FORNAX

Normalizing Flows: From NICE, RealNVP to GLOW

Krzysztof Kolasiński, 11.10.2018

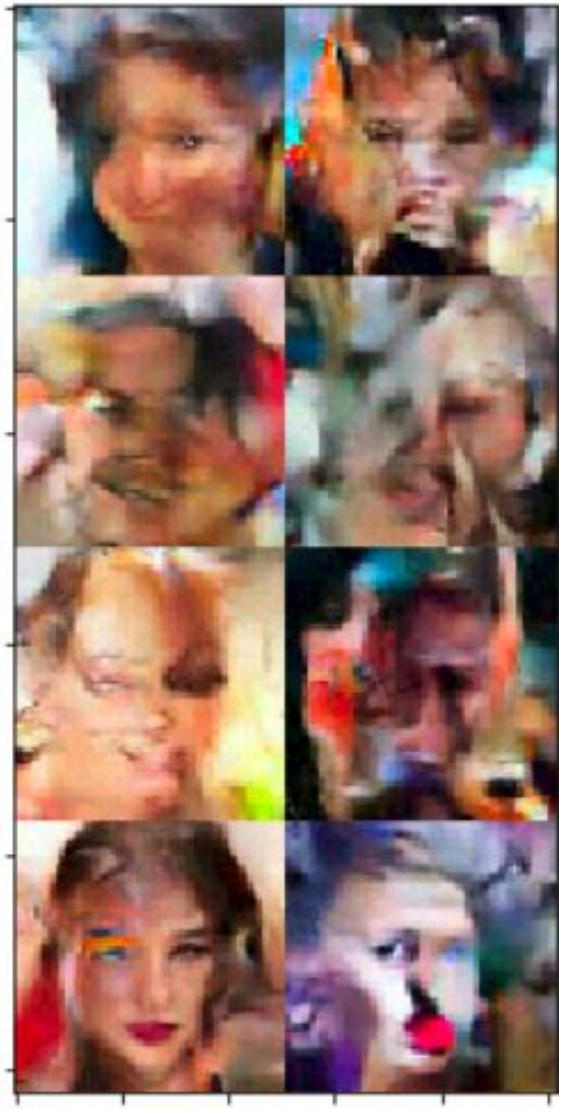
Content

Part 1. Introduction

- Motivations for probabilistic models
- Normalizing flows reminder

Part 2. Research papers study

- NICE, RealNVP, Glow



Artflow

Introduction and motivations for Normalizing Flows

Motivation: The problem of density estimation

Consider following problem:

- we observe some data **X** which are sample from some distribution p(**X**)
- we want to tell what is p(**X**') where **X'** is some new data.

Example: anomaly detection

The simplest classical approach for this problem would be:

Take the data

```
array([[ 1.35791411,  0.41725294],
       [-0.46717576,  0.72275205],
       [ 1.01548965,  0.65188145],
       [ 0.89550028, -0.41804664],
       [ 0.90974391,  0.40584828],
       [-0.59003111,  0.92042647],
       [-1.52335804, -0.91684024],
       [ 0.25105829, -0.66614504],
       [-0.77281828, -1.28366714],
       [-0.4297637 , -0.61117532]])
```

Fit e.g. multivariate gaussian

$$p_{\text{gaussian}}(\mathbf{X})$$

Find anomalies:

$$p_{\text{gaussian}}(\mathbf{X}') < \varepsilon$$

Motivation: The problem of sampling

Consider following problem:

- we observe some data **X** which are sample from some distribution p(**X**)
- we want to generate new data point **X'** according to p(**X**')

Example: dataset augmentation, latent space manipulation, generating new samples

The current state of the art approach for this problem would be to take some deep generative neural network e.g. Glow:

Example from: https://blog.openai.com/glow/

Motivation: probabilistic modeling

- Probabilistic models have nice theoretical interpretation
- However they are usually hard to train or they have intractable normalizing constants

Typical probabilistic model is specified as some function

$$q(\mathrm{x};\theta) = \frac{f(\mathrm{x};\theta)}{Z_\theta}$$

where:

- $f(\mathrm{x};\theta)$ is some positive function parameterized with theta, usually neural network
- *Z* is normalization constant, contains sum of f(x) over all possible configurations of x

Unfortunately:

- in many cases Z is intractable, inference is impossible or require approximations
- sampling may be hard, so we cannot generate samples easily

Normalizing Flows: try to overcome this problem by introducing continuous transformation of probability density which allows for tracking the change in the density p(x)

A brief introduction to Normalizing Flows

Normalizing flows - generalization

Let **z** be a random vector with probability density function q(**z**) and f(**z**) is an invertible function.

Transformation $\mathbf{z}^{(n+1)} = f\left(\mathbf{z}^{(n)}\right)$ of the random variable leads to the following change in the probability density:

$$q'(\mathbf{z}') = q(\mathbf{z}) \left| \det \frac{\partial f\left(\mathbf{z}^{(n)}\right)}{\partial \mathbf{z}^{(n)}} \right|^{-1}$$

det = determinant

jacobian matrix

$$\frac{\partial f(\mathbf{z})}{\partial \mathbf{z}} = \begin{pmatrix} \frac{\partial f_1}{\partial z_1} & \cdots & \frac{\partial f_1}{\partial z_n} \\ \frac{\partial f_2}{\partial z_2} & & \\ \vdots & \ddots & \vdots \\ & \frac{\partial f_{n-1}}{\partial z_{n-1}} & \\ \frac{\partial f_n}{\partial z_1} & \cdots & \frac{\partial f_n}{\partial z_n} \end{pmatrix}$$

Normalizing flows - generalization

- Stacking multiple mappings leads to final expression for the normalizing flow - a chain rule

$$
\begin{aligned}
\mathbf{z}^{(0)} &\sim p(\mathbf{z}) \\
\mathbf{z}^{(1)} &= f^{(1)}(\mathbf{z}^{(0)}) \\
\mathbf{z}^{(2)} &= f^{(2)}(\mathbf{z}^{(1)}) \\
&\dots \\
\mathbf{z}^{(n)} &= f^{(n)}(\mathbf{z}^{(n-1)})
\end{aligned}
$$

$$
q^{(n)}(\mathbf{z}^{(n)}) = q(\mathbf{z}^{(0)}) \prod_{i=1}^{n} \left| \det \frac{\partial \mathbf{z}^{(i)}}{\partial \mathbf{z}^{(i-1)}} \right|^{-1}
$$

density of the sampled point

- Note, that in practice due to the numerical errors it is more stable to work with logarithms

$$
\log q^{(n)}(\mathbf{z}^{(n)}) = \log q(\mathbf{z}^{(0)}) - \sum_{i=1}^{n} \log \left| \det \frac{\partial \mathbf{z}^{(i)}}{\partial \mathbf{z}^{(i-1)}} \right|
$$

Normalizing flows - technical details

- Combining multiple transformations yields in:

$$\begin{aligned}\mathbf{z}^{(0)} &\sim p(\mathbf{z}) \\ \mathbf{z}^{(1)} &= f^{(1)}(\mathbf{z}^{(0)}) \\ \mathbf{z}^{(2)} &= f^{(2)}(\mathbf{z}^{(1)}) \\ &\cdots \\ \mathbf{z}^{(n)} &= f^{(n)}(\mathbf{z}^{(n-1)})\end{aligned} \quad \Rightarrow \quad q^{(n)}(\mathbf{z}^{(n)}) = q(\mathbf{z}^{(0)}) \prod_{i=1}^{n} \left| \det \frac{\partial \mathbf{z}^{(i)}}{\partial \mathbf{z}^{(i-1)}} \right|^{-1}$$

- In most cases when people implement NFs they take into account following rules:
  - use **log densities**, since this is more **stable numerically**
  - use f such that we can **compute determinant easily**: diagonal or triangular matrix for which determinant can be easily computed analytically, or other not investigated options ???
  - use f such that computation of **f(x) is easy**: efficient sampling
  - use f such that computation of **$f^{-1}(y)$ is easy**: efficient estimation of log-likelihood
  - use f such that we **can compute derivative of it:** can be used with gradient descent
  - use f such that computation is **stable numerically**
  - use f such that it has **high expressive power** usually some non linear function

Note: Various methods have been developed, but usually only part of these conditions are satisfied.

The plan

NICE → RealNVP → Glow

(b) Model trained on TFD

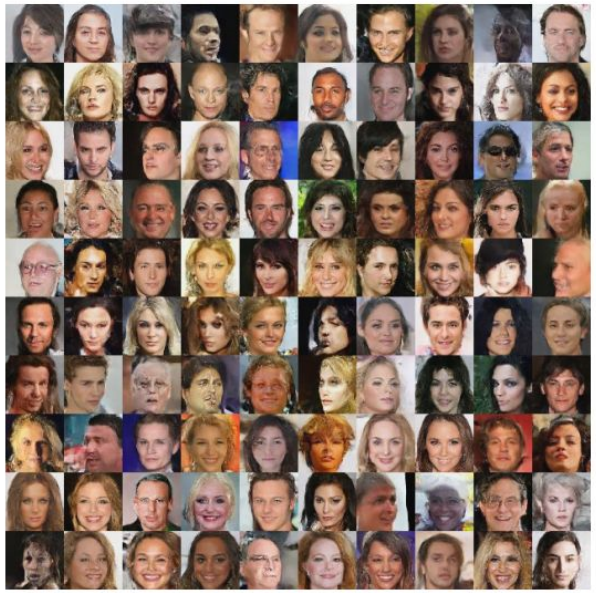


Figure 8: Samples from a model trained on *CelebA*.

Case study: NICE 2015

NICE: Non-linear Independent Components Estimation

Laurent Dinh David Krueger Yoshua Bengio*
Département d'informatique et de recherche opérationnelle
Université de Montréal
Montréal, QC H3C 3J7

probabilistic model

$$p_H(h) = \prod_d p_{H_d}(h_d)$$

ABSTRACT

We propose a deep learning framework for modeling complex high-dimensional densities called Non-linear Independent Component Estimation (NICE). It is based on the idea that a good representation is one in which the data has a distribution that is easy to model. For this purpose, a non-linear deterministic transformation of the data is learned that maps it to a latent space so as to make the transformed data conform to a factorized distribution, i.e., resulting in independent latent variables. We parametrize this transformation so that computing the determinant of the Jacobian and inverse Jacobian is trivial, yet we maintain the ability to learn complex non-linear transformations, via a composition of simple building blocks, each based on a deep neural network. The training criterion is simply the exact log-likelihood, which is tractable. Unbiased ancestral sampling is also easy. We show that this approach yields good generative models on four image datasets and can be used for inpainting.

$$p_X(x) = p_H(f(x))|\det \frac{\partial f(x)}{\partial x}|$$

$$\mathbf{z}^{(0)} \sim p(\mathbf{z})$$
$$\mathbf{z}^{(1)} = f^{(1)}(\mathbf{z}^{(0)})$$
$$\mathbf{z}^{(2)} = f^{(2)}(\mathbf{z}^{(1)})$$
$$\dots$$
$$\mathbf{z}^{(n)} = f^{(n)}(\mathbf{z}^{(n-1)})$$

NICE: Non-linear Independent Components Estimation

Key concepts:

- Let x be some high dimensional vector, and f(x) map it into new space such the resulting distribution factorizes, i.e. the components of **h** are independent

$$\mathbf{h} = f(\mathbf{x}) \qquad p(\mathbf{h}) = \prod_d p_d(h_d)$$

- f - is an invertible and continuous function (bijection), **x** and **h** are vectors of the same dimensionality
- The change of variables gives us:

$$p(\mathbf{x}) = p(\mathbf{h}) \left| \det \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}} \right|$$

- Sampling is trivial:

$$h \sim p_H(h)$$
$$x = f^{-1}(h)$$

Simple 1D “derivation”:

$$p(x)dx = p(h)dh$$
$$p(x) = p(h)\frac{dh}{dx} = p(h)\frac{df(x)}{dx}$$

NICE: Non-linear Independent Components Estimation

Key novelty:

- f with "**easy determinant of the Jacobian**" and "**easy inverse**"
- Forward pass: y=f(**x**) and **x** (D dimensional vector) is split into two blocks (**x**1, **x**2)

$$y_1 = x_1$$
$$y_2 = x_2 + m(x_1)$$

$$\mathbf{x} = \text{concat}\,(\mathbf{x}_1, \mathbf{x}_2)$$
$$\mathbf{y} = \text{concat}\,(\mathbf{y}_1, \mathbf{y}_2)$$

- m(x) is any function e.g. a neural network, which maps d-dimensional vector into D-d vector
- Inverse:

$$x_1 = y_1$$
$$x_2 = y_2 - m(y_1)$$

- Forward/inverse transformation Jacobian is triangular matrix with ones on the diagonal - a volume preserving transformation

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{pmatrix} 1 & & \\ & \ddots & \\ \frac{dy_2}{dx_1} & & 1 \end{pmatrix}$$

NICE: Non-linear Independent Components Estimation

Objective: maximum likelihood

- The probability of observing data is:

$$p(\mathbf{x}) = p(\mathbf{h}) \left| \det \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}} \right|$$

- We want to maximize logp(**x**) - remember this minimizes KL divergence between model distribution and data distribution:

$$\log(p_X(x)) = \log(p_H(f(x))) + \log(|\det(\frac{\partial f(x)}{\partial x})|)$$

prior distribution

zero for volume preserving operations

- If prior factorizes we have:

$$\log(p_X(x)) = \sum_{d=1}^{D} \log(p_{H_d}(f_d(x))) + \log(|\det(\frac{\partial f(x)}{\partial x})|)$$

NICE: Non-linear Independent Components Estimation

Prior distribution:

- Final objective is

$$\log(p_X(x)) = \sum_{d=1}^{D} \log(p_{H_d}(f_d(x))) + \log(|\det(\frac{\partial f(x)}{\partial x})|)$$

3.4 PRIOR DISTRIBUTION

As mentioned previously, we choose the prior distribution to be factorial, i.e.:

$$p_H(h) = \prod_{d=1}^{D} p_{H_d}(h_d)$$

We generally pick this distribution in the family of standard distribution, e.g. gaussian distribution:

$$\log(p_{H_d}) = -\frac{1}{2}(h_d^2 + \log(2\pi))$$

or logistic distribution:

$$\log(p_{H_d}) = -\log(1 + \exp(h_d)) - \log(1 + \exp(-h_d))$$

We tend to use the logistic distribution as it tends to provide a better behaved gradient.

