The 33rd Annual Conference of the Japanese Society for Artificial Intelligence, 2019

Pixyz：複雑な深層生成モデル開発のためのフレームワーク

Pixyz: a framework for developing complex deep generative models

鈴木 雅大 *1 Masahiro Suzuki　金子 貴輝 *1 Takaaki Kaneko　谷口 尚平 *1 Shohei Taniguchi　松嶋 達也 *1 Tatsuya Matsushima　松尾 豊 *1 Yutaka Matsuo

*1東京大学
The University of Tokyo

In recent years, as researches of deep generative models (DGMs) have progressed rapidly, we need a framework which can be used to implement them in a simple and versatile manner. In this research, we focus on the following two aspects as the features of the latest complex DGMs: 1. Deep networks of them are concealed by probability distributions. 2. Their objective functions are composed of multiple loss functions. Then, in order to achieve them, we propose a new DGM library, Pixyz. We show experimentally that our library is faster than existing probabilistic modeling language in training a simple DGM, and also demonstrate that our library makes it easy and concise to implement complex DGMs which cannot be done with existing libraries.

1. はじめに

近年，深層生成モデルの研究が急速に進んでおり，画像や文書生成の他，異常検出やノイズ除去，半教師あり学習，表現学習，メタ学習など，幅広い応用先があることから注目を集めている．そうした中で，多くのコミュニティで深層生成モデルを開発し活用していくためには，簡潔かつ汎用性の高い実装ができることが求められる．

深層生成モデルは，確率モデルを構成するそれぞれの確率分布が深層ニューラルネットワーク（DNN）で表されている．一般に DNN モデルの開発には，Tensorflow や，Keras，PyTorch，Chainer といったライブラリを利用することが主流である．しかし，特に Keras のように抽象度の高いライブラリは，簡単に DNN を設計できるようになっている一方で，複雑な深層生成モデルの実装には適していないことが指摘されている *1．

確率モデリング言語は，確率モデルを直感的かつ統一的に記述するための枠組みであり，最近では GPU で実行でき，DNN と確率分布を混ぜて記述できる Edward [Tran 16] や Pyro [Bingham 18] などが知られている．これらの言語は，単純な深層生成モデルや，ベイズ的ニューラルネットワークの実装などに使われているものの，近年数多く提案されている，複雑な深層生成モデルの実装には殆ど使われていない．その理由として，近年の深層生成モデルの次のような特徴に着目する：

確率分布によるネットワークの隠蔽 従来の確率モデルとは異なり，深層生成モデルでは，モデルを構成する確率分布がニューラルネットワークで設計される．つまり，DNN は確率分布によって隠蔽されており，それぞれが確率モデルを構成する「モジュール」のような役割を果たしている *2．近年の確率モデリング言語は，上記で述べたように DNN と確率分布の両方を混ぜて書くことができるものの，確率分布によって隠蔽した書き方をすることは困難であり，隠蔽した確率分布同士を演算する適切な枠組みもない．

複数の項（異なるモデルの誤差関数）で構成される目的関数 近年の深層生成モデルが従来の確率モデルと大きく異なるもう一つの点は，目的関数を設定し，そのネットワークパラメータにおける勾配をとって学習・推論することである．例えば，variational autoencoder（VAE）は対数周辺尤度の下界（ELBO），generative adversarial network（GAN）は交差エントロピー誤差（暗黙的なモデル分布とデータ分布の間のダイバージェンス），そして flow-based モデルでは対数尤度を目的関数として設定している．従来の確率モデリング言語では，それぞれを異なる「モデル」として学習用のクラスを用意しているが，近年の深層生成モデルでは，これらを組み合わせたモデル，すなわち複数の項の誤差関数で定義されるモデルが提案されている．そうしたモデルについては，従来のような生成モデルの生成過程を記述する方式では実装が難しい．

したがって，現在の深層生成モデルの実装には特に決まったライブラリやフレームワークが使われることあまりなく，様々な方法で実装されているのが現状である．そのため，他人の実装を利用するためには，それの実装方法を解読する必要があり，手間がかかる．また，複雑な深層生成モデルの場合，書いた本人すらわからなくなるような難解なコードになる可能性もあり，メンテナンスの点でも問題となる．

本研究では，こうした問題を解決するため，新たな深層生成モデルの実装フレームワーク Pixyz*3 を提案する．本ライブラリでは，上記の深層生成モデルにおける 2 つの特徴を考慮し，3 つの API（Distribution API，Loss API，Model API）による階層的な実装方法を提案する．これによって，複雑な深層生成モデルを記述できるだけではなく，設計したモデルの可読性が増し，改良や利用が容易になる．

本論文では，2 章で関連研究について説明した後，3 章で Pixyz の概要を説明する．その後 4 章で実行速度と複雑なモデ

連絡先: 鈴木雅大，東京大学工学系研究科技術経営戦略学専攻，〒 113-8656 東京都文京区本郷 7-3-1，masa@weblab.t.u-tokyo.ac.jp

*1 https://twitter.com/dustinvtran/status/850394381794373634，Dustin Tran，2017/04/08, Twitter.

