Deep Learning

Week 8: Normalising flows

Contents

Introduction

So far in this module we have covered many of the fundamental building blocks of deep learning: from mathematical neurons and multilayer perceptrons, to optimisation and regularisation of deep learning models, and important architectures such as convolutional and recurrent neural networks.

In the remaining weeks of the module, we will use these building blocks to focus our attention on the probabilistic approach to deep learning. This is a branch of deep learning that aims to make use of tools from probability theory to account for noise and uncertainty in the data. Probabilistic deep learning models make direct use of probability distributions and latent random variables in the model architecture.

In this week of the course we will develop normalising flow deep learning models. Normalising flows are a class of generative models, that were first popularised in the context of variational inference by Rezende & Mohamed 2015, and in the context of density estimation by Dinh et al 2015. In this week, we will focus on using normalising flows to estimate continuous data distributions.

When trained as a density estimator, this type of model is able to produce new instances that could plausibly have come from the same dataset that it is trained on, as well as tell you whether a given example instance is likely. However, for complex datasets the data distribution can be very difficult to model, so this is a highly nontrivial task in general. This is where the power of deep learning models can be leveraged to learn highly multimodal and complicated data distributions, and this type of model has been successfully applied to domains such as image generation (Ho et al 2019), noise modelling (Abdelhamed et al 2019), audio synthesis (Prenger et al 2019), and video generation (Kumar et al 2019).

In this week, we'll be going through a popular class of normalising flow models, which are known as the NICE (Dinh 2015), RealNVP (Dinh et al 2017) and Glow (Kingma et al 2018) models.

Change of variables formula

The approach taken by normalising flows to solve the density estimation task is to take an initial, simple density, and transform it - possibly using a series of parameterised transformations - to produce a rich and complex distribution.

If these transformations are smooth and invertible, then we are able to evaluate the density of the complex transformed distribution. This property is important, because it then allows to train such a model using maximum likelihood. This is the idea behind normalising flows.

We'll start this week by reviewing the change of variables formula, which forms the mathematical basis of normalising flows.

Statement of the formula

Let $Z := (z_1, \dots, z_D) \in \mathbb{R}^D$ be a D-dimensional continuous random variable, and suppose that $f : \mathbb{R}^D \mapsto \mathbb{R}^D$ is a smooth, invertible transformation. Now consider the change of variables $X = f(Z)$, with $X = (x_1, \dots, x_D)$, and denote the probability density functions of the random variables Z and X by p_Z and p_X respectively.

The change of variables formula states that

$$p_X(\mathbf{x}) = p_Z(\mathbf{z}) \cdot \left|\det J_f(\mathbf{z})\right|^{-1}, \tag{1}$$

where $\mathbf{x}, \mathbf{z} \in \mathbb{R}^D$, and $J_f(\mathbf{z}) \in \mathbb{R}^{D \times D}$ is the **Jacobian** of the transformation f, given by the matrix of partial derivatives

$$J_f(\mathbf{z}) = \begin{bmatrix} \frac{\partial f_1}{\partial z_1}(\mathbf{z}) & \cdots & \frac{\partial f_1}{\partial z_D}(\mathbf{z}) \\ \vdots & \ddots & \vdots \\ \frac{\partial f_D}{\partial z_1}(\mathbf{z}) & \cdots & \frac{\partial f_D}{\partial z_d}(\mathbf{z}) \end{bmatrix},$$

and $\left|\det J_f(\mathbf{z})\right|$ is the absolute value of the determinant of the Jacobian matrix. Note that (1) can also be written in the log-form

$$\log p_X(\mathbf{x}) = \log p_Z(\mathbf{z}) - \log \left|\det J_f(\mathbf{z})\right|. \tag{2}$$

Furthermore, we can equivalently consider the transformation $Z = f^{-1}(X)$. Then the change of variables formulae can be written as

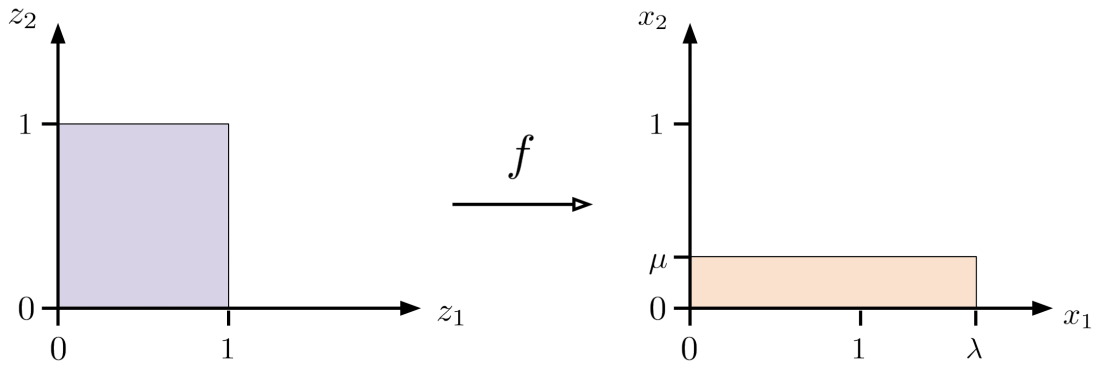
$$p_Z(\mathbf{z}) = p_X(\mathbf{x}) \cdot \left|\det J_{f^{-1}}(\mathbf{x})\right|^{-1}, \tag{3}$$
$$\log p_Z(\mathbf{z}) = \log p_X(\mathbf{x}) - \log \left|\det J_{f^{-1}}(\mathbf{x})\right|. \tag{4}$$

A simple example

We will demonstrate the change of variables formula with a simple example. Let $Z = (z_1, z_2)$ be a 2-dimensional random variable that is uniformly distributed on the unit square $[0, 1]^2 =: \Omega_Z$. We also define the transformation $f : \mathbb{R}^2 \mapsto \mathbb{R}^2$ as

$$f(z_1, z_2) = (\lambda z_1, \mu z_2) \tag{1}$$

for some nonzero $\lambda, \mu \in \mathbb{R}$. The random variable $X = (x_1, x_2)$ is given by $X = f(Z)$.



Linearly transformed uniformly distributed random variable

Since $\int_{\Omega_Z} p_Z(\mathbf{z}) d\mathbf{z} = 1$ and $\mathbf{z}$ is uniformly distributed, we have that

$$p_Z(\mathbf{z}) = 1 \quad \text{for} \quad \mathbf{z} \in \Omega_Z.$$

The random variable X is uniformly distributed on the region $\Omega_X = f(\Omega_Z)$ as shown in the figure above (for the case $\lambda, \mu > 0$). Since again $\int_{\Omega_X} p_X(\mathbf{x}) d\mathbf{x} = 1$, the probability density function for X must be given by

$$p_X(\mathbf{x}) = \frac{1}{|\Omega_X|} = \frac{1}{|\lambda\mu|} \quad \text{for} \quad \mathbf{x} \in \Omega_X.$$

This result corresponds to the equations $(1) - (4)$ above. In this simple example, the transformation f is linear, and the Jacobian matrix is given by

$$J_f(\mathbf{z}) = \begin{bmatrix} \lambda & 0 \\ 0 & \mu \end{bmatrix}. \tag{2}$$