NICE: Non-linear Independent Components Estimation

Architecture

- **Transformation function**: additive coupling layer (ACL)

$$y_1 = x_1$$
$$y_2 = x_2 + m(x_1)$$

- **m(x)** - is rectified neural network (a ReLU MLP)
- **Combining transformations**: chaining ACLs will remain some of the inputs unchanged, the authors propose to exchange the order of x1, x2 in alternating layers.

$$\left[\mathbf{y}_1^{(1)}, \mathbf{y}_2^{(1)}\right] \xrightarrow{f^1} \left[\mathbf{y}_1^{(2)}, \mathbf{y}_2^{(2)}\right] \xrightarrow{\text{swap}} \left[\mathbf{y}_2^{(2)}, \mathbf{y}_1^{(2)}\right] \xrightarrow{f^2} \left[\mathbf{y}_2^{(3)}, \mathbf{y}_1^{(3)}\right] \rightarrow \cdots$$

authors use 4 such steps

- **Rescaling output**: ACL does not change volume, the authors propose to add volume changing transformation at the last layer, this allows to add weights on some dimensions

$$\mathbf{h} = \exp(\mathbf{s}) \odot \mathbf{y}^{(n)} \qquad \log\_\det\_\text{jacobian} = \sum_i \log \exp(\mathbf{s}) = \sum_i s_i$$

NICE: Non-linear Independent Components Estimation

Architecture

- **Input data preparation:** MNIST add noise 1/256 and rescale to range [0, 1]

Dataset	MNIST	TFD	SVHN	CIFAR-10
# dimensions	784	2304	3072	3072
Preprocessing	None	Approx. whitening	ZCA	ZCA
# hidden layers	5	4	4	4
# hidden units	1000	5000	2000	2000
Prior	logistic	gaussian	logistic	logistic
Log-likelihood	1980.50	5514.71	11496.55	5371.78

Figure 3: Architecture and results. # hidden units refer to the number of units per hidden layer.

NICE: Non-linear Independent Components Estimation

Architecture - full definition (a MNIST case)

- **The model:**

$$h^{(1)}_{I_1} = x_{I_1}$$
$$h^{(1)}_{I_2} = x_{I_2} + m^{(1)}(x_{I_1})$$

$$h^{(2)}_{I_2} = h^{(1)}_{I_2}$$
$$h^{(2)}_{I_1} = h^{(1)}_{I_1} + m^{(2)}(x_{I_2})$$

$$h^{(3)}_{I_1} = h^{(2)}_{I_1}$$
$$h^{(3)}_{I_2} = h^{(2)}_{I_2} + m^{(3)}(x_{I_1})$$

$$h^{(4)}_{I_2} = h^{(3)}_{I_2}$$
$$h^{(4)}_{I_1} = h^{(3)}_{I_1} + m^{(4)}(x_{I_2})$$

$$h = \exp(s) \odot h^{(4)}$$

- m(x) - are NNs - with 5 layers, 1000 units and ReLU
- loss function (to maximize):

$$\log(p_H(h)) + \sum_{i=1}^{D} s_i$$

- the prior:

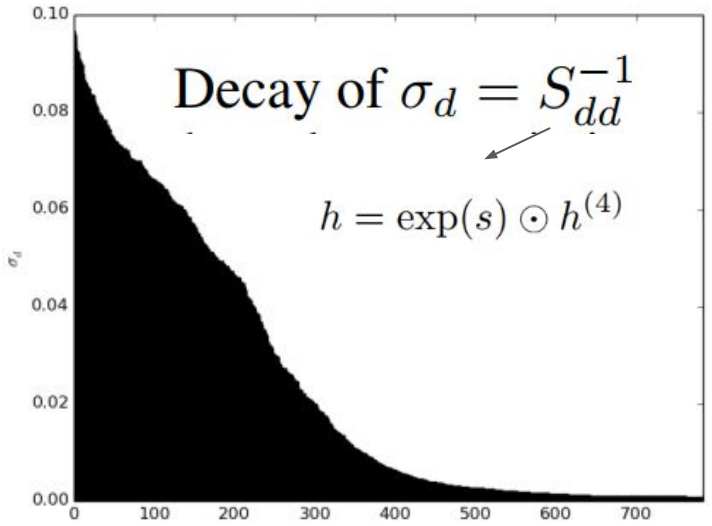
$$\log(p_{H_d}) = -\log(1+\exp(h_d)) - \log(1+\exp(-h_d))$$

- optimizer: Adam
- they needed 1500 epochs

NICE: Non-linear Independent Components Estimation

Results:

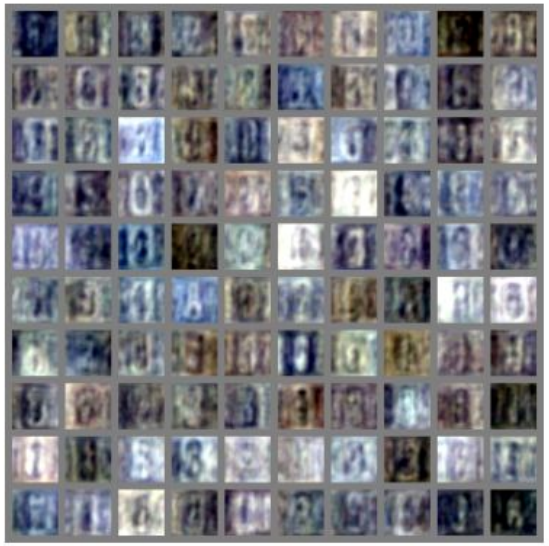
- **Good as for the 2015**



(a) Model trained on MNIST

(a) Model trained on MNIST

(b) Model trained on TFD



(c) Model trained on SVHN

(d) Model trained on CIFAR-10

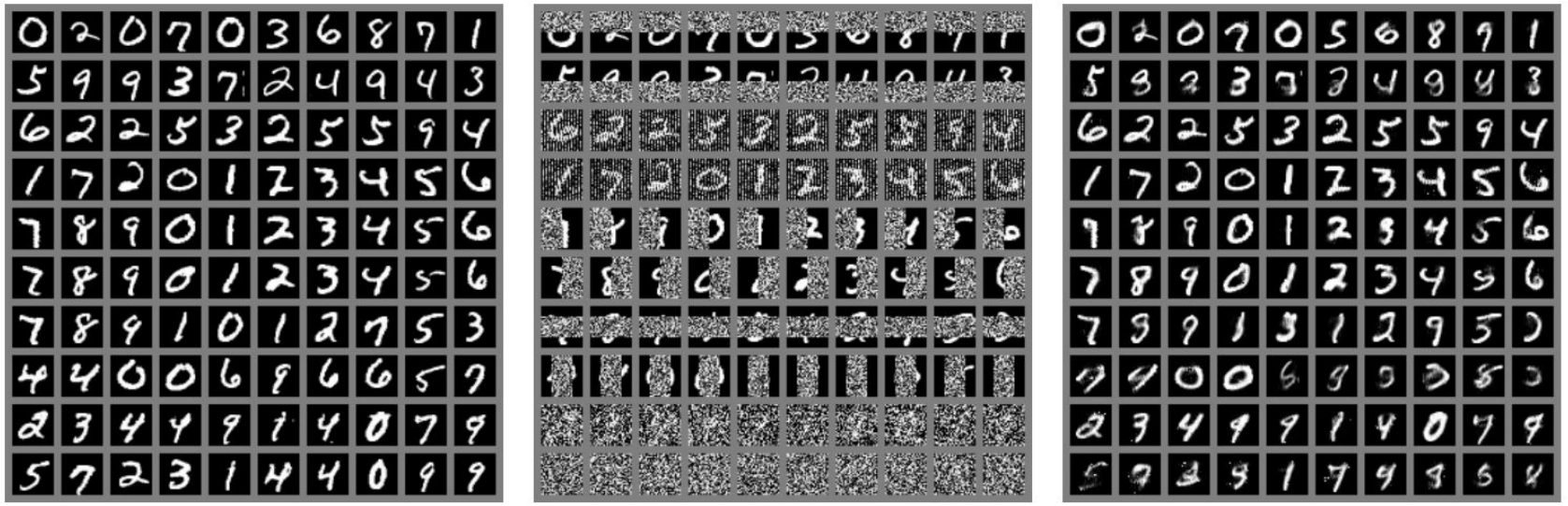
NICE: Non-linear Independent Components Estimation

Results:

- **Inpainting:** compute optimal xH which maximizes likelihood: this can be done with gradient descent

$$x_{H,i+1} = x_{H,i} + \alpha_i \left(\frac{\partial \log(p_X((x_O, x_{H,i})))}{\partial x_{H,i}} + \epsilon \right)$$

$$\epsilon \sim \mathcal{N}(0, I)$$



(a) MNIST test examples (b) Initial state (c) MAP inference of the state

Figure 6: Inpainting on MNIST. We list below the type of the part of the image masked per line of the above middle figure, from top to bottom: top rows, bottom rows, odd pixels, even pixels, left side, right side, middle vertically, middle horizontally, 75% random, 90% random. We clamp the pixels that are not masked to their ground truth value and infer the state of the masked pixels by projected gradient ascent on the likelihood. Note that with middle masks, there is almost no information available about the digit.

Case study: Density estimation using Real NVP (2016)

Density estimation using Real NVP

Jascha Sohl-Dickstein
Google Brain

Samy Bengio
Google Brain

Laurent Dinh*
Montreal Institute for Learning Algorithms
University of Montreal
Montreal, QC H3T1J4

ABSTRACT

Unsupervised learning of probabilistic models is a central yet challenging problem in machine learning. Specifically, designing models with tractable learning, sampling, inference and evaluation is crucial in solving this task. We extend the space of such models using real-valued non-volume preserving (real NVP) transformations, a set of powerful, stably invertible, and learnable transformations, resulting in an unsupervised learning algorithm with exact log-likelihood computation, exact and efficient sampling, exact and efficient inference of latent variables, and an interpretable latent space. We demonstrate its ability to model natural images on four datasets through sampling, log-likelihood evaluation, and latent variable manipulations.

Waffling: zero content text here

$$\log p(\mathbf{x})$$

Density estimation using Real NVP

The same approach, authors use change of variables to model complex densities

3.1 Change of variable formula

Given an observed data variable $x \in X$, a simple prior probability distribution p_Z on a latent variable $z \in Z$, and a bijection $f : X \to Z$ (with $g = f^{-1}$), the change of variable formula defines a model distribution on X by

$$p_X(x) = p_Z\big(f(x)\big) \left| \det \left(\frac{\partial f(x)}{\partial x^T} \right) \right| \tag{2}$$

$$\log\big(p_X(x)\big) = \log\Big(p_Z\big(f(x)\big)\Big) + \log\left(\left| \det \left(\frac{\partial f(x)}{\partial x^T} \right) \right| \right), \tag{3}$$

where $\frac{\partial f(x)}{\partial x^T}$ is the Jacobian of f at x.

Density estimation using Real NVP

Architecture: **Coupling layers: an extension to NICE work**

- **Coupling layer** is a bijection (an invertible function) with easy to compute Jacobian: diagonal matrix

$$\mathbf{y}_1 = \mathbf{x}_1$$
$$\mathbf{y}_2 = \mathbf{x}_2 \odot \exp\left(s\left(\mathbf{x}_1\right)\right) + t\left(\mathbf{x}_1\right)$$

scale - some NN
(change in volume)

scale - some NN

- **x, y** are partitioned in similar manner as in NICE

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{pmatrix} \mathbf{1} & 0 \\ \frac{\partial \mathbf{y}_2}{\partial \mathbf{x}_1} & \operatorname{diag} \exp\left(s\left(\mathbf{x}_1\right)\right) \end{pmatrix}$$

- **Jacobian:**

$$\prod_{i}^{D-d} \exp\left(s\left(\mathbf{x}_1\right)\right)_i$$

where D is the size of **x/y** and D-d size of $\mathbf{x}_2/\mathbf{y}_2$

Density estimation using Real NVP

Architecture: **Coupling layers: an extension to NICE work**

- **Coupling layer** is a bijection (an invertible function) with easy to compute Jacobian: diagonal matrix

$$\mathbf{y}_1 = \mathbf{x}_1$$
$$\mathbf{y}_2 = \mathbf{x}_2 \odot \exp\left(s\left(\mathbf{x}_1\right)\right) + t\left(\mathbf{x}_1\right)$$

scale - some NN (change in volume)

scale - some NN

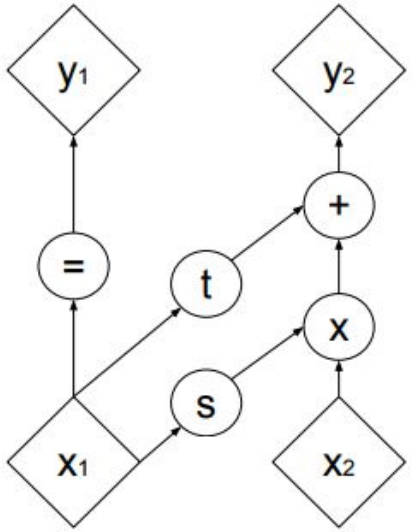
- **Inverse:** is as easy to compute as forward flow - easy inference and sampling

$$\mathbf{x}_1 = \mathbf{y}_1$$
$$\mathbf{x}_2 = \left(\mathbf{y}_2 - t\left(\mathbf{y}_1\right)\right) / \exp\left(s\left(\mathbf{y}_1\right)\right)$$

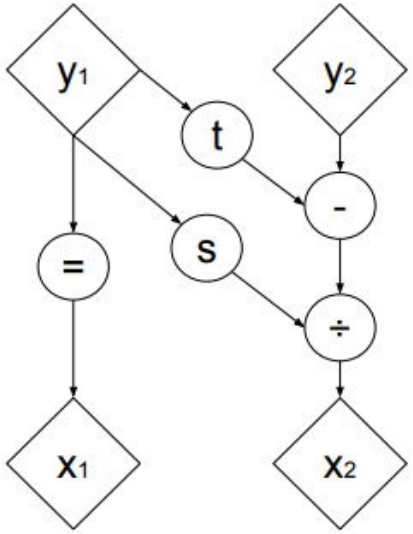
Density estimation using Real NVP

Architecture: **Coupling layers: an extension to NICE work**

- **Forward and inverse flows**



(a) Forward propagation



(b) Inverse propagation

$$\mathbf{y}_1 = \mathbf{x}_1$$
$$\mathbf{y}_2 = \mathbf{x}_2 \odot \exp\left(s\left(\mathbf{x}_1\right)\right) + t\left(\mathbf{x}_1\right)$$

$$\mathbf{x}_1 = \mathbf{y}_1$$
$$\mathbf{x}_2 = \left(\mathbf{y}_2 - t\left(\mathbf{y}_1\right)\right) / \exp\left(s\left(\mathbf{y}_1\right)\right)$$