*2 実際，近年の深層生成モデル論文の多くで，提案モデルがニューラルネットワークではなく確率モデルとして説明されている．

*3 https://github.com/masa-su/pixyz，なお本論文で説明する Pixyz は version 0.0.5 についてである．

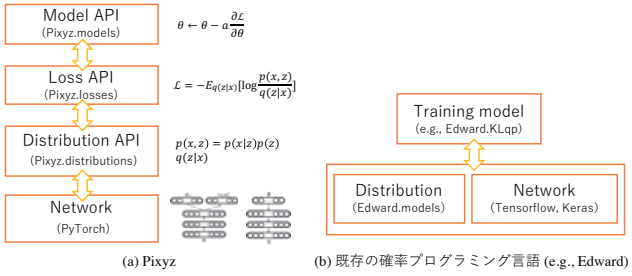


図 1: Pixyz の概要と既存ライブラリとの比較

ルの実装を検証して Pixyz の有効性について議論し，5 章でまとめと今後の展望を述べる．

2. 関連研究

既存の確率モデリング言語には，推論のための高速かつ様々なサンプリングアルゴリズムが実装されている．Stan は C++ で書かれているため高速にサンプリングできることが特徴で，モデル等を記述した Stan 形式のファイルを Python や R で読み込んで実行するインターフェースもある．最近の確率モデリング言語である Edward や Pyro は，Python を用い，TensorFlow や PyTorch 上で動作するように設計されているため，確率モデルを DNN と混ぜて実装でき，GPU 上でのサンプリングや推論，学習が可能である．こうした汎用的な確率モデリング言語は，多くの確率モデルを実装することができるが，1 章で述べた理由から，複雑な深層生成モデルを実装する上では限界がある．

一方で，本研究で提案するライブラリに最も近いのが，Probabilistic Torch（ProbTorch）[Siddharth 17] である．ProbTorch は，Pixyz 同様，深層生成モデルの実装に特化したライブラリであり，PyTorch におけるネットワークの設計の中で分布を定義する点も共通している．しかし次章で述べるように，Pixyz には，ProbTorch にはない，確率分布の積や誤差関数の演算を直感的に実装できるという特徴がある．また，ProbTorch では VAE の実装しか想定されていないが，Pixyz では VAE だけでなく，GAN や flow-based，そしてそれらを組み合わせた複雑な深層生成モデルを実装できる．

3. 提案ライブラリ：Pixyz

本章では，本研究で提案するライブラリ Pixyz の説明をする．Pixyz は Python に基づき，DNN の設計や確率分布のサンプリング及び尤度計算のために PyTorch を用いている．

先述したように，Pixyz は 3 つの API による階層的なフレームワークで構成される．大きな流れとしては，図 1(a) にあるように，1）ニューラルネットワークで確率分布を定義（**Distribution API**），2）確率分布同士を計算し，誤差関数によって確率モデルを定義（**Loss API**），3）定義した誤差関数を最適化するように学習（**Model API**），という形である．

以下，それぞれの API について順番に説明する．なお，図 2 に具体的な実装例を載せたので，各 API の説明の際，参照されたい．

3.1 Distribution API

Pixyz ではまず，確率モデルを構成する各確率分布を定義する．確率分布は，`pixyz.distributions` 以下にガウス分布（`Normal`）やベルヌーイ分布（`Bernoulli`）といった標準的な分布が実装されている．こうした分布クラスは，PyTorch Distribution[*4] とほぼ同じ API であり，初期値として分布のパラメータなどを設定する．このとき，`var` で確率変数名，`name` で確率分布の名前を設定できる．条件付き分布を設計したい場合は，条件付ける確率変数名を `cond_var` の値として指定して初期化する．このようにして，分布クラスのインスタンスを生成した後に，`sample` や `log_likelihood` といったメソッドでサンプリングや与えられたサンプルの尤度を計算することができる．なお，Pixyz ではサンプルは全て辞書形式（`dict`）で表現され，ユーザが定義した確率変数名とその実現値がそれぞれキーと値として格納されている．

