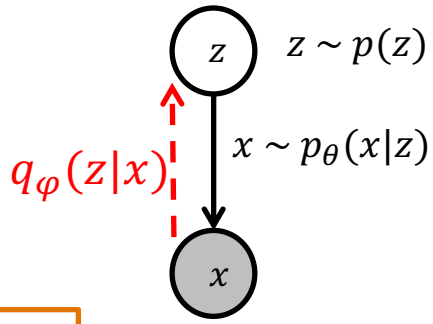
- Define the probability distribution by DNN
  - Inherit the pixyz.Distribution.
  - It is almost the same as the usual PyTorch implementation.

```python
# generative model p(x|z)
class Generator(Bernoulli):
    def __init__(self):
        super(Generator, self).__init__(cond_var=["z"], var=["x"],
                                        name="p")

        self.fc1 = nn.Linear(64, 784)

    def forward(self, z):
        return {"probs": torch.sigmoid(self.fc1(z))}

p = Generator().to(device)
```

$p_\theta(x|z) = \mathcal{B}(x; \lambda = g(z))$ (Bernoulli)

```python
# inference model q(z|x)
class Inference(Normal):
    def __init__(self):
        super(Inference, self).__init__(cond_var=["x"], var=["z"],
                                        name="q")

        self.fc1 = nn.Linear(784, 100)
        self.fc31 = nn.Linear(100, 64)
        self.fc32 = nn.Linear(100, 64)

    def forward(self, x):
        h = F.relu(self.fc1(x))
        return {"loc": self.fc31(h), "scale": F.softplus(self.fc32(h))}

q = Inference().to(device)
```

$q_\varphi(z|x) = \mathcal{N}(z; \mu = f_\mu(x), \sigma^2 = f_{\sigma^2}(x))$ (Gaussian)



- Define probability distribution without DNN

```python
prior = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.),
               var=["z"], features_shape=[z_dim], name="p_{prior}").to(device)
```

$p(z) = \mathcal{N}(z; 0,1)$ (Gaussian)

814

Implementation of VAE (1. Distribution API)

- The Distribution API can represent joint distributions by multiplying the distributions.

$p_\theta(x,z) = p_\theta(x|z)p(z)$

$x, z \sim p_\theta(x,z)$

```
p_joint = p * prior
samples = p_joint.sample(batch_n=64)
print(samples)
```

```
{'z': tensor([[ 0.7152, -1.4977, -0.0758,  ...,  1.1506,  0.4569, -1.4079],
        [-1.4547,  0.7707,  1.2808,  ..., -1.0872, -2.3142, -0.2347],
        [ 0.9625, -1.1309,  1.2302,  ...,  0.4613, -0.1019,  0.7201],
        ...,
        [ 0.3662,  0.2852,  0.3178,  ...,  0.2123, -2.9080,  1.3641],
        [-0.6231, -0.4798, -0.7857,  ..., -0.2356,  0.4331,  0.5930],
        [-1.2890, -0.0230,  1.3472,  ...,  0.7161, -0.6847, -1.0407]]), 'x': tensor([[1., 0., 1.,  ..., 0
        [1., 0., 1.,  ..., 1., 0., 0.],
        [0., 0., 1.,  ..., 1., 0., 1.],
```

- The modeled distribution can be confirmed with “print”.

```
print_latex(p_joint)
```

$$p(x,z) = p(x|z)p_{prior}(z)$$

Implementation of VAE (2. Loss API)

- Loss API defines “Loss classes” with Distribution classes as an argument.
  - We can calculate between Loss.

    -> We can convert the equation written in the paper directly into the implementation.

The objective of VAE : $E_{p_{data}(x)}[-D_{KL}[q_\varphi(z|x) \parallel p(z)] + E_{q_\varphi(z|x)}[\log p_\theta(x|z)]]$

```
elbo = (-KullbackLeibler(q, prior) + E(q, LogProb(p))).mean()
```

Loss class Loss class

- The defined “Loss” can also check what objective function is implemented by printing.

```
print_latex(elbo)
```

$$mean\left(-D_{KL}\left[q(z|x)||p_{prior}(z)\right] + \mathbb{E}_{q(z|x)}\left[\log p(x|z)\right]\right)$$

Implementation of VAE (2. Loss API)

- The value of Loss can be evaluated by giving data (= lazy evaluation)

training data (tensor)

```
loss.eval({"x": x})
```

```
tensor(565.5946, grad_fn=<MeanBackward0>)
```

817

Implementation of VAE (3. Model API)

- Set Loss and optimization algorithm, and train given data.