Density estimation using Real NVP

Architecture: **Partitioning input vector**

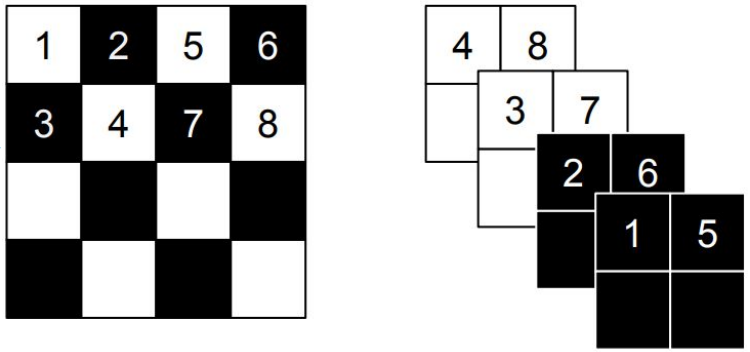
- **Masking convolutions:** instead of explicitly partitioning x we can use binary mask

$$\mathbf{y}_1 = \mathbf{x}_1$$
$$\mathbf{y}_2 = \mathbf{x}_2 \odot \exp\left(s\left(\mathbf{x}_1\right)\right) + t\left(\mathbf{x}_1\right)$$

- Same equation expressed in terms of binary mask

$$\mathbf{y} = \mathbf{b} \odot \mathbf{x} + (1 - \mathbf{b}) \odot \{\mathbf{x} \odot \exp\left(s\left(\mathbf{b} \odot \mathbf{x}\right)\right) + t\left(\mathbf{b} \odot \mathbf{x}\right)\}$$

binary mask



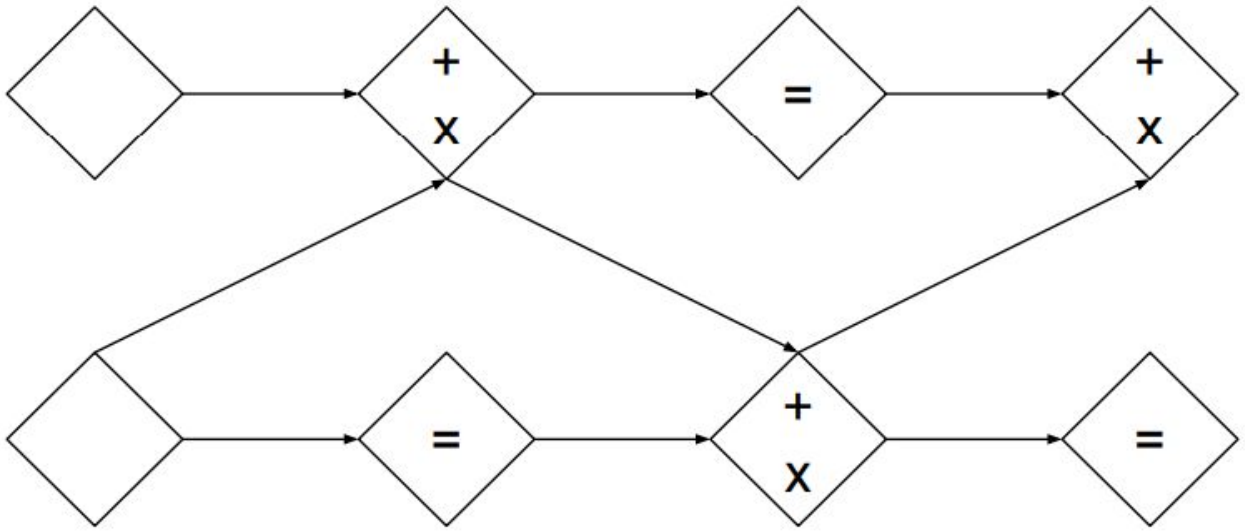
- checkerboard pattern
- channel wise masking

Figure 3: Masking schemes for affine coupling layers. On the left, a spatial checkerboard pattern mask. On the right, a channel-wise masking. The squeezing operation reduces the $4 \times 4 \times 1$ tensor (on the left) into a $2 \times 2 \times 4$ tensor (on the right). Before the squeezing operation, a checkerboard pattern is used for coupling layers while a channel-wise masking pattern is used afterward.

Density estimation using Real NVP

Architecture: **Combining multiple coupling layers**

- In order to enhance mixing between variables, multiple layers are stacked in alternating way



(a) In this alternating pattern, units which remain identical in one transformation are modified in the next.

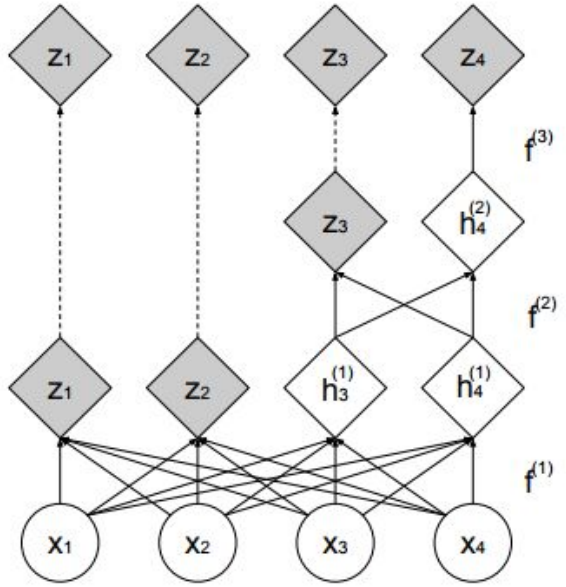
Density estimation using Real NVP

Architecture: **multiscale architecture**

- Multiscale architecture is implemented by squeezing operation together with factoring out half of variables:
- Squeezing operation change tensor shape from `[s, s, c]` to `[s/2, s/2, 4c]`
- They factor out half of the dimensions at regular intervals:

$$
\begin{aligned}
h^{(0)} &= x \\
(z^{(i+1)}, h^{(i+1)}) &= f^{(i+1)}(h^{(i)}) \\
z^{(L)} &= f^{(L)}(h^{(L-1)}) \\
z &= (z^{(1)}, \ldots, z^{(L)}).
\end{aligned}
$$

z variables are suppose to come from Gaussian distribution



(b) Factoring out variables. At each step, half the variables are directly modeled as Gaussians, while the other half undergo further transformation.

- This force the model to gaussianize some of the units at different scales
- This also reduces memory requirements, and we can train larger models

Density estimation using Real NVP

github: tensorflow/models//research/real_nvp

Architecture:

- They use Batch Norm, Weight Norm with Deep Residual Networks in s and t functions
- They use Batch Norm bijector after each coupling layer (see paper for more details)
- They convert input image to logit space, use Adam, and L2 regularization

1. Prepare input:

```
logit_x_in = 2. * x_in  # [0, 2]
logit_x_in -= 1.  # [-1, 1]
logit_x_in *= data_constraint  # [-.9, .9]
logit_x_in += 1.  # [.1, 1.9]
logit_x_in /= 2.  # [.05, .95]
# logit the data
logit_x_in = tf.log(logit_x_in) - tf.log(1. - logit_x_in)
```

2. Transform input to prior density space

```
# INFERENCE AND COSTS
z_out, log_diff = encoder(
    input_=logit_x_in, hps=hps, n_scale=hps.n_scale,
    use_batch_norm=hps.use_batch_norm, weight_norm=True,
    train=True)
```

3. Define cost function: -logp(x)

```
prior_ll = standard_normal_ll(z_out)
prior_ll = tf.reduce_sum(prior_ll, [1, 2, 3])
log_diff = tf.reduce_sum(log_diff, [1, 2, 3])
log_diff += transform_cost
cost = -(prior_ll + log_diff)
```

4. Compute gradients

```
grads_and_vars = optimizer.compute_gradients(
    cost + hps.l2_coeff * l2_reg,
    tf.trainable_variables())
grads, vars_ = zip(*grads_and_vars)
capped_grads, gradient_norm = tf.clip_by_global_norm(
    grads, clip_norm=hps.clip_gradient)
```

Density estimation using Real NVP

Results: samples

dataset

sampled

Density estimation using Real NVP

Results: manifold interpolations

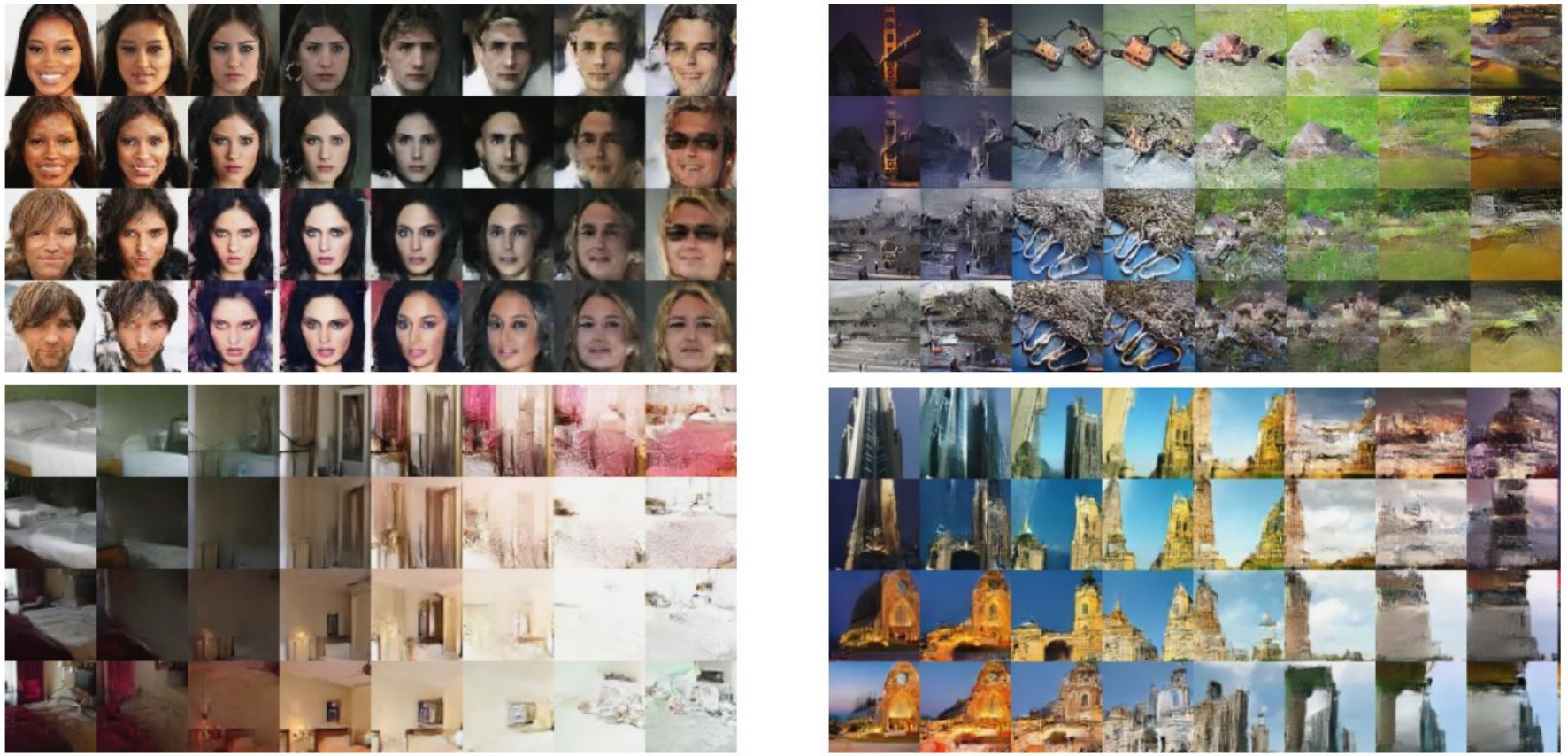


Figure 6: Manifold generated from four examples in the dataset. Clockwise from top left: CelebA, Imagenet (64×64), LSUN (tower), LSUN (bedroom).

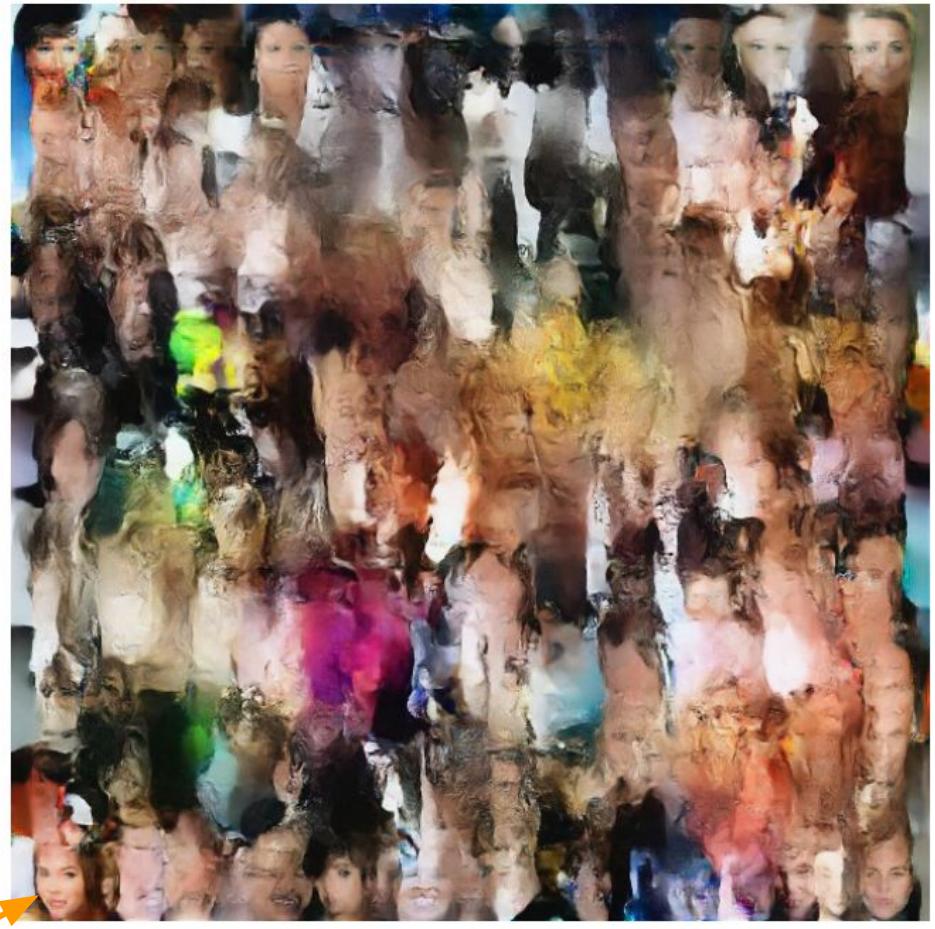
Key advantages of RealNVP:

- fast inference
- fast sampling
- easy to compute jacobian

Density estimation using Real NVP

Results: funny extrapolations - generating images for higher resolutions than during training (more examples in paper)

(a) ×2



(b) ×10

Best faces at the edges, in my opinion this is an effect of padding

Figure 18: We generate samples a factor bigger than the training set image size on *CelebA*.