次にニューラルネットワークで確率分布を定義する方法を説明する．例えば，ガウス分布を DNN で定義する場合，それらのパラメータ，つまり平均 μ と分散 σ^2 を，DNN によって $\mu = f(x;\theta)$ および $\sigma^2 = g(x;\theta)$（θ はネットワークパラメータ）のように設定する．これを Pixyz で実現するには，用いる分布クラスを継承し，（PyTorch で通常設計するように）コンストラクタにネットワーク，`forward` に入力（条件づける変数）から出力（分布のパラメータ）の流れを記述する．このクラスから生成したインスタンスも，同様に `sample` や `log_likelihood` を使うことができる．つまり，一度インスタンスを生成してしまえば，分布がどのようなネットワークで定義されているかを意識せずに計算を実行できる．これは，深層生成モデルの 1 つ目の特徴（1 章参照）と合致する．

さらに Distribution API では，分布同士の掛け算を直感的にできるという特徴がある．例えば，$p(x|z)$ と $p(z)$ という確率分布を定義した場合，同時分布はそれらの積として $p(x,z) = p(x|z)p(z)$ となるが，Pixyz でも分布インスタンスの積として同時分布を定義することができる．この時，異なる分布クラスに同じ変数名があった場合は同じ変数として処理される．この掛け算したインスタンスもまた分布クラスなので，どのような分布で構成されているかを意識せずに，サンプリングや尤度計算ができる．

なお，このように分布インスタンスが抽象化されると，手元の分布がどのような分布・ネットワークで設計されているかを失念する可能性があるが，Distribution API では，Python の `print` メソッドで分布インスタンスの情報を確認することができる．

既存の確率モデリング言語と比べると，従来は確率モデルを生成過程によって表現していたが，Pixyz では確率モデルを同時分布によって表現する枠組みとなっている．

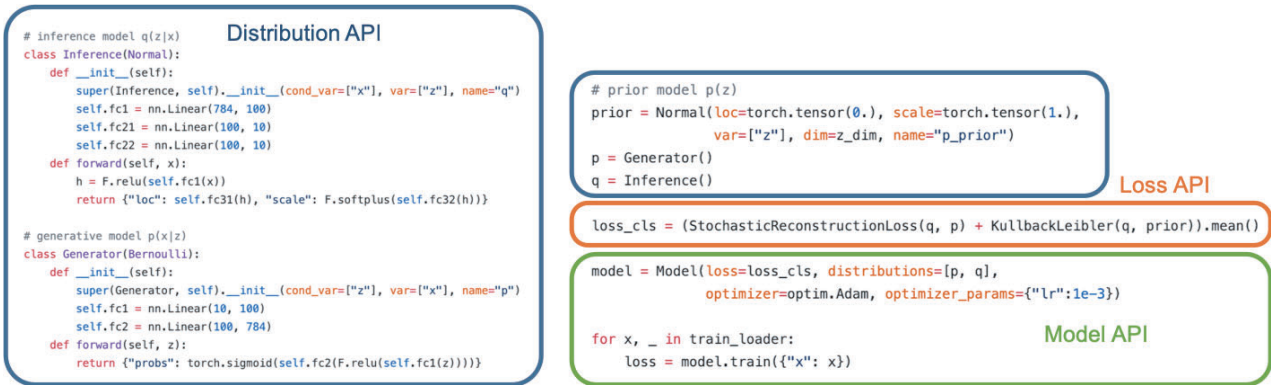
さらに，この API で設計するのは，確率モデルだけでなく，決定論的なニューラルネットワークや，要素の抽出・並び替えといった何らかの「処理」でも良い．その場合は `Deterministic` クラスを継承して定義し，デルタ分布 $p(x|z) = \delta(x - f(z))$ として扱われ，サンプリング結果は決定論的になる．

3.2 Loss API

サンプリングと尤度の計算によって任意の誤差関数を設計できるが，Pixyz では上位の API として，与えられた分布から誤差関数を計算する Loss API がある．Loss API には `pixyz.losses` 以下に，対数尤度，エントロピー，期待値を始め，分布間のダイバージェンスや敵対的損失などが含まれる．これらはいずれも初期化時に分布インスタンスを引数に取り，誤差関数インスタンスを生成する．