The absolute value of the determinant is $\left|\det J_{f^{-1}}(\mathbf{x})\right| = |\lambda\mu| \neq 0$. Equation (1) then implies

$$\begin{aligned} p_X(\mathbf{x}) &= p_Z(\mathbf{z}) \cdot \left|\det J_f(\mathbf{z})\right|^{-1} && (3) \\ &= \frac{1}{|\lambda\mu|}. && (4) \end{aligned}$$

Writing in the log-form as in equation (2) gives

$$\begin{aligned} \log p_X(\mathbf{x}) &= \log p_Z(\mathbf{z}) - \log \left|\det J_f(\mathbf{z})\right| && (5) \\ &= \log(1) - \log|\lambda\mu| && (6) \\ &= -\log|\lambda\mu|. && (7) \end{aligned}$$

Sketch of proof in 1-D

We now provide a sketch of the proof of the change of variables formula in one dimension. Let Z and X be random variables such that $X = f(Z)$, where $f : \mathbb{R} \mapsto \mathbb{R}$ is a C^k diffeomorphism with $k \geq 1$. The change of variables formula in one dimension can be written

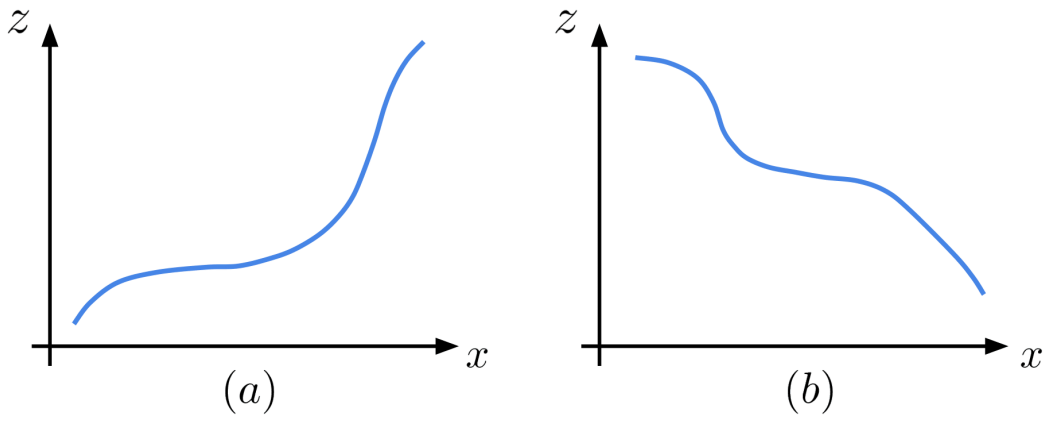
$$p_X(x) = p_Z(z) \cdot \left|\frac{d}{dz} f(z)\right|^{-1}, \qquad (\text{cf. equation } (1))$$

or equivalently as

$$p_X(x) = p_Z(z) \cdot \left|\frac{d}{dx} f^{-1}(x)\right|. \qquad (\text{cf. equation } (3))$$

Sketch of proof. For f to be invertible, it must be strictly monotonic. That means that for all $x^{(1)}, x^{(2)} \in \mathbb{R}$ with $x^{(1)} < x^{(2)}$, we have $f(x^{(1)}) < f(x^{(2)})$ (strictly monotonically increasing) or

$f(x^{(1)}) > f(x^{(2)})$ (strictly monotonically decreasing).



Sketch of monotonic functions: (a) strictly increasing, (b) strictly decreasing

Suppose first that f is strictly increasing. Also let F_X and F_Z be the cumulative distribution functions of the random variables X and Z respectively. Then we have

$$\begin{aligned} F_X(x) &= P(X \leq x) && (8)\\ &= P(f(Z) \leq x) && (9)\\ &= P(Z \leq f^{-1}(x)) \qquad (\text{since } f \text{ is monotonically increasing}) && (10)\\ &= F_Z(f^{-1}(x)) && (11) \end{aligned}$$

By differentiating on both sides with respect to x, we obtain the probability density function:

$$\begin{aligned} p_X(x) &= \frac{d}{dx} F_X(x) && (12)\\ &= \frac{d}{dx} F_Z(f^{-1}(x)) && (13)\\ &= \frac{d}{dz} F_Z(z) \cdot \frac{d}{dx} f^{-1}(x) && (14)\\ &= p_Z(z) \frac{d}{dx} f^{-1}(x) && (5) \end{aligned}$$

Now suppose first that f is strictly decreasing. Then

$$\begin{aligned} F_X(x) &= P(X \leq x) && (15)\\ &= P(f(Z) \leq x) && (16)\\ &= P(Z \geq f^{-1}(x)) \qquad (\text{since } f \text{ is monotonically decreasing}) && (17)\\ &= 1 - F_Z(f^{-1}(x)) && (18) \end{aligned}$$

Again differentiating on both sides with respect to x:

$$\begin{aligned} p_X(x) &= \frac{d}{dx} F_X(x) && (19)\\ &= -\frac{d}{dx} F_Z(f^{-1}(x)) && (20)\\ &= -F_Z'(f^{-1}(x)) \frac{d}{dx} f^{-1}(x) && (21)\\ &= -p_Z(z) \frac{d}{dx} f^{-1}(x) && (6) \end{aligned}$$

Now note that the inverse of a strictly monotonically increasing (resp. decreasing) function is again strictly monotonically increasing (resp. decreasing). This implies that the quantity $\frac{d}{dx} f^{-1}(x)$ is positive in (5) and negative in (6), and so these two equations can be combined into the single equation:

$$p_X(x) = p_Z(z) \left| \frac{d}{dx} f^{-1}(x) \right|$$

which completes the proof.

Application to normalising flows

Normalising flows are a class of models that exploit the change of variables formula to estimate an unknown target data density.

Suppose we have data samples $\mathcal{D} := \{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(n)}\}$, with each $\mathbf{x}^{(i)} \in \mathbb{R}^D$, and assume that these samples are generated i.i.d. from the underlying distribution p_X.

A normalising flow models the distribution p_X using a random variable Z (also of dimension D) with a simple distribution p_Z (e.g. an isotropic Gaussian), such that the random variable X can be written as a change of variables $X = f_\theta(Z)$, where θ is a parameter vector that parameterises the smooth invertible function f_θ.

The function f_θ is modelled using a neural network with parameters θ, which we want to learn from the data. An important point is that this neural network must be designed to be invertible, which is not the case in general with deep learning models. In practice, we often construct the neural network by composing multiple simpler blocks together.

We use the principle of maximum likelihood to learn the optimal parameters θ; that is:

$$\begin{aligned} \theta_{ML} &:= \arg\max_\theta P(\mathcal{D}; \theta) && (22) \\ &= \arg\max_\theta \log P(\mathcal{D}; \theta). && (23) \end{aligned}$$

In order to compute $\log P(\mathcal{D}; \theta)$ we can use the change of variables formula:

$$\begin{aligned} P(\mathcal{D}; \theta) &= \prod_{\mathbf{x} \in \mathcal{D}} p_Z(f_\theta^{-1}(\mathbf{x})) \cdot \left| \det J_{f_\theta^{-1}}(\mathbf{x}) \right| && (24) \\ \log P(\mathcal{D}; \theta) &= \sum_{x \in \mathcal{D}} \log p_Z(f_\theta^{-1}(\mathbf{x})) + \log \left| \det J_{f_\theta^{-1}}(\mathbf{x}) \right| && (7) \end{aligned}$$

The term $p_Z(f_\theta^{-1}(\mathbf{x}))$ can be computed for a given data point $\mathbf{x} \in \mathcal{D}$ since the neural network f_θ is designed to be invertible, and the distribution p_Z is known. The term $\det J_{f_\theta^{-1}}(\mathbf{x})$ is also computable, although this also highlights another important aspect of normalising flow models: they should be designed such that the determinant of the Jacobian can be efficiently computed.