Case study: Glow: Generative Flow with Invertible 1x1 Convolutions (2018)

Glow: Generative Flow with Invertible 1x1 Convolutions

Diederik P. Kingma*, **Prafulla Dhariwal***
OpenAI, San Francisco

Abstract

Flow-based generative models (Dinh et al., 2014) are conceptually attractive due to tractability of the exact log-likelihood, tractability of exact latent-variable inference, and parallelizability of both training and synthesis. In this paper we propose *Glow*, a simple type of generative flow using an invertible 1×1 convolution. Using our method we demonstrate a significant improvement in log-likelihood on standard benchmarks. Perhaps most strikingly, we demonstrate that a generative model optimized towards the plain log-likelihood objective is capable of efficient realistic-looking synthesis and manipulation of large images. The code for our model is available at `https://github.com/openai/glow`.

NICE

a continuous generalization for discrete rotations

Glow: Generative Flow with Invertible 1x1 Convolutions

Summary of likelihood-based generative models:

The discipline of generative modeling has experienced enormous leaps in capabilities in recent years, mostly with likelihood-based methods (Graves, 2013; Kingma and Welling, 2013, 2018; Dinh et al., 2014; van den Oord et al., 2016a) and generative adversarial networks (GANs) (Goodfellow et al., 2014) (see Section 4). Likelihood-based methods can be divided into three categories:

1. Autoregressive models (Hochreiter and Schmidhuber, 1997; Graves, 2013; van den Oord et al., 2016a,b; Van Den Oord et al., 2016). Those have the advantage of simplicity, but have as disadvantage that synthesis has limited parallelizability, since the computational length of synthesis is proportional to the dimensionality of the data; this is especially troublesome for large images or video.

2. Variational autoencoders (VAEs) (Kingma and Welling, 2013, 2018), which optimize a lower bound on the log-likelihood of the data. Variational autoencoders have the advantage of parallelizability of training and synthesis, but can be comparatively challenging to optimize (Kingma et al., 2016).

3. Flow-based generative models, first described in NICE (Dinh et al., 2014) and extended in RealNVP (Dinh et al., 2016). We explain the key ideas behind this class of model in the following sections.

Glow: Generative Flow with Invertible 1x1 Convolutions

Summary of flow based methods

Flow-based generative models have so far gained little attention in the research community compared to GANs (Goodfellow et al., 2014) and VAEs (Kingma and Welling, 2013). Some of the merits of flow-based generative models include:

- Exact latent-variable inference and log-likelihood evaluation. In VAEs, one is able to infer only approximately the value of the latent variables that correspond to a datapoint. GAN's have no encoder at all to infer the latents. In reversible generative models, this can be done exactly without approximation. Not only does this lead to accurate inference, it also enables optimization of the exact log-likelihood of the data, instead of a lower bound of it.

- Efficient inference and efficient synthesis. Autoregressive models, such as the PixelCNN (van den Oord et al., 2016b), are also reversible, however synthesis from such models is difficult to parallelize, and typically inefficient on parallel hardware. Flow-based generative models like Glow (and RealNVP) are efficient to parallelize for both inference and synthesis.

- Useful latent space for downstream tasks. The hidden layers of autoregressive models have unknown marginal distributions, making it much more difficult to perform valid manipulation of data. In GANs, datapoints can usually not be directly represented in a latent space, as they have no encoder and might not have full support over the data distribution. (Grover et al., 2018). This is not the case for reversible generative models and VAEs, which allow for various applications such as interpolations between datapoints and meaningful modifications of existing datapoints.

- Significant potential for memory savings. Computing gradients in reversible neural networks requires an amount of memory that is constant instead of linear in their depth, as explained in the RevNet paper (Gomez et al., 2017).

Glow: Generative Flow with Invertible 1x1 Convolutions

Definition of the cost function: MLE

2 Background: Flow-based Generative Models

Let $\mathbf{x}$ be a high-dimensional random vector with unknown true distribution $\mathbf{x} \sim p^*(\mathbf{x})$. We collect an i.i.d. dataset $\mathcal{D}$, and choose a model $p_{\boldsymbol{\theta}}(\mathbf{x})$ with parameters $\boldsymbol{\theta}$. In case of discrete data $\mathbf{x}$, the log-likelihood objective is then equivalent to minimizing:

$$\mathcal{L}(\mathcal{D}) = \frac{1}{N} \sum_{i=1}^{N} -\log p_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}) \tag{1}$$

In case of *continuous* data $\mathbf{x}$, we minimize the following:

$$\mathcal{L}(\mathcal{D}) \simeq \frac{1}{N} \sum_{i=1}^{N} -\log p_{\boldsymbol{\theta}}(\tilde{\mathbf{x}}^{(i)}) + c \tag{2}$$

where $\tilde{\mathbf{x}}^{(i)} = \mathbf{x}^{(i)} + u$ with $u \sim \mathcal{U}(0, a)$, and $c = -M \cdot \log a$ where a is determined by the discretization level of the data and M is the dimensionality of $\mathbf{x}$. Both objectives (eqs. (1) and (2)) measure the expected compression cost in nats or bits; see (Dinh et al., 2016). Optimization is done through stochastic gradient descent using minibatches of data (Kingma and Ba, 2015).

Glow: Generative Flow with Invertible 1x1 Convolutions

Normalizing flows basic definitions

- generative flow is defined as

$$\mathbf{z} \sim p_{\boldsymbol{\theta}}(\mathbf{z})$$
$$\mathbf{x} = \mathbf{g}_{\boldsymbol{\theta}}(\mathbf{z})$$

- Prior distribution: spherical Gaussian: $p_{\boldsymbol{\theta}}(\mathbf{z}) = \mathcal{N}(\mathbf{z}; 0, \mathbf{I})$, g is bijective
- Inference is defined as: $\mathbf{z} = \mathbf{f}_{\boldsymbol{\theta}}(\mathbf{x}) = \mathbf{g}_{\boldsymbol{\theta}}^{-1}(\mathbf{x})$
- Normalizing flow is defined as sequence of transformations:

$$\mathbf{x} \overset{\mathbf{f}_1}{\longleftrightarrow} \mathbf{h}_1 \overset{\mathbf{f}_2}{\longleftrightarrow} \mathbf{h}_2 \cdots \overset{\mathbf{f}_K}{\longleftrightarrow} \mathbf{z}$$

- The log of pdf of the model given data point is then computed as:

$$\log p_{\boldsymbol{\theta}}(\mathbf{x}) = \log p_{\boldsymbol{\theta}}(\mathbf{z}) + \log \left| \det(d\mathbf{z}/d\mathbf{x}) \right|$$
$$= \log p_{\boldsymbol{\theta}}(\mathbf{z}) + \sum_{i=1}^{K} \log \left| \det(d\mathbf{h}_i/d\mathbf{h}_{i-1}) \right|$$

log-determinant of the transformation, this can be easy to compute for certain choices e.g. MADE, NICE, RealNVP

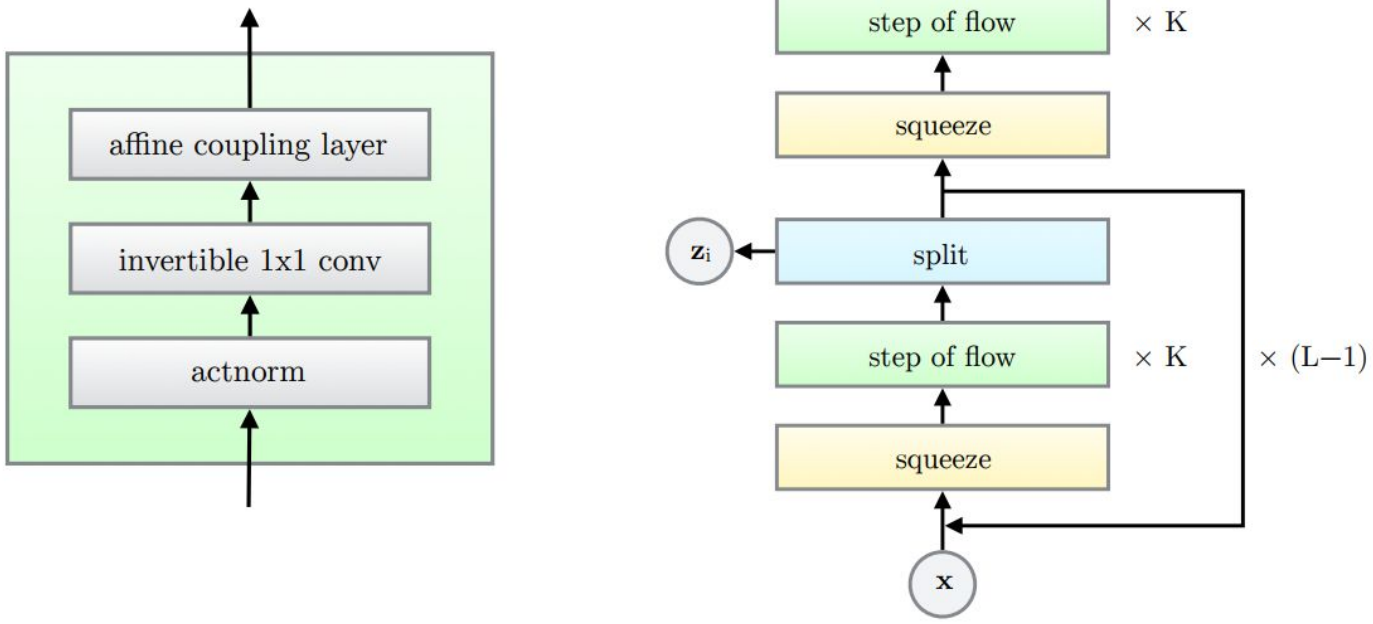
Glow: Generative Flow with Invertible 1x1 Convolutions

Proposed generative flow

- simplified version of NICE and RealNVP works
- multiscale architecture same as in RealNVP
- affine coupling layer

New things:

- actnorm (no Batchnorm)
- data dependent initialization
- invertible 1x1 conv



(a) One step of our flow.

(b) Multi-scale architecture (Dinh et al., 2016).

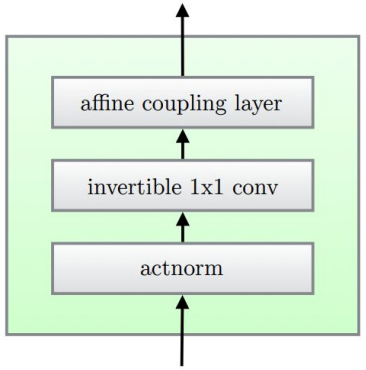
Figure 2: We propose a generative flow where each step (left) consists of an *actnorm* step, followed by an invertible 1×1 convolution, followed by an affine transformation (Dinh et al., 2014). This flow is combined with a multi-scale architecture (right). See Section 3 and Table 1.

Glow: Generative Flow with Invertible 1x1 Convolutions

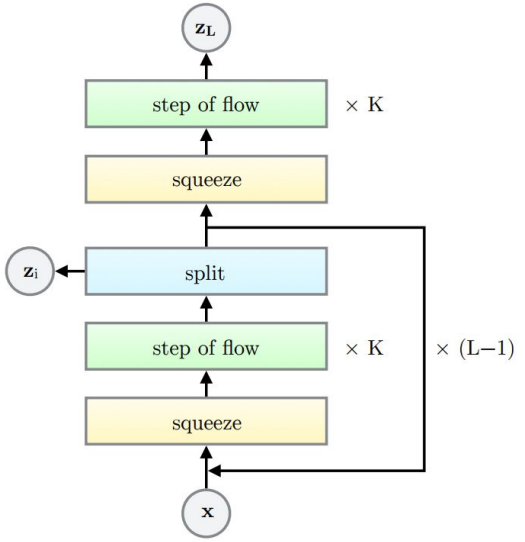
Proposed generative flow blocks: **actnorm - activation normalization**

Description	Function	Reverse Function	Log-determinant
Actnorm. See Section 3.1.	$\forall i,j : \mathbf{y}_{i,j} = \mathbf{s} \odot \mathbf{x}_{i,j} + \mathbf{b}$	$\forall i,j : \mathbf{x}_{i,j} = (\mathbf{y}_{i,j} - \mathbf{b})/\mathbf{s}$	$h \cdot w \cdot \mathbf{sum}(\log \lvert\mathbf{s}\rvert)$

- resigned from Batch Normalization due to large variance of normalization for small batch size
- they can train with batch size 1
- **actnorm**: `i, j` denote spatial index of the feature map `[height, width, num_channels]`
- **initialization of s and b:** initialize them such that post-activations have mean zero and unit variance, then they treat scale and bias as normal trainable parameters



(a) One step of our flow.



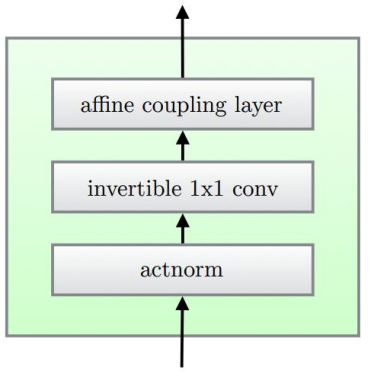
(b) Multi-scale architecture (Dinh et al., 2016).