Loss API の特徴は，Distribution API と同様，誤差関数インスタンス間の演算ができるということである．例えば VAE の目的関数（負の ELBO）は，再構成誤差と Kullback-Leibler

*4 `https://pytorch.org/docs/stable/distributions.html`




```
# 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.fc21 = nn.Linear(100, 10)
        self.fc22 = nn.Linear(100, 10)
    def forward(self, x):
        h = F.relu(self.fc1(x))
        return {"loc": self.fc31(h), "scale": F.softplus(self.fc32(h))}

# 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(10, 100)
        self.fc2 = nn.Linear(100, 784)
    def forward(self, z):
        return {"probs": torch.sigmoid(self.fc2(F.relu(self.fc1(z))))}
```

```
# prior model p(z)
prior = Normal(loc=torch.tensor(0.), scale=torch.tensor(1.),
               var=["z"], dim=z_dim, name="p_prior")
p = Generator()
q = Inference()
```

```
loss_cls = (StochasticReconstructionLoss(q, p) + KullbackLeibler(q, prior)).mean()
```

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

for x, _ in train_loader:
    loss = model.train({"x": x})
```

図 2: Pixyz による VAE の実装例

（KL）ダイバージェンスで構成される．したがって，それぞれの誤差関数インスタンスを生成して，それらを足し合わせることで，VAE の目的関数を定義できる．第 1 章でも述べたように，近年の複雑な深層生成モデルは，複数の項の誤差関数で定義されるが，Loss API によって，任意の深層生成モデルについて，論文の式をそのまま書くように実装することが可能になる．また，Distribution API と同様，Python の `print` メソッドによって，どのような式を定義したかを確認できる．

なお，インスタンス間の演算は目的関数の式を定義するものであり，それがどのような値をとるかは，観測変数に対応するデータを入れることで評価できる．Loss API では，`estimate` メソッドの引数にデータを入れることで，式を評価することができる．このように，先に式を定義し，その後に値を評価するという方法は，define-and-run 的であるといえる．その一方で，各分布のサンプリングの流れを定義する Distribution API の内部（`forward` メソッド）の記述方法は，これまでの PyTorch と同じなので，define-by-run である．つまり Pixyz は，各確率分布の定義については，柔軟に設計できる方式を採用しつつ，その確率分布から構成される確率モデルと誤差関数の定義については式をそのまま記述できる方式を採っており，両方の方式の利点を採用した形となっている．

また Loss API のもう一つの利点は，誤差関数の定義と学習アルゴリズムを切り離した点である．例えば Edward [Tran 16] のようなライブラリでは，モデル（誤差関数）と学習アルゴリズムが一体となっているため（図 1(b) を参照），複雑な深層生成モデルに拡張することは困難である．一方，Pixyz では Loss API で任意の誤差関数を作れるため，例えば VAE と敵対的訓練を組み合わせたモデル（FactorVAE など）も実装することができる．

3.3 Model API

Model API では，モデルクラス（`pixyz.models.Model`）の初期化時に，Loss API で定義した誤差関数インスタンスや，学習する分布インスタンス，そして最適化アルゴリズムを渡し，モデルインスタンスを生成する．このインスタンスの train メソッドや test メソッドにデータを渡して実行することで，モデルの訓練やテストを行うことができる．