The log-likelihood (7) is usually optimised as usual in minibatches, with gradient-based optimisation methods.

NICE / RealNVP

NICE

NICE stands for "nonlinear independent components estimation", and is a deep learning architecture framework for density estimation tasks. A key motivation for the proposed framework given in the abstract of the original paper (Dinh 2015) is as follows:

> 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.

As with many normalising flow examples, a typical choice for a base distribution would be an isotropic Gaussian, which is then transformed by the deep learning model. An important aspect is the efficient calculation of the Jacobian determinant of the transformation.

In this section, we will describe the NICE architecture, and the RealNVP architecture that is built upon it. We will follow the exposition of the original papers, and think of the forward transformation as acting on the data input example. Note however that this is in contrast to the usual bijector convention of using the forward transformation for sampling, and the inverse transformation for computing log probs.

Affine coupling layer

The basic building block of the NICE architecture is the affine coupling layer. Given an input $\mathbf{x} \in \mathbb{R}^D$, we split it into two blocks $(\mathbf{x}_{1:d}, \mathbf{x}_{d+1:D})$, where $d < D$ (usually $d \approx D/2$), and apply a transformation of the form

$$\mathbf{z}_{1:d} = \mathbf{x}_{1:d}, \tag{1}$$
$$\mathbf{z}_{d+1:D} = \mathbf{x}_{d+1:D} + t(\mathbf{x}_{1:d}), \tag{2}$$

where $t : \mathbb{R}^d \mapsto \mathbb{R}^{D-d}$ is an arbitrarily complex function, such as a neural network. It is easy to see that the coupling layer as above has an identity Jacobian matrix, and is trivially invertible:

$$\mathbf{x}_{1:d} = \mathbf{z}_{1:d}, \tag{25}$$
$$\mathbf{x}_{d+1:D} = \mathbf{z}_{d+1:D} - t(\mathbf{z}_{1:d}). \tag{26}$$

Several coupling layers can be composed together to obtain a more complex, layered transformation. Note that a coupling layer leaves part of its input unchanged, and so the roles of the two subsets should be interchanged in alternating layers.

If we examine the Jacobian, we can see that at least three coupling layers are needed to allow all dimensions to influence each other (this is left as an exercise for the reader). In the NICE paper, networks were composed with four coupling layers.

RealNVP

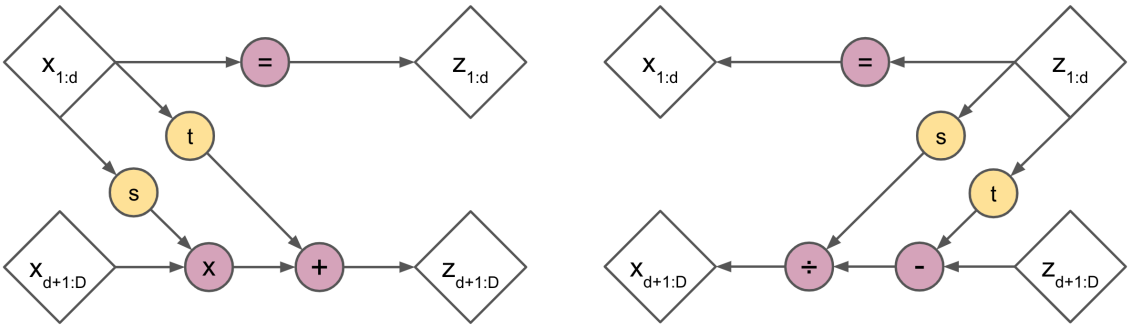
RealNVP stands for real-valued, non-volume preserving (Dinh et al 2017). It was a follow-up work to the NICE paper, in which the affine coupling layer was modified as follows:

$$\mathbf{z}_{1:d} = \mathbf{x}_{1:d}, \tag{3}$$
$$\mathbf{z}_{d+1:D} = \mathbf{x}_{d+1:D} \odot \exp(s(\mathbf{x}_{1:d})) + t(\mathbf{x}_{1:d}), \tag{4}$$

where s and t stand for scale and translation, and are both functions that map from $\mathbb{R}^d$ to $\mathbb{R}^{D-d}$. The name RealNVP emphasises the fact that the transformation $(3) - (4)$ is no longer volume-preserving, as is the case with $(1) - (2)$, due to the additional scaling provided by the term $\exp(s(\mathbf{x}_{1:d}))$. We use the network output $s(\mathbf{x}_{1:d})$ as a log-scale parameter for numerical stability.

As before, the inverse transformation is no more complex than the forward propagation:

$$\mathbf{x}_{1:d} = \mathbf{z}_{1:d}, \tag{5}$$
$$\mathbf{x}_{d+1:D} = (\mathbf{z}_{d+1:D} - t(\mathbf{z}_{1:d})) \odot \exp(-s(\mathbf{z}_{1:d})). \tag{6}$$



(a) forward pass (a) inverse pass

The forward and inverse passes of the RealNVP affine coupling layer

Now, the Jacobian is given by

$$\frac{\partial \mathbf{z}}{\partial \mathbf{x}} = \begin{bmatrix} \mathbb{I}_d & \mathbf{0} \\ \frac{\partial \mathbf{z}_{d+1:D}}{\partial \mathbf{x}_{1:d}} & \operatorname{diag}\left(\exp(s(\mathbf{x}_{1:d}))\right) \end{bmatrix} \in \mathbb{R}^{D \times D}$$

and the log of the absolute value of the Jacobian determinant is easily calculated as $\sum_j s(\mathbf{x}_{1:d})_j$.

Spatial and channel-wise masking

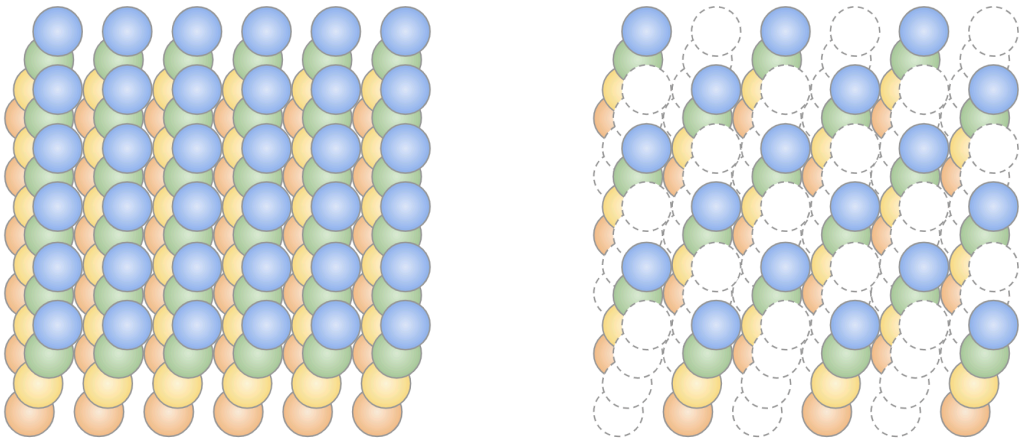
Observe that the partitioning $\mathbf{x} \to (\mathbf{x}_{1:d}, \mathbf{x}_{d+1:D})$ can be implemented using a binary mask $b \in \{0,1\}^{n_h \times n_w \times c}$, so that the forward pass $(3) - (4)$ can be written

$$\mathbf{z} = b \odot \mathbf{x} + (1 - b) \odot (\mathbf{x} \odot \exp(s(b \odot \mathbf{x})) + t(b \odot \mathbf{x})). \tag{7}$$