Glow: Generative Flow with Invertible 1x1 Convolutions

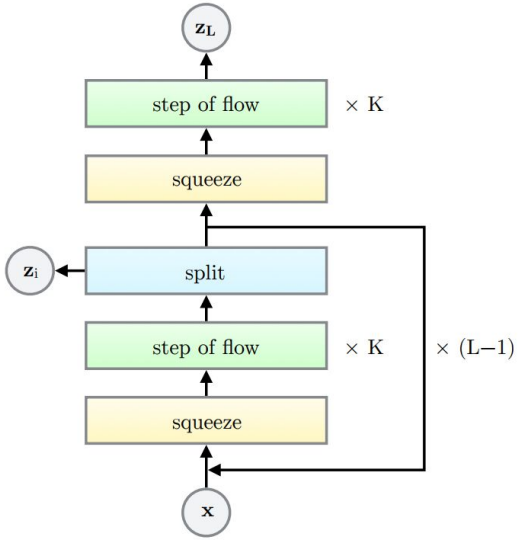
Proposed generative flow blocks: **invertible 1x1 convolution**

Description	Function	Reverse Function	Log-determinant
Invertible 1×1 convolution. $\mathbf{W} : [c \times c]$. See Section 3.2.	$\forall i, j : \mathbf{y}_{i,j} = \mathbf{W}\mathbf{x}_{i,j}$	$\forall i, j : \mathbf{x}_{i,j} = \mathbf{W}^{-1}\mathbf{y}_{i,j}$	$h \cdot w \cdot \log \lvert \det(\mathbf{W}) \rvert$ or $h \cdot w \cdot \mathbf{sum}(\log \lvert \mathbf{s} \rvert)$ (see eq. (10))

- use 1x1 conv instead of fixed reversing of channels, or instead of random permutation
- 1x1 can be considered as Dense layer which rotates and scale channels in continuous manner
- the cost of computing det(**W**) ~O(c^3) is comparable with cost of conv2d O(h*w*c^2)
- they initialize **W** as random rotation matrix, and use LU decomposition to speed up determinant computation (see paper for more details)



(a) One step of our flow.



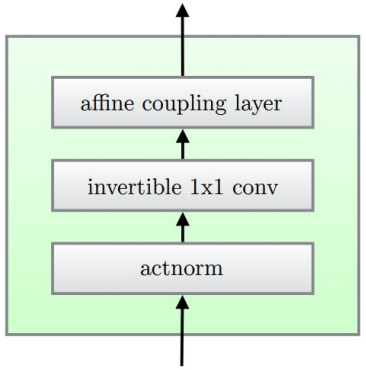
(b) Multi-scale architecture (Dinh et al., 2016).

Glow: Generative Flow with Invertible 1x1 Convolutions

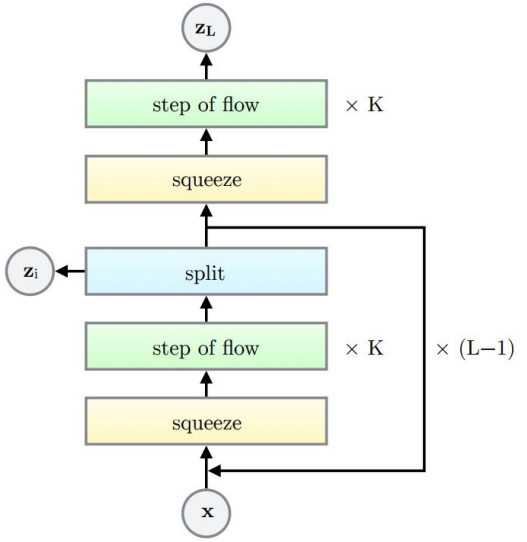
Proposed generative flow blocks: **affine coupling layer**

Description	Function	Reverse Function	Log-determinant
Affine coupling layer. See Section 3.3 and (Dinh et al., 2014)	$\mathbf{x}_a, \mathbf{x}_b = \mathtt{split}(\mathbf{x})$ $(\log \mathbf{s}, \mathbf{t}) = \mathtt{NN}(\mathbf{x}_b)$ $\mathbf{s} = \exp(\log \mathbf{s})$ $\mathbf{y}_a = \mathbf{s} \odot \mathbf{x}_a + \mathbf{t}$ $\mathbf{y}_b = \mathbf{x}_b$ $\mathbf{y} = \mathtt{concat}(\mathbf{y}_a, \mathbf{y}_b)$	$\mathbf{y}_a, \mathbf{y}_b = \mathtt{split}(\mathbf{y})$ $(\log \mathbf{s}, \mathbf{t}) = \mathtt{NN}(\mathbf{y}_b)$ $\mathbf{s} = \exp(\log \mathbf{s})$ $\mathbf{x}_a = (\mathbf{y}_a - \mathbf{t})/\mathbf{s}$ $\mathbf{x}_b = \mathbf{y}_b$ $\mathbf{x} = \mathtt{concat}(\mathbf{x}_a, \mathbf{x}_b)$	$\mathtt{sum}(\log(\lvert\mathbf{s}\rvert))$

- similar as in **RealNVP**
- They use **zero initialization**: last layer of NN is initializer to zero, such that coupling layer is identity function at the beginning
- **split** only along channel dimension (no checkerboard pattern), **concat** is a reverse operation to split
- **no fixed permutations** - they mix channels with invertible 1x1 convolutions
- NN is a shallow, 3 convolutions with 512 units: 3x3, 1x1, 3x3, the last one returns shift and logscale



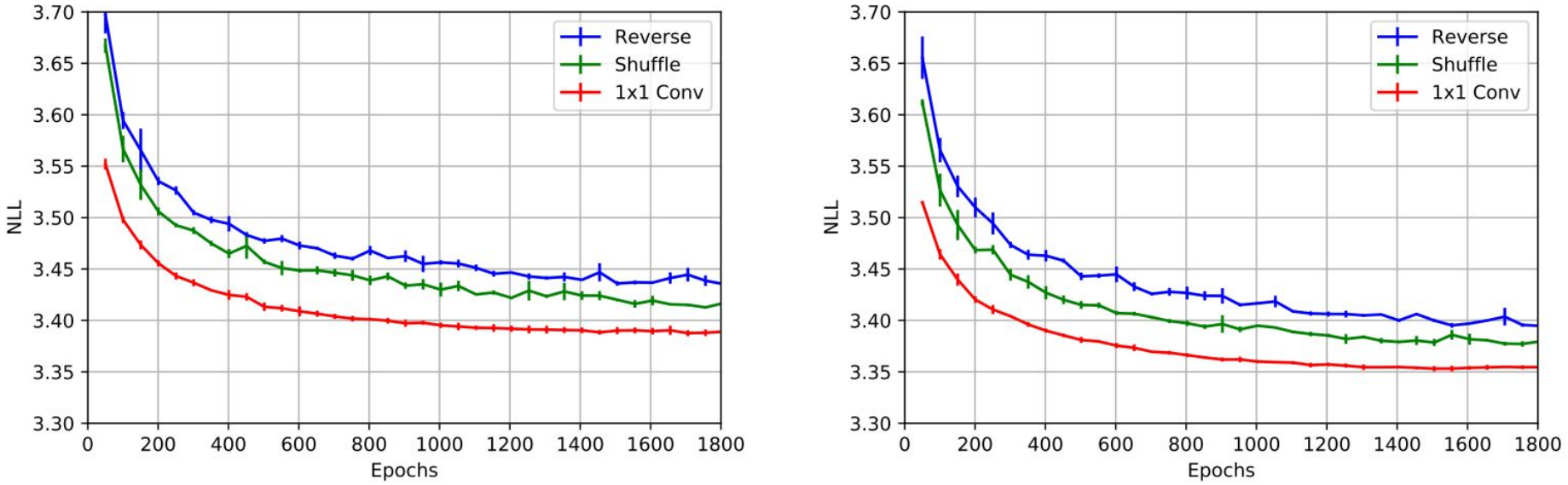
(a) One step of our flow.



(b) Multi-scale architecture (Dinh et al., 2016).

Glow: Generative Flow with Invertible 1x1 Convolutions

Comparison of different permutation methods and affine and additive coupling



(a) Additive coupling.

(b) Affine coupling.

Figure 3: Comparison of the three variants - a reversing operation as described in the RealNVP, a fixed random permutation, and our proposed invertible 1×1 convolution, with additive (left) versus affine (right) coupling layers. We plot the mean and standard deviation across three runs with different random seeds.

- Invertible convolutions were 7% slower than simple permutation
- Much better performance

Glow: Generative Flow with Invertible 1x1 Convolutions

More tricks: Image preprocessing -> image quantization

8 bits

2 bits

5 bits

- Reducing complexity increased the quality in their case

Glow: Generative Flow with Invertible 1x1 Convolutions

Results:

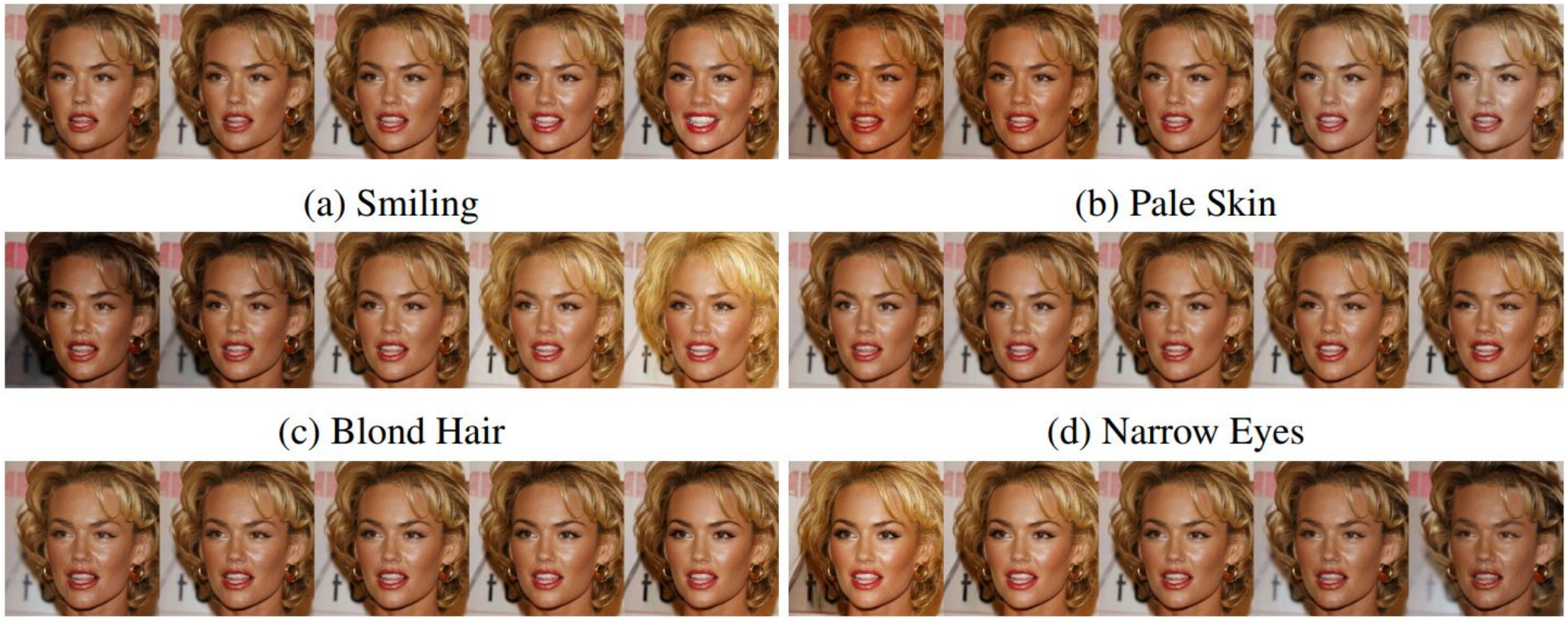
Table 2: Best results in bits per dimension of our model compared to RealNVP.

Model	CIFAR-10	ImageNet 32x32	ImageNet 64x64	LSUN (bedroom)	LSUN (tower)	LSUN (church outdoor)
RealNVP	3.49	4.28	3.98	2.72	2.81	3.08
Glow	**3.35**	**4.09**	**3.81**	**2.38**	**2.46**	**2.67**

Figure 5: Linear interpolation in latent space between real images

Glow: Generative Flow with Invertible 1x1 Convolutions

Results:



(a) Smiling (b) Pale Skin

(c) Blond Hair (d) Narrow Eyes

(e) Young (f) Male

Figure 6: Manipulation of attributes of a face. Each row is made by interpolating the latent code of an image along a vector corresponding to the attribute, with the middle image being the original image

Glow: Generative Flow with Invertible 1x1 Convolutions

Sampling from reduced temperature: **x ~ N(0, T)**

T=0

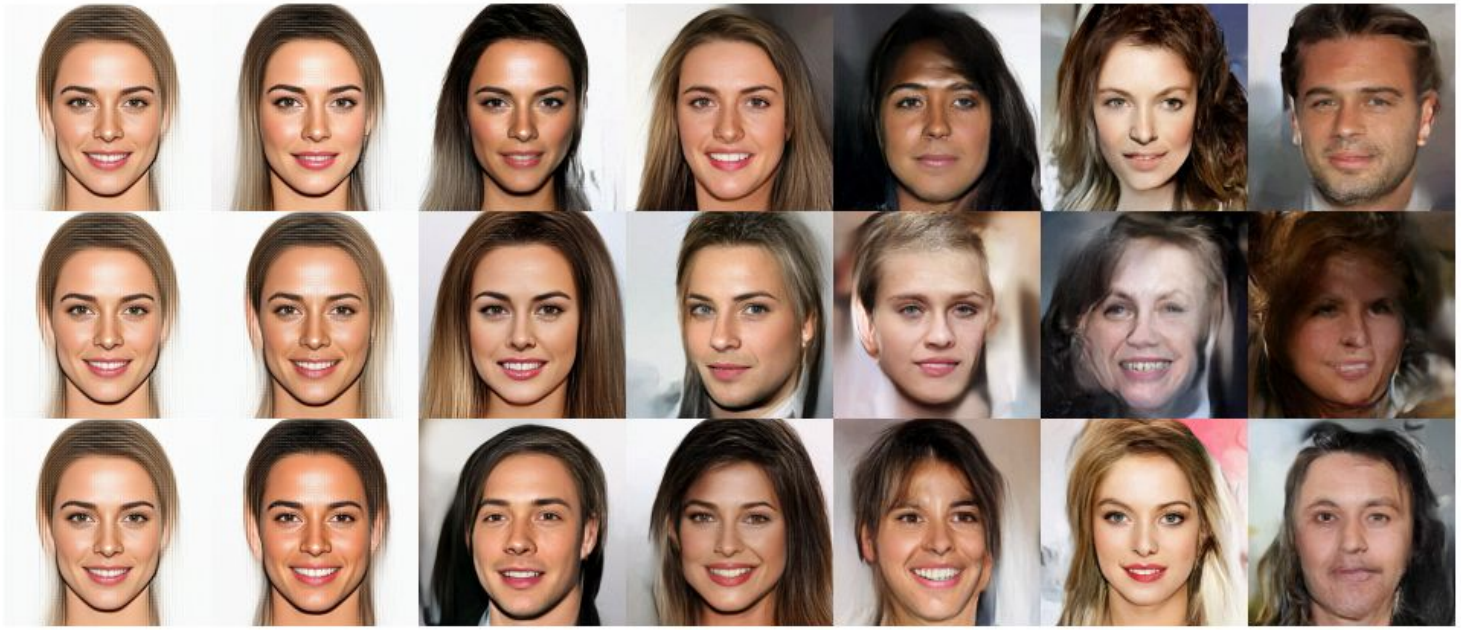


Figure 8: Effect of change of temperature. From left to right, samples obtained at temperatures $0, 0.25, 0.6, 0.7, 0.8, 0.9, 1.0$

- we found that sampling from a reduced-temperature model often results in higher-quality samples
  This may result from imperfect match between data distribution p(z) and unit gaussian (the prior).