任意の誤差関数を Loss API で設計して誤差逆伝播を用いて学習する場合は，`pixyz.models.Model` を利用する．また，VAE や GAN などの基本的な深層生成モデルを直ぐに学習させてみたいユーザ向けに，誤差関数が内部で定義されているモデルクラスもある（`pixyz.models.VAE`，`pixyz.models.GAN`）．

以上が，Pixyz の実装の流れである．こうした API の階層性の最大の利点は，それぞれの API が互いに干渉しないということである．例えば，確率分布を変更したとしても誤差関数やモデルを変更する必要はないし，誤差関数を変更するときには，Loss API 内で操作が完結するため，ネットワークの構造を変える必要はない．

4. 実験と実装例の検証

本章では，まず Pixyz の実行速度を他のライブラリと比較する．VAE の実装について，PyTorch，Pyro，Pixyz で比較した．Pyro と Pixyz は PyTorch 上に実装されており，さらに確率モデリングライブラリとしての演算処理を含むため，この実装例より速度が遅くなると考えられるが，どれだけ速度を落とさずに実行できるかが複雑な深層生成モデルを実装する上で重要となる．PyTorch と Pyro の実装は Pyro ライブラリの実装例 *5 を利用した．また全ての実装で，[Bingham 18] での実験と同様，エンコーダとデコーダに 2 層の多層パーセプトロンを利用し，バッチサイズは 128 とした．本実験では，実験環境として CPU : Intel® Core™ i5-7500T CPU @ 2.70GHz (4core)，GPU : GeForce GTX 1070（メモリ 32G）を利用した．

表 1 は 1 ステップの更新あたりにかかる時間を示したものである．[Bingham 18] での実験と同様に，10 エポックにおける平均で示しており，隠れ層の次元 $\#h$ と潜在変数の次元 $\#z$ を変えた時の結果を載せている *6．

結果より，Pyro と Pixyz はいずれも PyTorch と比べると遅くなっているものの，Pixyz の方が Pyro よりも速い結果となった．なお，PyTorch の実装では VAE の正則化項（KL ダイバージェンス）を解析的に計算しているのに対して，Pyro ではモンテカルロ近似で計算するように予め設計されている．一方，Pixyz は Loss API のおかげでどちらの目的関数でも容易に実装できるので，両方の場合の結果を載せている．モンテカルロ近似によって遅くなるものの，それでも Pyro よりも高速に学習を進められることがわかる．これは，Pixyz の方が内部の演算や操作がより簡潔であるためと思われる *7．さらに [Bingham 18] では，Pyro の利点として隠れ層が増えると PyTorch との速度のギャップが小さくなることが主張されていたが，Pixyz でも同様であり，特に $h = 2000$ のときは解析

*5 `https://github.com/pyro-ppl/pyro/blob/dev/examples/vae/vae_comparison.py`

*6 $\#h$ と $\#z$ の設定も [Bingham 18] と同じにしているが，実行環境が異なるため，PyTorch と Pyro の速度結果も [Bingham 18] とは異なる．

*7 ただし，Pyro は Pixyz ではできないパラメータのベイズ推論やサンプリング等もサポートしているため，速度だけから一概に良し悪しを結論づけることはできないことにも注意されたい．

表 1: 各ライブラリでの VAE のステップあたりの実行時間の比較．Pixyz については，VAE の正則化項を解析的（analytical）・モンテカルロ近似（Monte Carlo）でそれぞれ計算した結果を載せている．

$\#z$	$\#h$	PyTorch (ms)	Pyro (ms)	Pixyz (analytical) (ms)	Pixyz (Monte Carlo) (ms)
10	400	2.47 ± 0.11	4.91 ± 0.12	3.61 ± 0.11	4.01 ± 0.09
30	400	2.49 ± 0.10	4.94 ± 0.13	3.58 ± 0.10	4.04 ± 0.10
10	2000	3.26 ± 0.11	4.93 ± 0.12	3.62 ± 0.09	4.14 ± 0.10
30	2000	3.28 ± 0.10	4.95 ± 0.12	3.65 ± 0.09	4.11 ± 0.10

$$\sum_{x,y\sim p_{data}(x,y)} [E_{q(z|x,y)}[\log \frac{p(x,y,z)}{q(z|x,y)}]] + \alpha \log q(y|x)] + \sum_{x_u\sim p_{data}(x_u)} [E_{q(z|x_u,y)q(y|x_u)}[\log \frac{p(x_u,y,z)}{q(z|x_u,y)q(y|x_u)}]]$$