Similarly, the inverse pass $(5) - (6)$ can be written

$$\mathbf{x} = b \odot \mathbf{z} + (1 - b) \odot ((\mathbf{z} - t(b \odot \mathbf{z})) \odot \exp(-s(b \odot \mathbf{z}))). \tag{8}$$

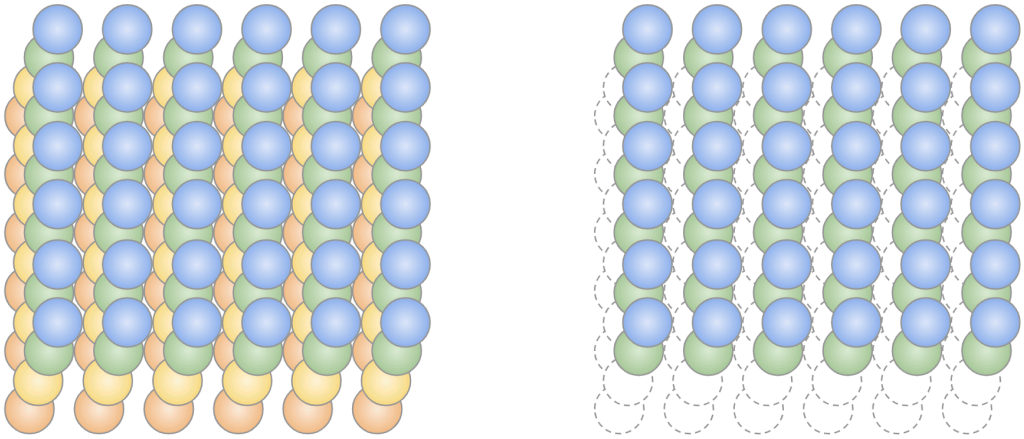
RealNVP implements two types of masking for image data $\mathbf{x} \in \mathbb{R}^{n_h \times n_w \times c}$: spatial checkerboard and channel-wise masking. A spatial checkerboard mask applies the same partitioning to every channel dimension, as illustrated in the following figure.



(a) (b)

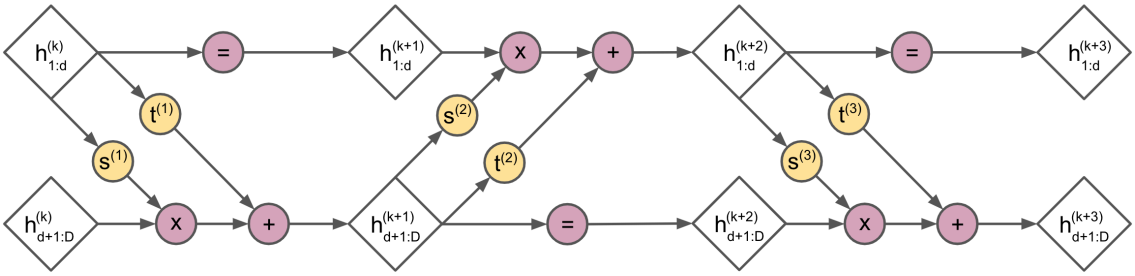
Spatial checkerboard masking in RealNVP. (a) A layer input $\mathbf{h} \in \mathbb{R}^{6 \times 6 \times 4}$ without masking, and (b) multiplied elementwise by a spatial checkerboard mask $b_s \in \{0,1\}^{6 \times 6}$, which is broadcast along the channel dimension

A channel mask instead operates along the channel dimension, and applies the same partitioning at every spatial location, as in the following figure.



Channel masking in RealNVP. (a) A layer input $\mathbf{h} \in \mathbb{R}^{6\times6\times4}$ without masking, and (b) multiplied elementwise by a channel mask $b_c \in \{0, 1\}^4$, which is broadcast across the spatial dimensions

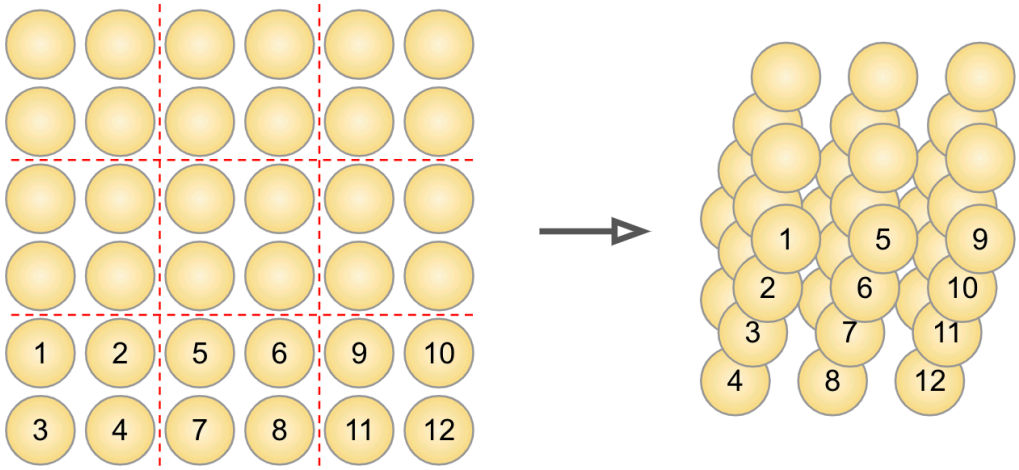
As in the NICE framework, we want to ensure that all dimensions are able to interact with each other. The RealNVP architecture consists of three layers of alternating checkerboard masks, where the partitions are permuted.