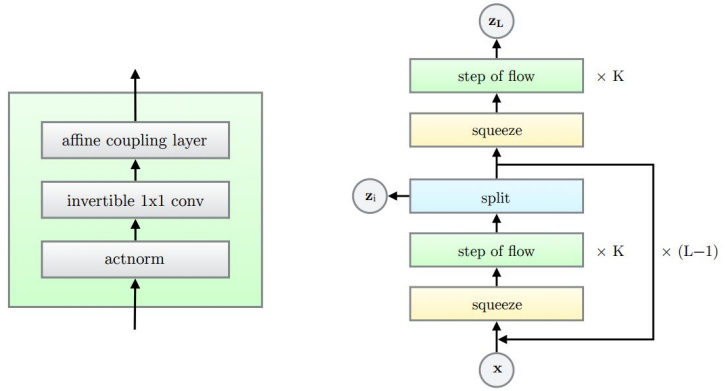
Glow: Generative Flow with Invertible 1x1 Convolutions

Effect of the depth of the model

Figure 9: Samples from shallow model on left vs deep model on right. Shallow model has $L = 4$ levels, while deep model has $L = 6$ levels

Summary:

- K=32, L ~6
- 5 bit images
- gradient checkpointing to save memory
- no batchnorm, DDI
- trained on cluster of 5 machines with 8 GPUs each
- 200M parameters and 600 layers



(a) One step of our flow. (b) Multi-scale architecture (Dinh et al., 2016).

Figure 2: We propose a generative flow where each step (left) consists of an *actnorm* step, followed by an invertible 1×1 convolution, followed by an affine transformation (Dinh et al., 2014). This flow is combined with a multi-scale architecture (right). See Section 3 and Table 1.

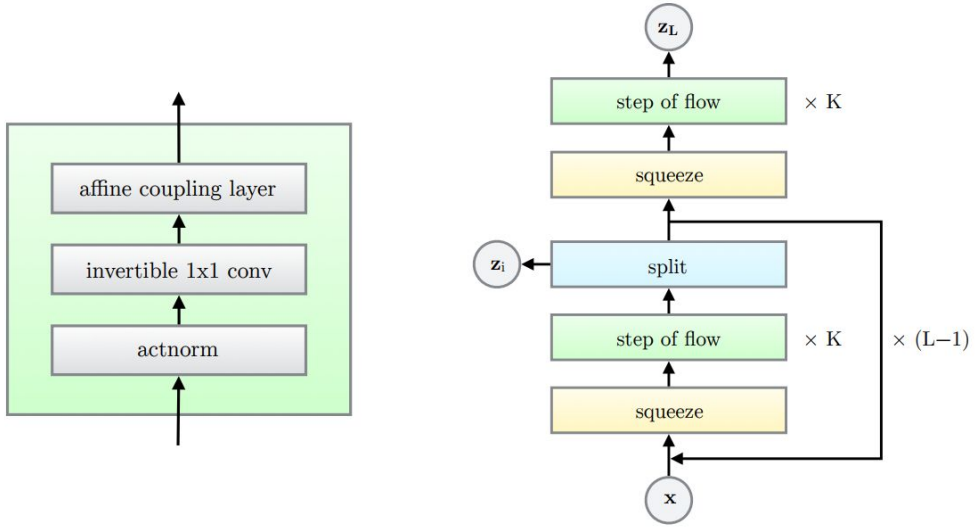
New paper which is under review (ICLR 2019 Conference)

Flow++: Improving Flow-Based Generative Models with Variational Dequantization and Architecture Design

My experiments

- I have re-implemented the openAI GLOW paper (Copy pasted most important functions)
- Simplified and refactored code a little bit
- Flow definition is quite simple (keras like API):

```
layers = [
    fl.QuantizeImage(num_bits=5),
    # START GLOW BLOCKS
    fl.SqueezingLayer(name="L=1"),
    # SCALE L=1
    fl.ChainLayer([
        fl.ChainLayer([
            fl.ActnormLayer(),
            fl.InvertibleConv1x1Layer(),
            fl.AffineCouplingLayer(shift_and_log_scale_fn)
        ], name="K=1"),
        fl.ChainLayer([
            fl.ActnormLayer(),
            fl.InvertibleConv1x1Layer(),
            fl.AffineCouplingLayer(shift_and_log_scale_fn)
        ], name="K=2"),
        # MORE STEPS ...
    ], name="L=1"),
    fl.FactorOutLayer(name="L=1")
    # SCALE K=2
    # ...
]
```



(a) One step of our flow.

(b) Multi-scale architecture (Dinh et al., 2016).

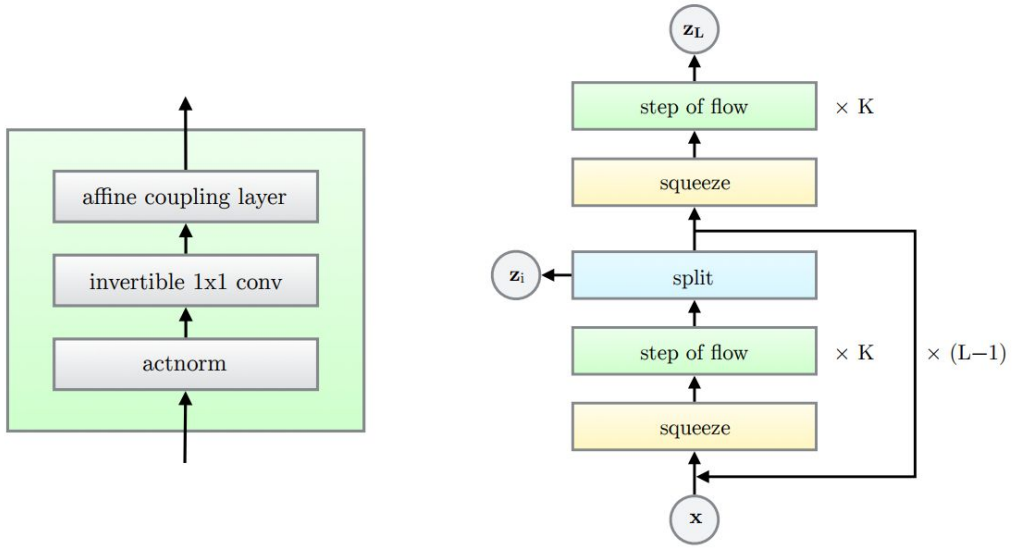
Figure 2: We propose a generative flow where each step (left) consists of an *actnorm* step, followed by an invertible 1×1 convolution, followed by an affine transformation (Dinh et al., 2014). This flow is combined with a multi-scale architecture (right). See Section 3 and Table 1.

My experiments

- I have re-implemented the openAI GLOW paper (Copy pasted most important functions)
- Simplified and refactored code a little bit
- Flow definition is quite simple:

Easy model creation:

```
images = tf.placeholder(tf.float32, [16, 64, 64, 3])
layers, actnorm_layers = nets.create_simple_flow(
    num_steps=32,
    num_scales=4,
    template_fn=nets.OpenAITemplate(width=512)
)
flow = fl.InputLayer(images)
model = fl.ChainLayer(layers)
# forward flow
output_flow = model(flow, forward=True)
# backward flow
reconstruction = model(output_flow, forward=False)
# flow is a tuple of three tensors
x, logdet, z = flow
```



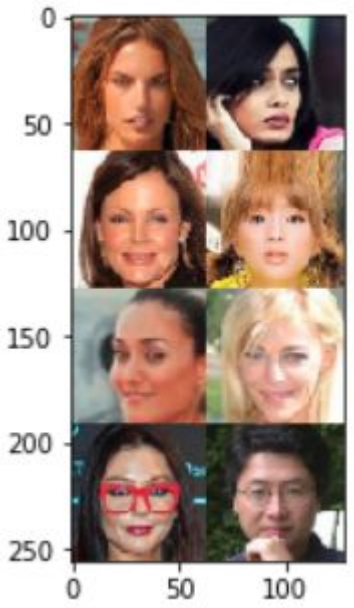
(a) One step of our flow.

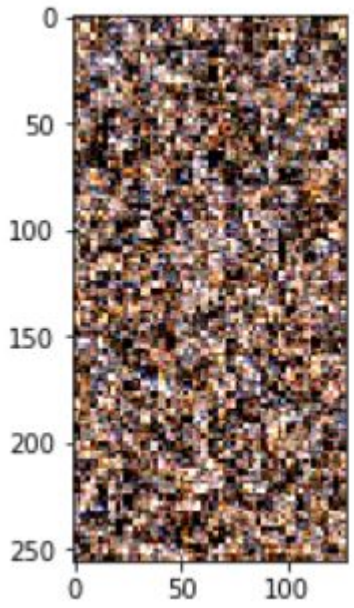
(b) Multi-scale architecture (Dinh et al., 2016).

Figure 2: We propose a generative flow where each step (left) consists of an *actnorm* step, followed by an invertible 1×1 convolution, followed by an affine transformation (Dinh et al., 2014). This flow is combined with a multi-scale architecture (right). See Section 3 and Table 1.

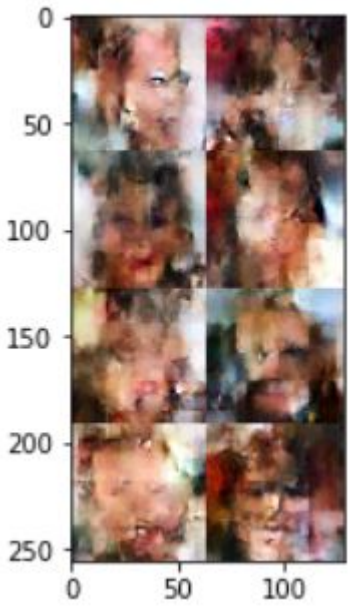
My experiments - selected results

- Depth (K) matters, the more the better,
- The choice of the NN in affine coupling layer, can also matter
- Tested on MNIST dataset and Celeba (64x64)
- Celeba 64x64, model with ~9M params need around 2h for first decent results with GTX1080
- learning rate must be small, no batch norm, hundred of layers, 5 bits for image
- my loss is no compatible with the paper one, I have omitted constant factors which do not affect the gradient

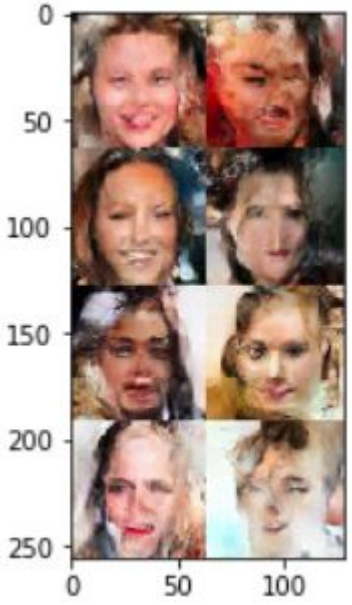




Initial samples



After ~4min

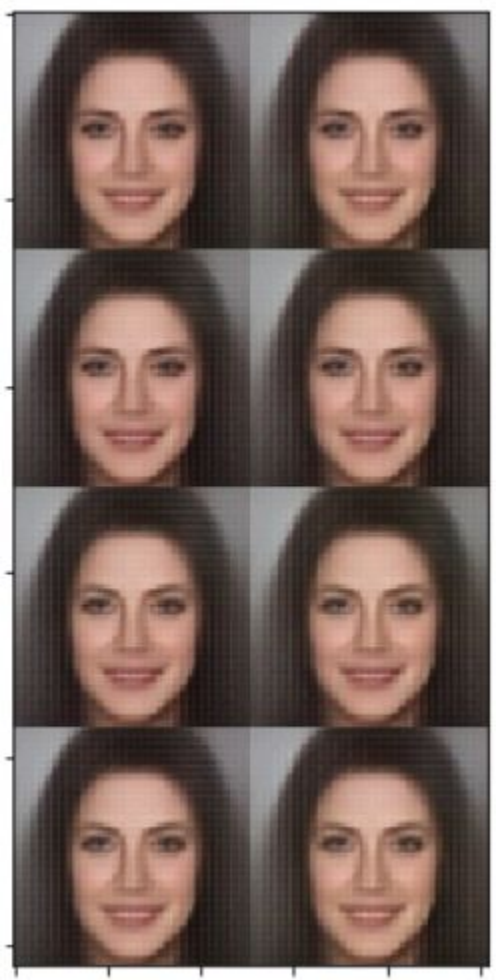


After ~2h, but this is for T=1

My experiments - selected results

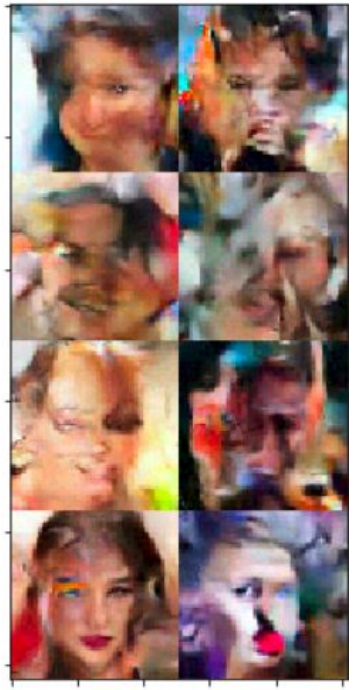
- The effect of the temperature: **z~N(0, T)**