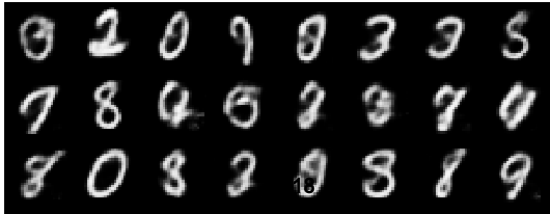
```
model = Model(loss=loss, distributions=[p, q],
              optimizer=optim.Adam, optimizer_params={"lr": 1e-3})

model.train({"x": x})
```

- After training, we can easily infer latent variables and generate images using inference models and generation models.

$z \sim q_\varphi(z|x)$

$x, z \sim p_\theta\,(x, z)$

```
z_sample = q.sample({"x": x}) # inference model
x_sample = p_joint.sample(batch_n=64) # generative model
```



Implementations of complex DGMs (Loss API)

- TD-VAE

```
kl = KullbackLeibler(q, p_b1)
reconst = E(q,  -p_t.log_prob() - p_d.log_prob() + p_b2.log_prob())
step_loss = E(p_b2, reconst + kl)
_loss = IterativeLoss(step_loss, max_iter=seq_len-1,
                      series_var=["x", "b"], timestep_var=["t"],
                      slice_step=slice_step)
loss_cls = E(belief_state_net, _loss).mean()
print_latex(loss_cls)
```

$$mean\left(\mathbb{E}_{p(b|x)}\left[\sum_{t=1}^{19}\mathbb{E}_{f(x_{t2},b_{t1},b_{t2},t,x,b)}\left[\mathbb{E}_{p_\theta(z_{t1}|b_{t2})}\left[D_{KL}\left[q(z_{t1}|z_{t2},b_{t1},b_{t2})||p_b(z_{t1}|b_{t1})\right]+\mathbb{E}_{q(z_{t1}|z_{t2},b_{t1},b_{t2})}\left[\log p_b(z_{t2}|b_{t2})-\log p_d(x_{t2}|z_{t2})-\log p_t(z_{t2}|z_{t1})\right]\right]\right]\right]\right)$$

- FactorVAE

```
reconst = StochasticReconstructionLoss(q, p)
kl = KullbackLeibler(q, prior)
tc = AdversarialKullbackLeibler(q, q_shuffle, discriminator=d, optimizer=optim.Adam, optimizer_params={"lr":1e-3})
loss_cls = reconst.mean() + kl.mean() + 10*tc
print_latex(loss_cls)
```

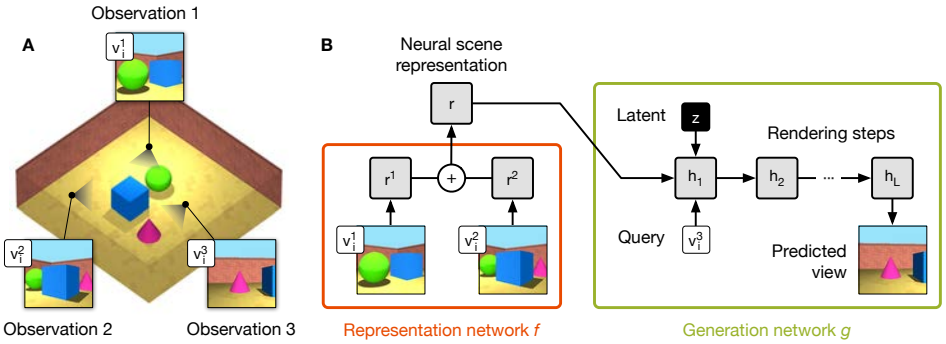
$$mean\left(-\mathbb{E}_{q(z|x)}\left[\log p(x|z)\right]\right)+mean\left(D_{KL}\left[q(z|x)||p_{prior}(z)\right]\right)+10mean(D_{KL}^{Adv}\left[q(z|x)||q_{shf}(z|x_{shf})\right])$$