Three affine coupling layers, with alternating masks in between layers

Squeeze operation

In the RealNVP architecture, after the three affine coupling layers with checkerboard masking there is a squeeze operation, where the spatial dimensions of the layer are divided into $2 \times 2 \times c$ subsquares, and reshaped into $1 \times 1 \times 4c$. The figure below illustrates this operation for a single channel:



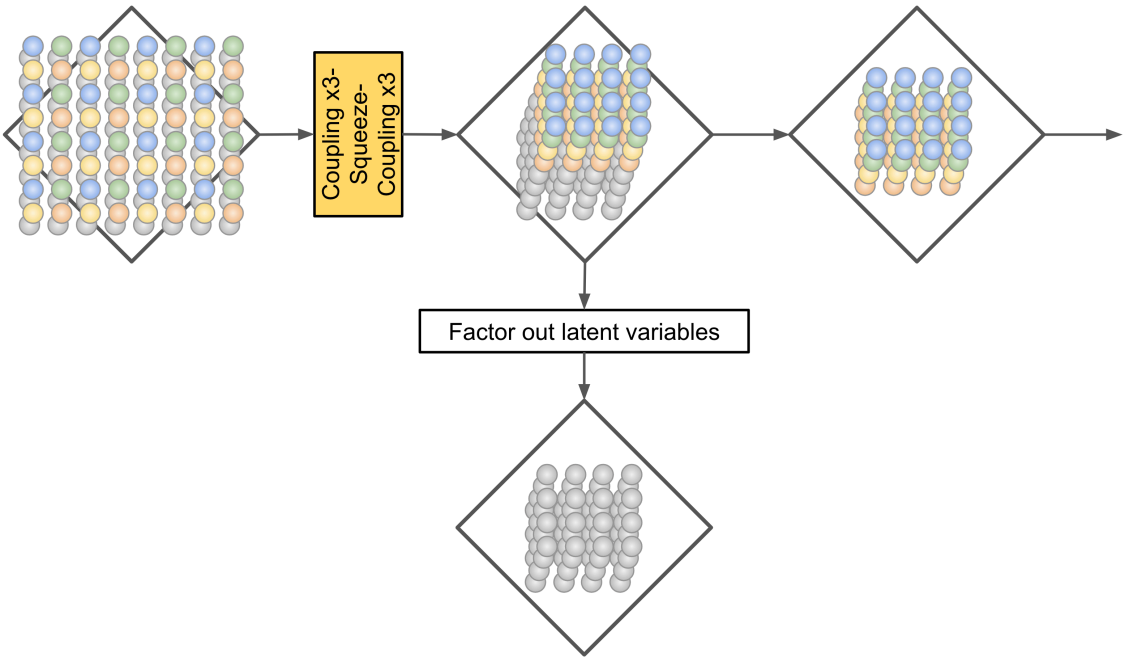
The squeeze operation. The spatial dimensions are halved, and the channel dimension is quadrupled

Following the squeeze operation, there are three more affine coupling layers, this time using channel masking, and again permuting the partitions between each layer.

Multiscale architecture

The final component of the RealNVP framework is the multiscale architecture. With the squeeze operation, the spatial dimensions are downsampled, but the channel dimensions are increased. In order to reduce the overall layer sizes in the deeper layers, dimensions are factored out as latent variables at regular intervals.

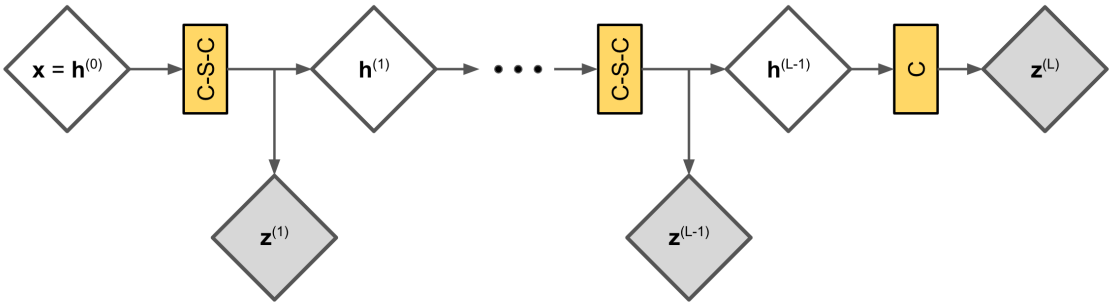
After one of the blocks of coupling-squeeze-coupling described above, half of the dimensions are factored out as latent variables, while the other half is further processed through subsequent layers.



Example showing how latent variables are factored out in the multiscale architecture. A layer input $\mathbf{h}^{(k)} \in \mathbb{R}^{8\times8\times2}$ will be reshaped to a $4 \times 4 \times 8$-shaped tensor after the coupling-squeeze-coupling block. Half of this tensor is absorbed into the base distribution as a latent variable $\mathbf{z}^{(k+1)} \in \mathbb{R}^{4\times4\times4}$ and the remainder $\mathbf{h}^{(k+1)} \in \mathbb{R}^{4\times4\times4}$ is processed through further layers of the network

The complete RealNVP model has multiple levels of the multiscale architecture. This results in latent variables that represent different scales of features in the model. After a number of these levels, the

final scale does not use the squeezing operation, and instead applies four affine coupling layers with alternating checkerboard masks to produce the final latent variable.



The end-to-end RealNVP architecture. Each scale consists of a block of 3 coupling layers (with checkerboard mask), squeeze, 3 coupling layers (with channel mask), followed by half of the dimensions factored out as a latent variable. The final scale consists only of 4 coupling layers (with checkerboard mask) to produce the final latent variable

The following summarises the forward pass $\mathbf{z} = f(\mathbf{x})$ of the overall architecture with L scales. The functions $f^{(1)}, \ldots, f^{(L-1)}$ consist of the coupling-squeeze-coupling block, whereas the function $f^{(L)}$ consists of 4 coupling layers with checkerboard masks.

> $$\begin{align}\mathbf{h}^{(0)}&=\mathbf{x}\ (\mathbf{z}^{(k+1)}, \mathbf{h}^{(k+1)})&=f^{(k+1)}(\mathbf{h}^{(k)}),\qquad k=0,\ldots, L-2\ \mathbf{z}^{(L)}&= f^{(L)}(\mathbf{h}^{(L-1)})\

\mathbf{z} &= (\mathbf{z}^{(1)},\ldots,\mathbf{z}^{(L)})\end{align}$$

The latent variables factored out at each scale are reshaped and concatenated to produce a single latent variable $\mathbf{z} = (\mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(L)})$, which is assumed to be distributed according to a known base distribution (e.g. a diagonal Gaussian).

As a final note, the architecture described in this section was further developed with the Glow model (Kingma and Dhariwal 2018), where the checkerboard and channel-wise masking was replaced with 1x1 convolutions.

Affine Coupling Layer

In this section we will build a partial implementation of the RealNVP architecture. In particular, we will write a custom layer to implement the affine coupling layer, using a binary mask:

$$\mathbf{z} = b \odot \mathbf{x} + (1 - b) \odot (\mathbf{x} \odot \exp(s(b \odot \mathbf{x})) + t(b \odot \mathbf{x})) \quad \text{(forward pass)} \quad (27)$$
$$\mathbf{x} = b \odot \mathbf{z} + (1 - b) \odot ((\mathbf{z} - t(b \odot \mathbf{z})) \odot \exp(-s(b \odot \mathbf{z}))) \quad \text{(inverse pass)} \quad (28)$$

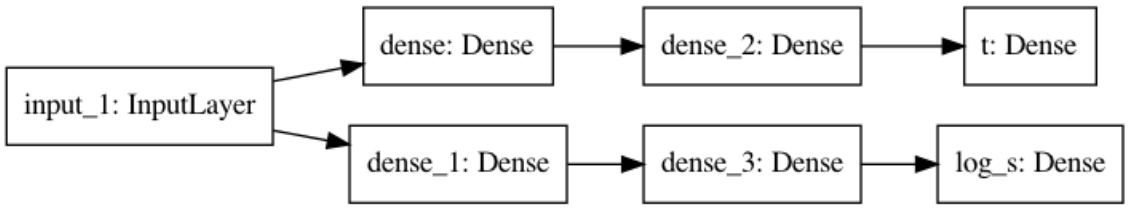
In [2]:
```
import os
os.environ['KERAS_BACKEND'] = 'torch'
import keras
from keras import ops
```

Our affine coupling layer will take `shift_and_log_scale_fn` and `mask` arguments, which represent the s and t functions and b mask respectively. In addition, we will implement an `inverse` method for the inverse pass (the `call` method implements the forward pass).