T=0



T=1

T=1.17

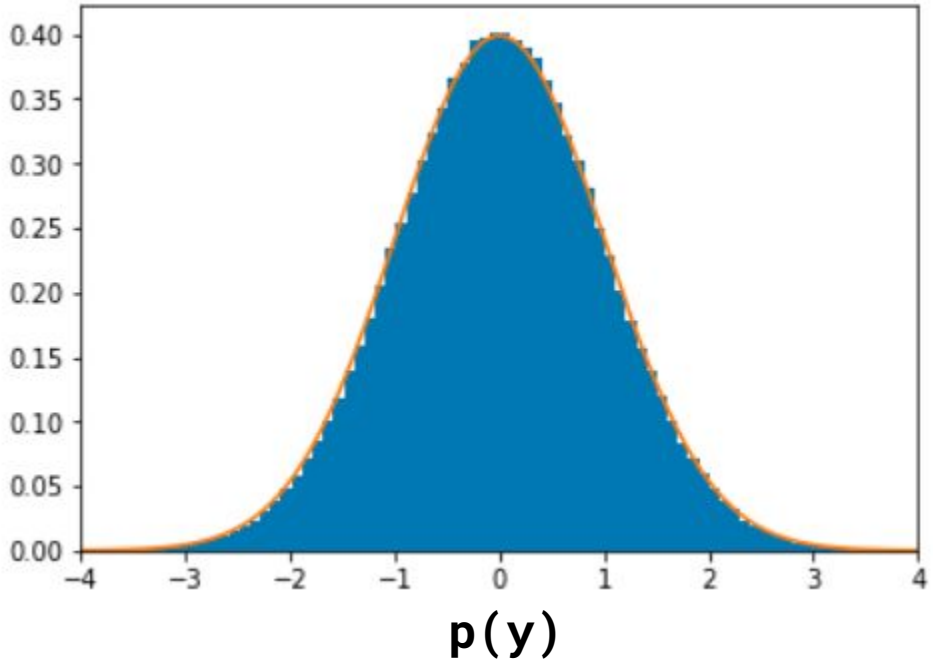


My experiments - selected results

- Checking latent vectors statistics

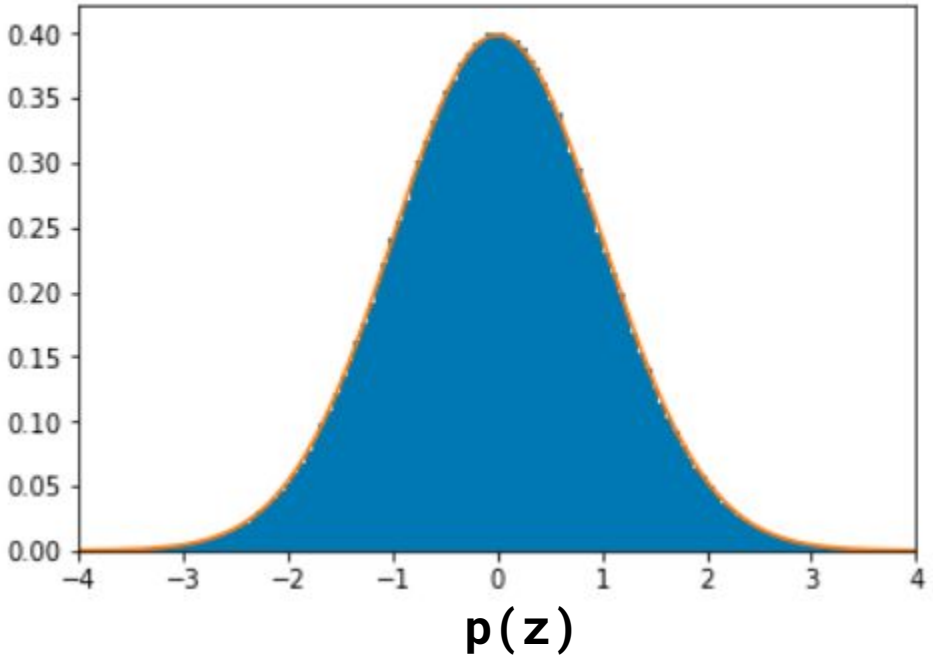
```
input_flow = images, tf.zeros([batch_size]), None
y, logdet, z = model(input_flow, forward=True)
```

- Both **y** and **z** should be factorized Normals



p(y)

- we want to y keep the information about the image



p(z)

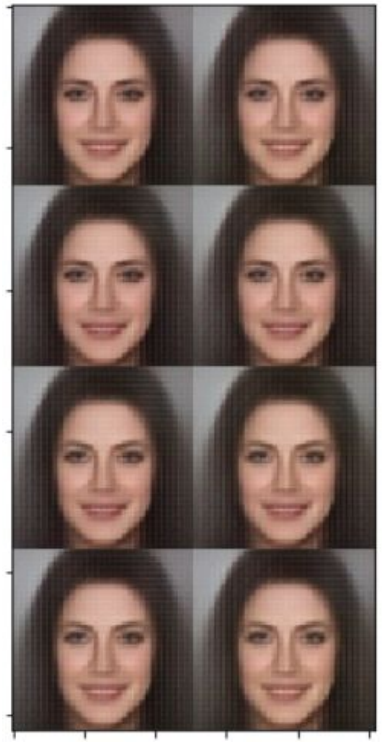
- we want to p(z) keep the noise,
- we could train model with different penalties for p(y) and p(z)

My experiments - selected results

- Checking latent vectors statistics

```
input_flow = images, tf.zeros([batch_size]), None
y, logdet, z = model(input_flow, forward=True)
```

- Both **y** and **z** should be factorized Normals, I have checked samples when sampling from p(y, T=1) and p(z, T=0) i.e. with different temperatures.



p(y,T=0)
p(z,T=0)

p(y,T=1)
p(z,T=0)

p(y,T=1)
p(z,T=0.3)

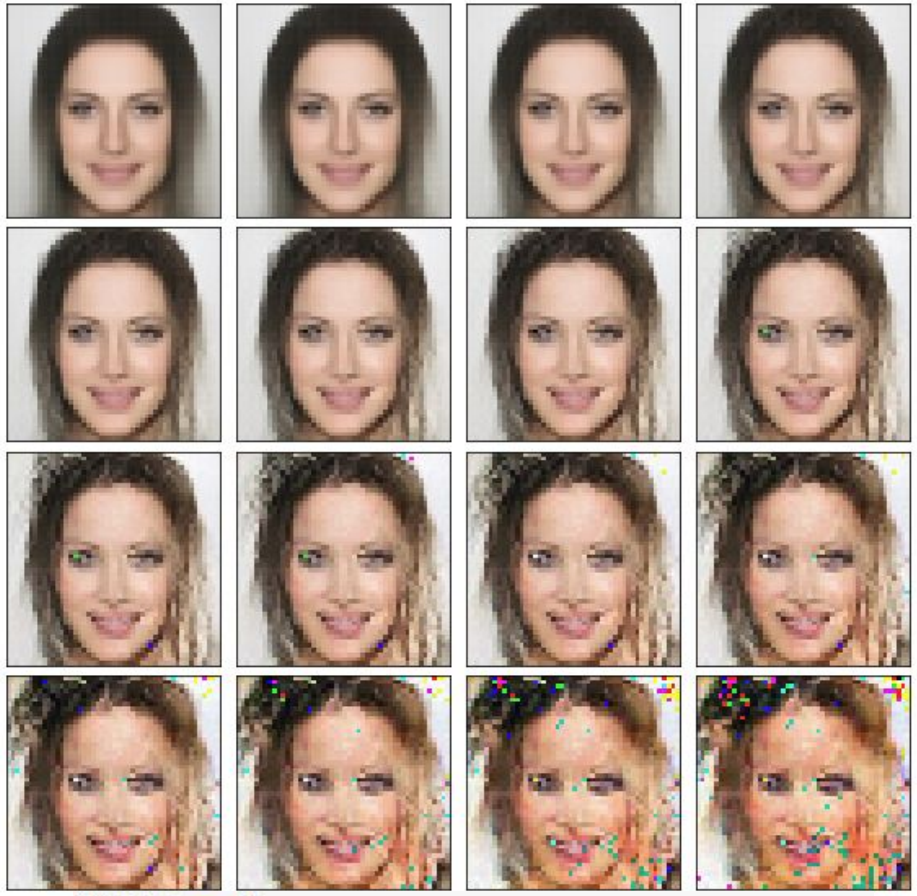
p(y,T=1)
p(z,T=0.5)

p(y,T=1)
p(z,T=1)

My experiments - selected results

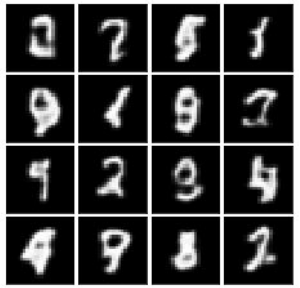
- Testing p(y;T) with p(z;T=0) and the opposite: p(y;T=0) with p(z;T)
- Clearly p(z) contains less information than p(y), which means that NF did information compression!

```
images = []
for w in np.linspace(0, 1.5, 16):
    sampled_image = x_flow_sampled_prim.eval({
        y_prior_ph: w*noise_y,
        z_prior_ph: 0*noise_z
    })[0]
    images.append(sampled_image)
```

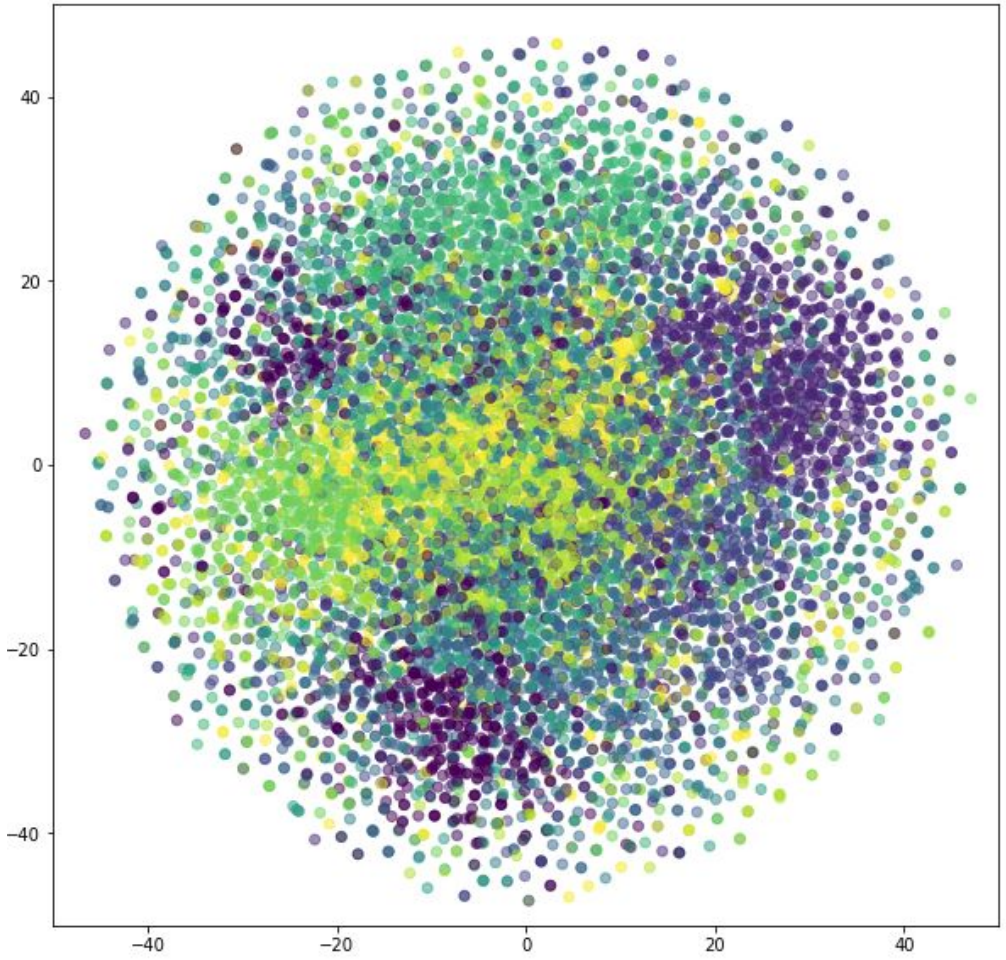


```
images = []
for w in np.linspace(0, 1.5, 16):
    sampled_image = x_flow_sampled_prim.eval({
        y_prior_ph: 0*noise_y,
        z_prior_ph: w*noise_z
    })[0]
    images.append(sampled_image)
```

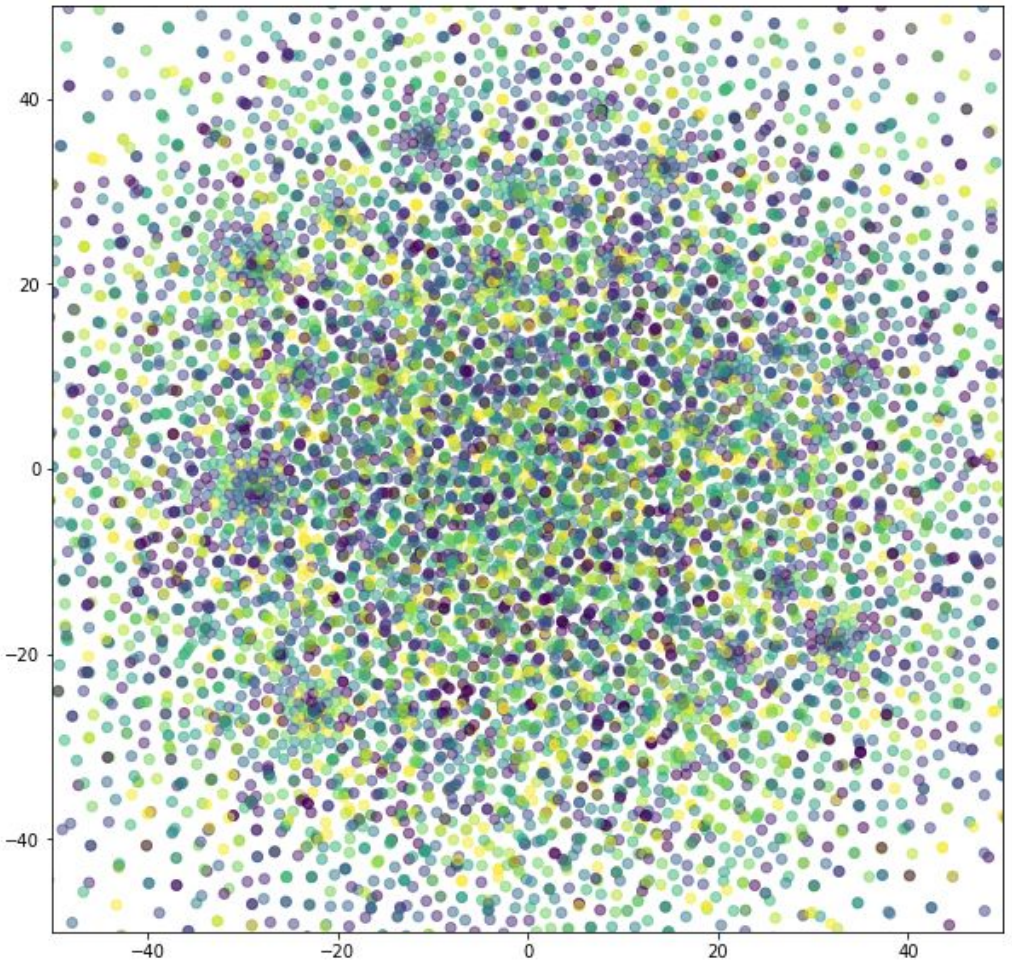
My experiments - selected results



- I checked the same for **MNIST** model
- and plot **tSNE** embeddings for z and y latent variables and for the test set