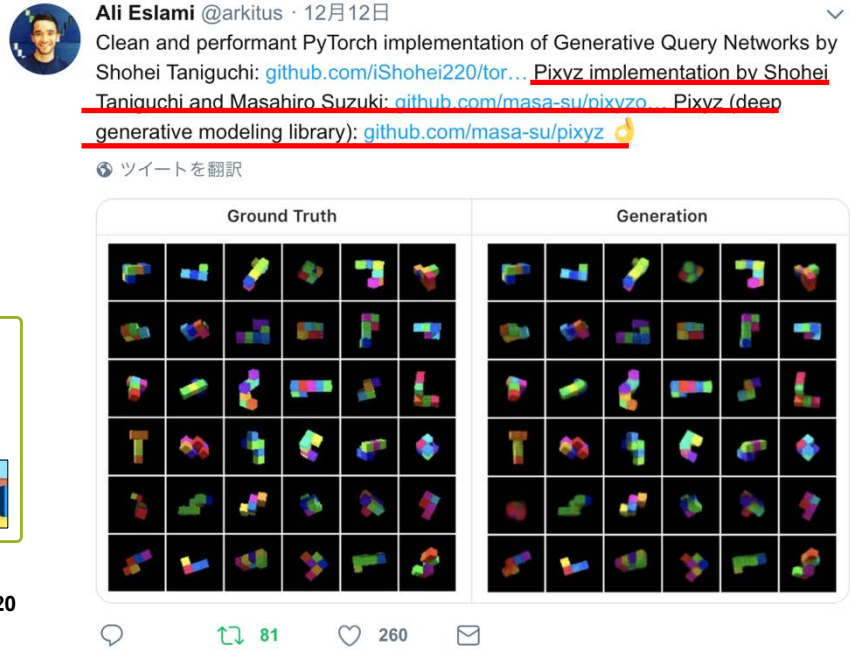
Pixyz allows you to implement each model in a simple and readable manner.

819

Implementations of complex DGMs

- GQN[Eslami+ 18]: Neural rendering model that generates images from different viewpoints based on multiple viewpoints





Learning speed in Pixyz

- We compared with Pyro, the probabilistic programming language implemented on PyTorch.
  - Also compared with raw implementation in PyTorch (this should be the fastest)
  - Comparison of learning time of VAE per step (z: dimension of latent variable, h: that of hidden layer)
- Results :

$\#z$	$\#h$	PyTorch (ms)	Pyro (ms)	Pixyz (ms)
10	400	2.47 ± 0.11	4.91 ± 0.12	3.61 ± 0.11
30	400	2.49 ± 0.10	4.94 ± 0.13	3.58 ± 0.10
10	2000	3.26 ± 0.11	4.93 ± 0.12	3.62 ± 0.09
30	2000	3.28 ± 0.10	4.95 ± 0.12	3.65 ± 0.09

  - Pixyz is faster than Pyro.
  - Compared to the raw PyTorch implementation, the speed has not dropped significantly.

Pixyz is suitable for implementing complex deep generative models.

Pros and cons of Pixyz

- Pros
  - We can intuitively implement arbitrary deep generative models.
  - The implemented code is easy to read and easy to reuse.
  - The speed is not much slower than the raw PyTorch implementation.

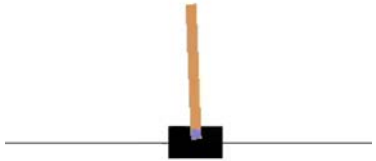
- Cons
  - It cannot be applied to Bayesian deep learning.
    - In Bayesian deep learning, it is necessary to infer the posterior distribution of network parameters.

Outline

- A brief description of deep generative models
- Pixyz: a framework for developing deep generative models
- Tutorial with Jupyter notebook

Hands-on

- Overview of Pixyz
  - 00-PixyzOverview.ipynb
- Details of each API (skip in this presentation)
  - Distribution API description
    - 01-DistributionAPITutorial.ipynb
  - Loss API description
    - 02-LossAPITutorial.ipynb
  - Model API description
    - 03-ModelAPITutorial.ipynb
- Application: deep Markov models using cartpole dataset
  - 04-DeepMarkovModel.ipynb



=> Move to Jupyter Lab!

824

Summary

- A brief description of deep generative models
- Pixyz background & description
- Tutorial with Jupyter notebook

- Acknowledgements
  - This library is based on results obtained from a project commissioned by the New Energy and Industrial Technology Development Organization (NEDO).

825