The forward pass will be used during training (the forward pass as above maps $\mathbf{x} \mapsto \mathbf{z}$), and so our custom layer also makes use of the in-built `add_loss` method (see this guide) that will add the layer's contribution to the negative log likelihood loss. Recall that the Jacobian of a composition of functions is the product of Jacobians, so the layer's contribution is the negative log Jacobian determinant of this layer transformation.

In [3]:
```
# Create the AffineCouplingLayer class

from keras.layers import Layer

class AffineCouplingLayer(Layer):

    def __init__(self, shift_and_log_scale_fn, mask, **kwargs):
        super().__init__(**kwargs)
        self.shift_and_log_scale_fn = shift_and_log_scale_fn
        self.b = ops.cast(mask, 'float32')

    def build(self, input_shape):
        self.event_dims = list(range(1, len(input_shape)))

    def call(self, x):
        t, log_s = self.shift_and_log_scale_fn(self.b * x)
        z = self.b * x + (1 - self.b) * (x * ops.exp(log_s) + t)

        self.add_loss(-ops.mean(ops.sum(log_s * (1 - self.b),
                                        axis=self.event_dims)))
        return z

    def inverse(self, z):
        t, log_s = self.shift_and_log_scale_fn(self.b * z)
        x = self.b * z + (1 - self.b) * ((z - t) * ops.exp(-log_s))
        return x
```

We will build a `shift_and_log_scale_fn` with the following structure:



In [4]:
```
# Create an example shift_and_log_scale_fn

from keras.models import Model
from keras.layers import Input, Dense

def get_shift_and_log_scale_fn(input_size, hidden_units=[32, 32],
                               activation='relu'):
    inputs = Input(shape=[input_size])
    h1 = inputs
    h2 = inputs
    for units in hidden_units:
        h1 = Dense(units, activation=activation)(h1)
        h2 = Dense(units, activation=activation)(h2)
    shift = Dense(input_size)(h1)
    log_scale = Dense(input_size, activation='tanh')(h2)
    return Model(inputs=inputs, outputs=[shift, log_scale])

shift_and_log_scale = get_shift_and_log_scale_fn(input_size=8)
```

In [5]:
```
# Define a binary mask
```

```
mask = ops.array([1, 1, 1, 1, 0, 0, 0, 0])
```

In [6]:
```
# Create the AffineCouplingLayer

aff = AffineCouplingLayer(shift_and_log_scale, mask)
```

In [7]:
```
# Test the AffineCouplingLayer

x = keras.random.normal((2, 8))
print(x)
z = aff(x)
print(z)
print(aff.inverse(z))
```

```
tensor([[-0.7209, -1.6071,  0.5782, -0.9089, -0.5766,  1.0481,  0.2713, -1.2844],
        [ 1.0921, -0.1416,  1.7041, -0.3765,  0.4813, -0.6875, -0.2419, -0.9440]],
       device='cuda:0')
tensor([[-0.7209, -1.6071,  0.5782, -0.9089, -0.6982,  1.0753,  1.1346, -1.1020],
        [ 1.0921, -0.1416,  1.7041, -0.3765,  0.4525, -1.3017, -0.0128, -0.5098]],
       device='cuda:0', grad_fn=<AddBackward0>)
tensor([[-0.7209, -1.6071,  0.5782, -0.9089, -0.5766,  1.0481,  0.2713, -1.2844],
        [ 1.0921, -0.1416,  1.7041, -0.3765,  0.4813, -0.6875, -0.2419, -0.9440]],
       device='cuda:0', grad_fn=<AddBackward0>)
```

In [8]:
```
# Inspect the loss contribution

aff.losses
```

Out[8]:
```
[tensor(-0.7647, device='cuda:0', grad_fn=<NegBackward0>)]
```

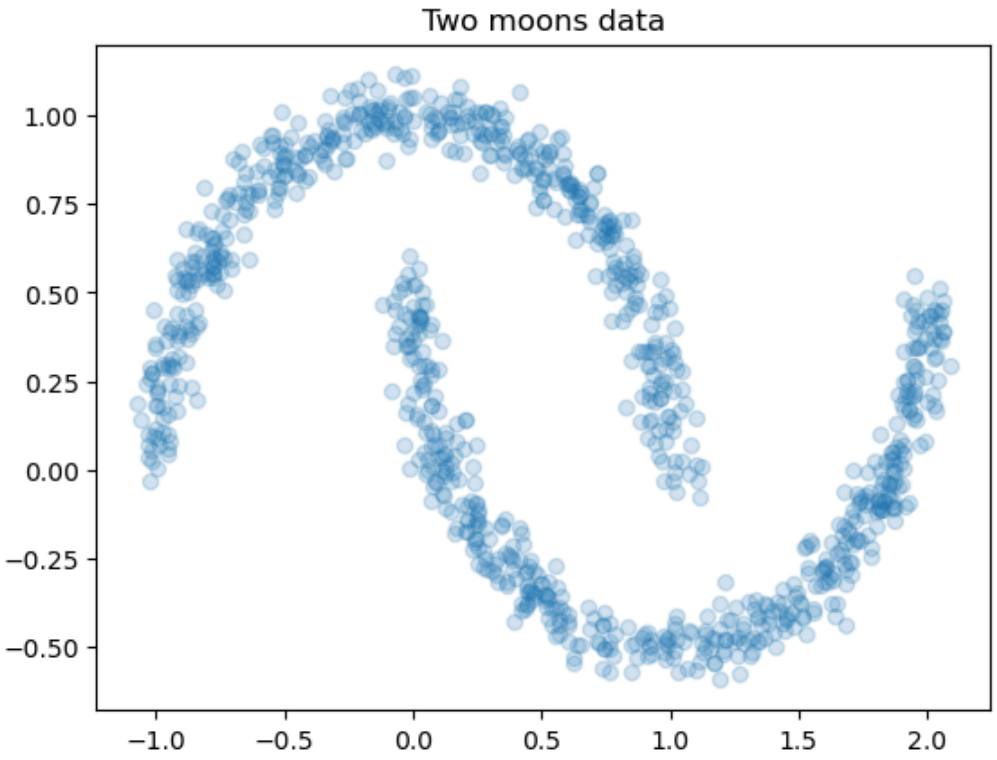
Exercise. Verify the log Jacobian determinant value that is returned above by computing it directly using the `shift_and_log_scale` Model and the input Tensor `x` above.

Two moons dataset

We will now create a normalising flow using the `AffineCouplingLayer` and train it on a two moons dataset.