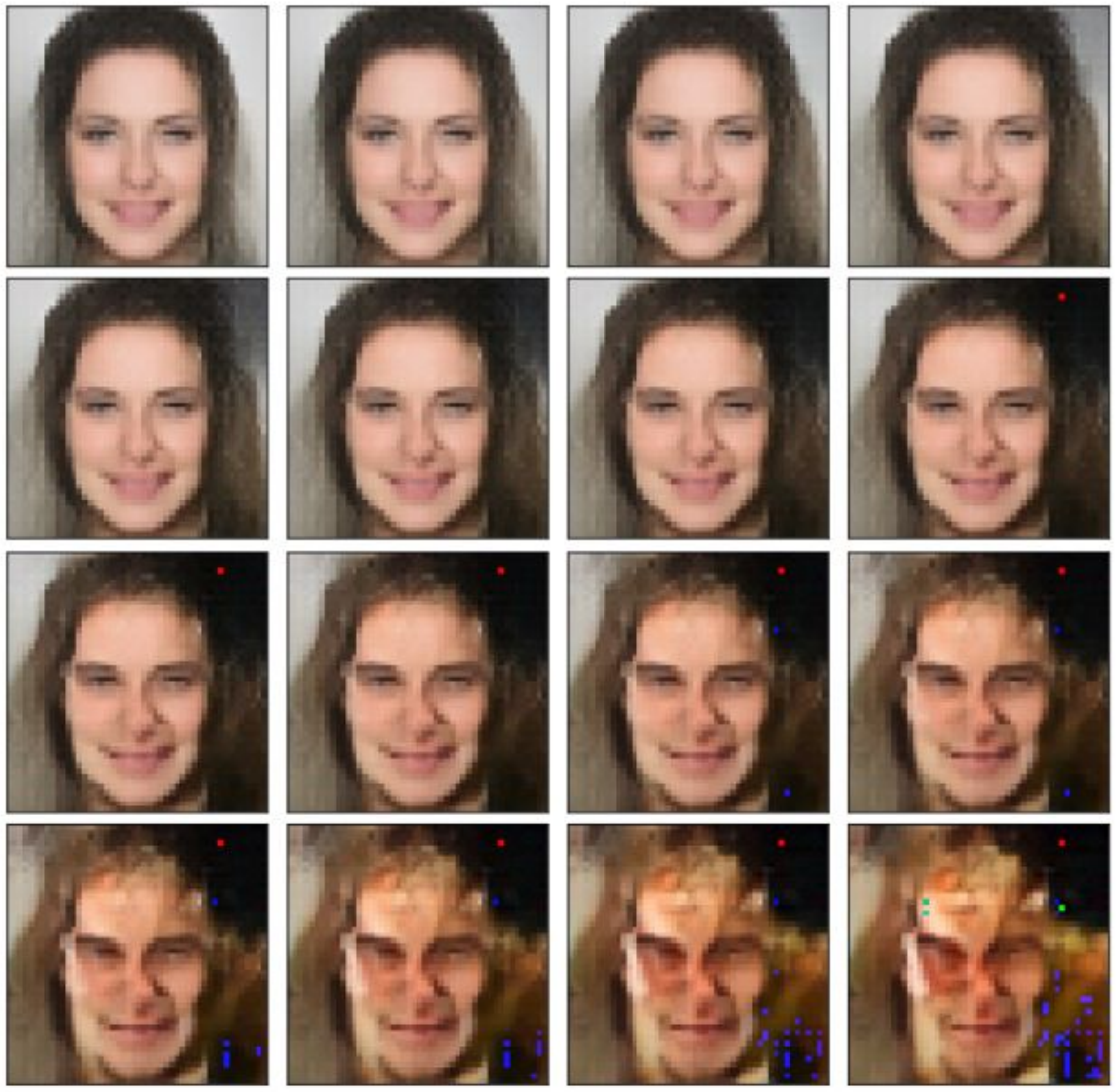
y - variable: visible some clusters

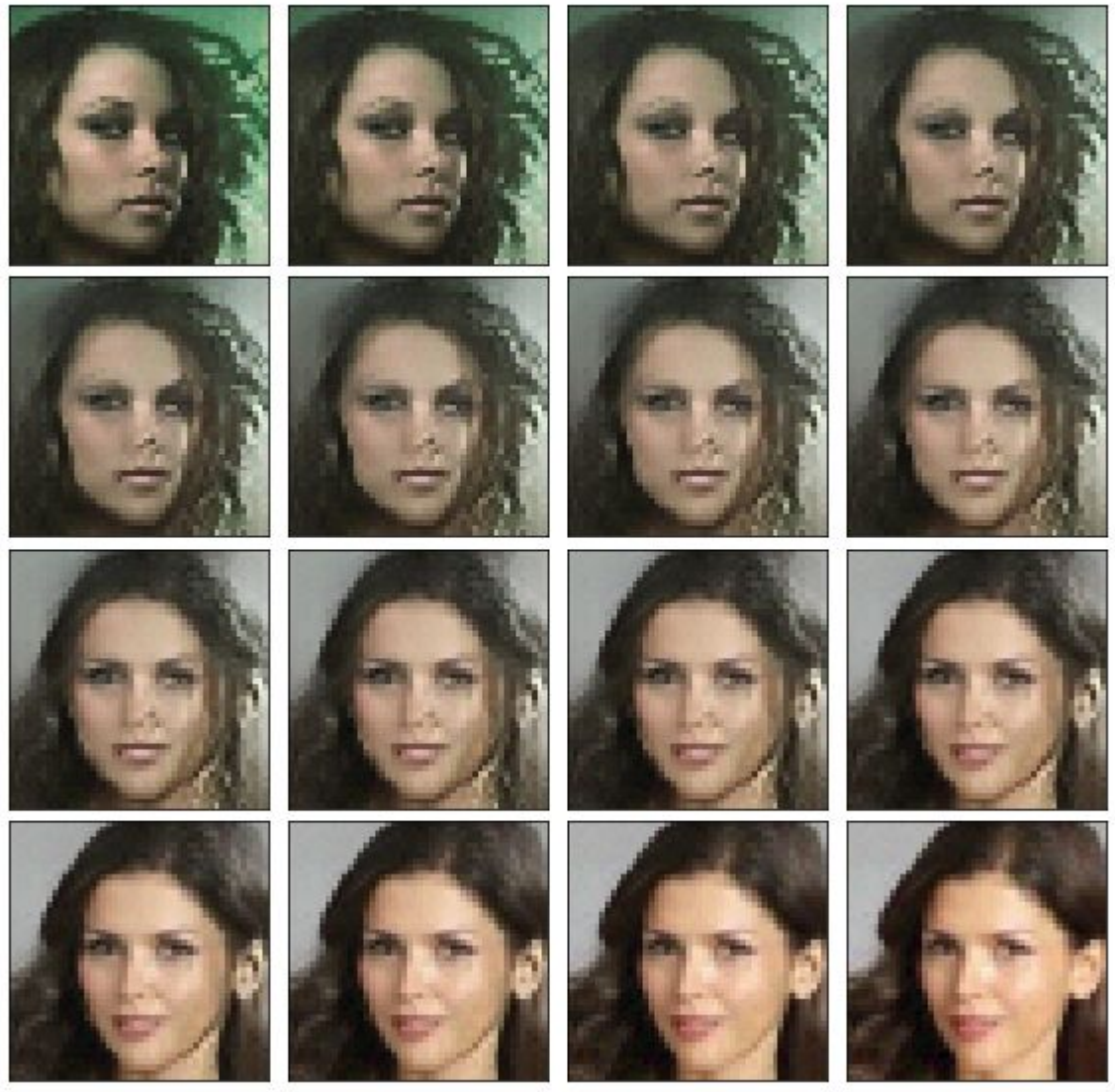


z - variable looks like noise

My experiments - selected results

- More results in notebooks: interpolations and effect of T





My experiments - selected results

- Things which are worth to check:
    - separate cost for p(z) and p(z) to force p(z) have smaller sigma. This will force network to compress more information in p(y) distribution.
    - Similarly add scale layer in the last layer in the same manner as it was done in NICE work, the network should learn which components are important.
    - effect of the binarization of the input image. I did not have time to check it.
    - Try different NNs in the affine coupling layer.
    - Try image augmentation
    - Try different datasets
    - Add classification cost

Questions?

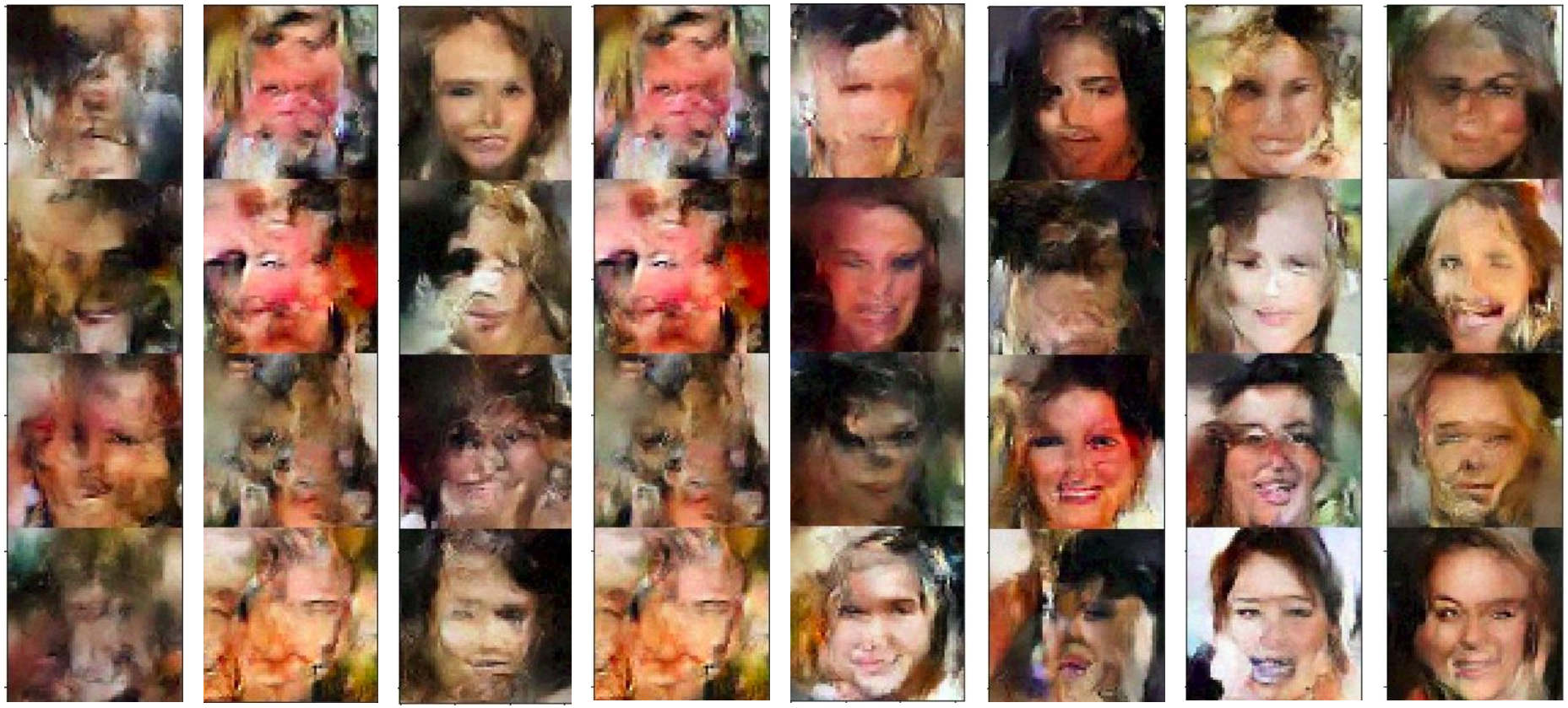
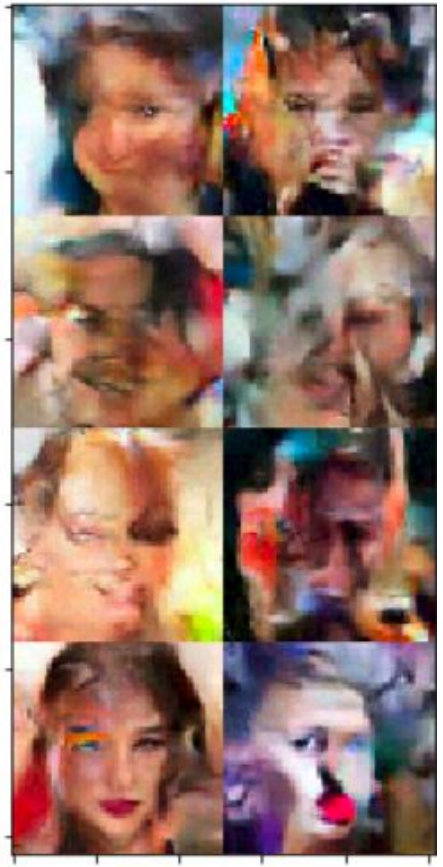


Fig.1 Cherry picked results

References:

- GLOW https://arxiv.org/pdf/1807.03039.pdf
- RealNVP https://arxiv.org/pdf/1605.08803.pdf
- NICE https://arxiv.org/pdf/1410.8516.pdf
- Flow++ https://openreview.net/forum?id=Hyg74h05tX



FORNAX

WWW.FORNAX.AI