```
elbo_u = ELBO(p_joint_u, q_u)
elbo = ELBO(p_joint, q)
nll = NLL(f)
loss_cls = -(elbo - (0.1 * nll)).sum() - elbo_u.sum()
```

(a)

$$\sum_t E_{p_B(z_t|b_t)}[E_{q(z_{t-1}|z_t,b_{t-1},b_t)}[\log p(x_t|z_t) + \log p(z_t|z_{t-1}) - \log p_B(z_t|b_t)] - KL[q(z_{t-1}|z_t,b_{t-1},b_t)||p_B(z_{t-1}|b_{t-1})]]$$

```
kl = KullbackLeibler(q, p_b1)
reconst = LossExpectation(q,  NLL(p_t) + NLL(p_d) - NLL(p_b2))
step_loss = LossExpectation(p_b2, reconst + kl)

_loss = IterativeLoss(step_loss, max_iter=seq_len-1,
                      series_var=["x", "b"],
                      slice_step=slice_step)
loss_cls = LossExpectation(belief_state_net, _loss).mean()
```

(b)

図 3: Pixyz による複雑な深層生成モデルの実装例. (a) : M2 モデル [Kingma 14]，(b) : TD-VAE [Gregor 18]．Distribution API と Model API は図 2 と同様に書けるので，ここでは省略している．

的な場合において PyTorch とほぼ変わらない速度となっている．この結果から，Pixyz は速度面からも，複雑な深層生成モデルの実装に適していると考えられる．

次に，Pixyz を使った複雑な深層生成モデルの実装例を載せる．図 3(a) は，VAE を用いた半教師あり学習モデルである M2 モデル [Kingma 14] の式と，Pixyz による実装例である．複数の項があるモデルでも，Loss API を用いて，式をそのまま書くように目的関数を設計できることがわかる．

さらに複雑なモデルとして，近年注目されている環境をそのまま深層生成モデルで学習する「世界モデル」研究の一つである，TD-VAE [Gregor 18] の式と実装例を図 3(b) に示す．TD-VAE は系列モデルであり，近年の深層生成モデルの中でも最も複雑なものの一つであるものの，非常に簡潔かつ可読性が高く実装できることが確認できる．

その他にも，Generative Query Network (GQN) などの世界モデルや，Deep Markov Model，Variational RNN (VRNN) といった系列情報を扱う深層生成モデルも実装することができる [*8]．

5. まとめ

本研究では，複雑な深層生成モデルを実装するためのライブラリとして Pixyz を提案した．Pixyz は，近年の深層生成モデルが持つ「ネットワークが分布に隠蔽されている」「目的関数が複数の項の誤差関数で構成される」という特徴に対応できるように 3 つの API を活用している．これによって，複雑な深層生成モデルを簡潔に実装できることを示した他，速度面からも複雑な深層生成モデルの設計に適していることを示した．

ただし，確率モデリング言語として Pixyz を捉えると，実現できていないことが多い．例えば，サンプリングについては確率分布からの伝承サンプリングしかサポートしておらず，またベイズ的ニューラルネットワークのような，パラメータにおけるベイズ推論はできない．

今後は，確率モデリング言語としての整備をしつつ，他のライブラリや枠組みとの組み合わせを考えていきたい．例えば，深層生成モデルや世界モデルは，強化学習のアプローチから見ると，状態表現学習の一手法である．したがって，強化学習ライブラリと状態表現学習ライブラリとしての Pixyz を組み合わせた新たなライブラリの開発が考えられる [*9]．

深層生成モデルや世界モデルは，難解さや実装の困難さから，通常の DNN に比べて活用の敷居が高い．本ライブラリが，そうした敷居を無くす役割を少しでも果たせるように，今後も開発を続けるつもりである．

*8 その他の実装例については `https://github.com/masa-su/pixyzoo` を参照されたい．

*9 「組み合わせる」という意味では，既に S-RL Toolbox（`https://github.com/araffin/robotics-rl-srl`）と呼ばれる強化学習と状態表現学習のツールボックスが存在する．

謝辞

本研究は JSPS 科研費 18H06458 および国立研究開発法人新エネルギー・産業技術総合開発機構（ＮＥＤＯ）の委託業務の結果得られたものです．

参考文献

[Bingham 18] Bingham, E., Chen, J. P., Jankowiak, M., Obermeyer, F., Pradhan, N., Karaletsos, T., Singh, R., Szerlip, P., Horsfall, P., and Goodman, N. D.: Pyro: Deep universal probabilistic programming, *arXiv preprint arXiv:1810.09538* (2018)

[Gregor 18] Gregor, K. and Besse, F.: Temporal Difference Variational Auto-Encoder, *arXiv preprint arXiv:1806.03107* (2018)

[Kingma 14] Kingma, D. P., Mohamed, S., Rezende, D. J., and Welling, M.: Semi-supervised learning with deep generative models, in *Advances in neural information processing systems*, pp. 3581–3589 (2014)

[Siddharth 17] Siddharth, N., Paige, B., Meent, van de J.-W., Desmaison, A., Goodman, N. D., Kohli, P., Wood, F., and Torr, P.: Learning Disentangled Representations with Semi-Supervised Deep Generative Models, in Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R. eds., *Advances in Neural Information Processing Systems 30*, pp. 5927–5937, Curran Associates, Inc. (2017)

[Tran 16] Tran, D., Kucukelbir, A., Dieng, A. B., Rudolph, M., Liang, D., and Blei, D. M.: Edward: A library for probabilistic modeling, inference, and criticism, *arXiv preprint arXiv:1610.09787* (2016)