In [9]:
```
# Create the dataset

from sklearn.datasets import make_moons
import numpy as np

train_data = make_moons(n_samples=50000, noise=0.05)[0].astype(np.float32)
val_data = make_moons(n_samples=1000, noise=0.05)[0].astype(np.float32)
```

In [10]:
```
# Visualise the dataset

import matplotlib.pyplot as plt

plt.scatter(train_data[:1000, 0], train_data[:1000, 1], alpha=0.2)
plt.title("Two moons data")
plt.show()
```



In [11]:
```
# Create training and validation DataLoaders

import torch

train_ds = torch.utils.data.TensorDataset(torch.from_numpy(train_data),
                                          torch.from_numpy(train_data))
val_ds = torch.utils.data.TensorDataset(torch.from_numpy(val_data),
                                        torch.from_numpy(val_data))

train_dl = torch.utils.data.DataLoader(train_ds, batch_size=128,
                                       shuffle=True,drop_last=True)
val_dl = torch.utils.data.DataLoader(val_ds, batch_size=128,
                                     shuffle=False, drop_last=True)
```

Define and train the normalising flow

Our normalising flow model will consist of a number of affine coupling layers, with the mask reversed between each layer. The model maps the data $\mathbf{x}$ to the latent variable $\mathbf{z}$.

In [12]:
```
# Define the coupling layer chain

from keras.models import Sequential

model = Sequential([
    Input(shape=[2]),
])
num_layers = 8
mask = ops.array([1, 0])

for l in range(num_layers):
    shift_and_log_scale = get_shift_and_log_scale_fn(2, hidden_units=[256, 256],
                                                     activation='relu')
    aff = AffineCouplingLayer(shift_and_log_scale, mask)
    mask = 1 - mask
    model.add(aff)

model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
affine_coupling_layer_1 (AffineCouplingLayer)	(None, 2)	134,148
affine_coupling_layer_2 (AffineCouplingLayer)	(None, 2)	134,148
affine_coupling_layer_3 (AffineCouplingLayer)	(None, 2)	134,148
affine_coupling_layer_4 (AffineCouplingLayer)	(None, 2)	134,148
affine_coupling_layer_5 (AffineCouplingLayer)	(None, 2)	134,148
affine_coupling_layer_6 (AffineCouplingLayer)	(None, 2)	134,148
affine_coupling_layer_7 (AffineCouplingLayer)	(None, 2)	134,148
affine_coupling_layer_8 (AffineCouplingLayer)	(None, 2)	134,148

Total params: 1,073,184 (4.09 MB)
Trainable params: 1,073,184 (4.09 MB)
Non-trainable params: 0 (0.00 B)

Recall the negative log-likelihood of the data under our model f_θ is given by

$$-\log P(\mathcal{D};\theta) = \sum_{x\in\mathcal{D}} -\log p_Z(f_\theta(\mathbf{x})) - \log\left|\det J_{f_\theta}(\mathbf{x})\right|, \tag{29}$$

where in our example f_θ is mapping from X to Z.

Each affine coupling layer adds its (negative) log Jacobian determinant contribution to the negative log-likelihood loss. We will write a custom loss function to take care of the final $-\log p_Z(f_\theta(\mathbf{x}))$ term (see here for more information on creating custom loss functions).

In [13]:
```
# Create a custom loss function

def normal_log_pdf_loss(y_true, y_pred):
    event_dims = list(range(1, len(y_pred.shape)))
    const = 0.5 * ops.log(2. * np.pi)
    return ops.sum(const + ops.square(y_pred)/2., axis=event_dims)
```

In [14]:
```
# Define the inverse mapping Z -> X

from keras.models import Model

z_sample = Input(shape=[2])
h = z_sample
for layer in reversed(model.layers):
    h = layer.inverse(h)
inverse_model = Model(inputs=z_sample, outputs=h)
```

In [15]:
```
import os
from keras.callbacks import Callback

class SaveSamples(Callback):
```

```
    def __init__(self, plot_folder='./plots', **kwargs):
        super(SaveSamples, self).__init__(**kwargs)
        self.plot_folder = plot_folder

    def on_epoch_begin(self, epoch, logs=None):
        self.epoch = epoch

    def plot(self, num):
        plt.figure(figsize=(8, 5))
        ax = plt.gca()
        plt.xlim([-1.5, 2.5])
        plt.ylim([-1, 1.5])
        ax.set_aspect('equal')
        samples = ops.convert_to_numpy(inverse_model(keras.random.normal([2000, 2])))
        plt.scatter(samples[:, 0], samples[:, 1], alpha=0.3)
        plt.tight_layout()
        plt.savefig("./plots/{:05d}.png".format(num))
        plt.close()

    def on_train_begin(self, logs=None):
        if not os.path.exists(self.plot_folder):
            os.makedirs(self.plot_folder)
        self.iteration = 0
        self.plot(self.iteration + 1)

    def on_train_batch_end(self, batch, logs=None):
        self.iteration += 1
        if self.iteration % 30 == 0:
            self.plot((self.iteration // 30) + 1)

save_samples = SaveSamples()
```

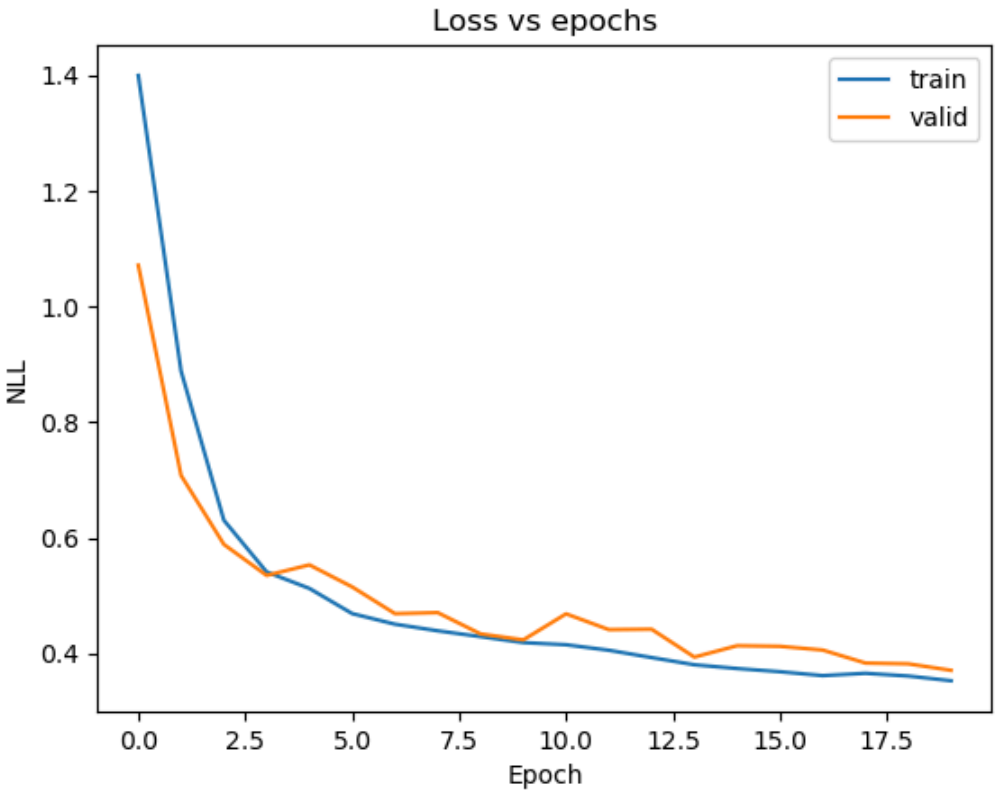
We can pass our custom loss function directly into the `compile` method as usual.

In [16]:
```
# Compile and train the model

model.compile(loss=normal_log_pdf_loss,
              optimizer=keras.optimizers.Adam(learning_rate=1e-4))
history = model.fit(train_dl, validation_data=val_dl,
                    epochs=20, callbacks=[save_samples])
```

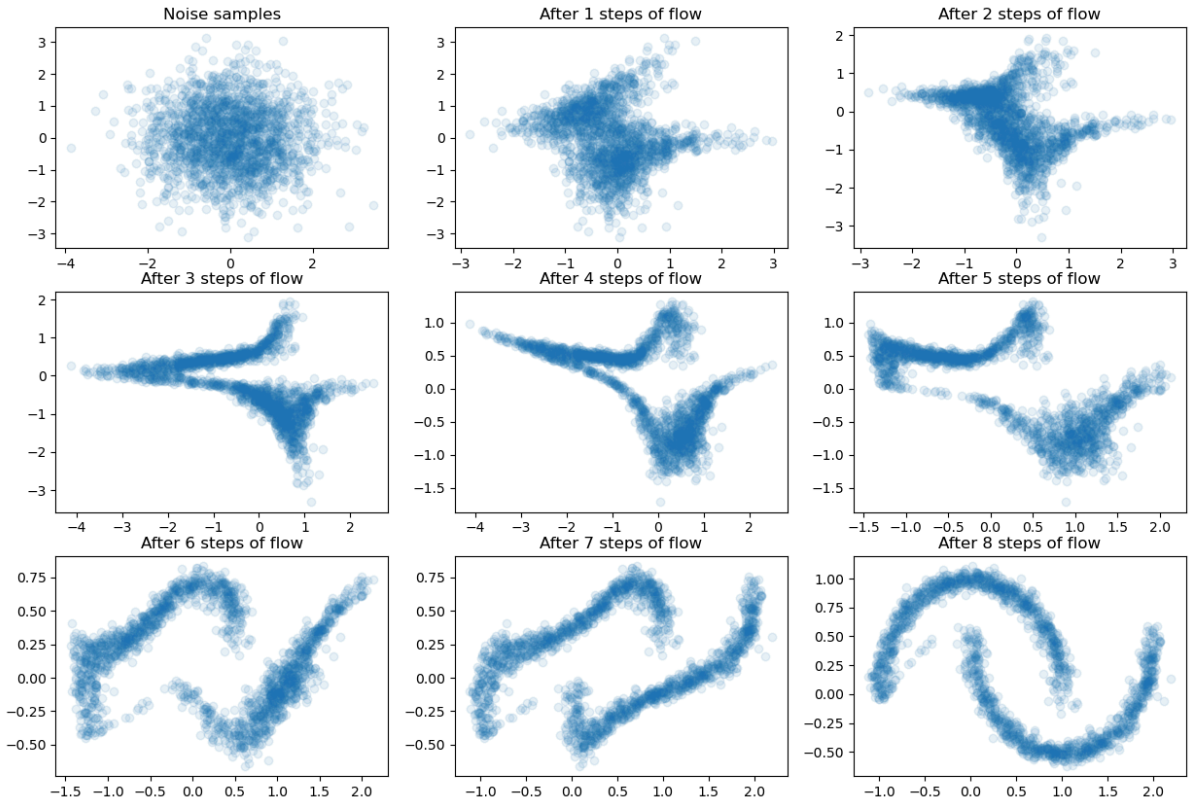
```
Epoch 1/20
390/390 ━━━━━━━━━━━━━━━━━━━━ 13s 32ms/step - loss: 1.6518 - val_loss: 1.0713
Epoch 2/20
390/390 ━━━━━━━━━━━━━━━━━━━━ 13s 32ms/step - loss: 0.9873 - val_loss: 0.7077
Epoch 3/20
390/390 ━━━━━━━━━━━━━━━━━━━━ 13s 32ms/step - loss: 0.6700 - val_loss: 0.5885
Epoch 4/20
390/390 ━━━━━━━━━━━━━━━━━━━━ 13s 32ms/step - loss: 0.5608 - val_loss: 0.5348
Epoch 5/20
390/390 ━━━━━━━━━━━━━━━━━━━━ 13s 32ms/step - loss: 0.5150 - val_loss: 0.5530
Epoch 6/20
390/390 ━━━━━━━━━━━━━━━━━━━━ 12s 32ms/step - loss: 0.4801 - val_loss: 0.5149
Epoch 7/20
390/390 ━━━━━━━━━━━━━━━━━━━━ 13s 33ms/step - loss: 0.4510 - val_loss: 0.4687
Epoch 8/20
390/390 ━━━━━━━━━━━━━━━━━━━━ 12s 32ms/step - loss: 0.4394 - val_loss: 0.4704
Epoch 9/20
390/390 ━━━━━━━━━━━━━━━━━━━━ 13s 32ms/step - loss: 0.4350 - val_loss: 0.4336
Epoch 10/20
390/390 ━━━━━━━━━━━━━━━━━━━━ 13s 32ms/step - loss: 0.4178 - val_loss: 0.4232
Epoch 11/20
390/390 ━━━━━━━━━━━━━━━━━━━━ 13s 32ms/step - loss: 0.4202 - val_loss: 0.4683
Epoch 12/20
390/390 ━━━━━━━━━━━━━━━━━━━━ 13s 33ms/step - loss: 0.4067 - val_loss: 0.4410
Epoch 13/20
390/390 ━━━━━━━━━━━━━━━━━━━━ 13s 32ms/step - loss: 0.3962 - val_loss: 0.4417
Epoch 14/20
390/390 ━━━━━━━━━━━━━━━━━━━━ 13s 33ms/step - loss: 0.3791 - val_loss: 0.3935
Epoch 15/20
390/390 ━━━━━━━━━━━━━━━━━━━━ 15s 38ms/step - loss: 0.3737 - val_loss: 0.4132
Epoch 16/20
390/390 ━━━━━━━━━━━━━━━━━━━━ 13s 33ms/step - loss: 0.3770 - val_loss: 0.4121
Epoch 17/20
390/390 ━━━━━━━━━━━━━━━━━━━━ 13s 34ms/step - loss: 0.3561 - val_loss: 0.4057
Epoch 18/20
390/390 ━━━━━━━━━━━━━━━━━━━━ 14s 36ms/step - loss: 0.3642 - val_loss: 0.3831
Epoch 19/20
390/390 ━━━━━━━━━━━━━━━━━━━━ 13s 33ms/step - loss: 0.3676 - val_loss: 0.3818
Epoch 20/20
390/390 ━━━━━━━━━━━━━━━━━━━━ 12s 32ms/step - loss: 0.3555 - val_loss: 0.3706
```

In [17]:

```python
# Plot the learning curves

plt.plot(history.history['loss'], label='train')
plt.plot(history.history['val_loss'], label='valid')
plt.title("Loss vs epochs")
plt.ylabel("NLL")
plt.xlabel("Epoch")
plt.legend()
plt.show()
```



In [19]:
```
# Plot the model transformations

noise = keras.random.normal([2000, 2])

fig, axs = plt.subplots(3, 3, figsize=(15,10))
h = noise
axs[0, 0].scatter(ops.convert_to_numpy(h)[:, 0],
                  ops.convert_to_numpy(h)[:, 1], alpha=0.1)
axs[0, 0].set_title("Noise samples")

for i, (layer, ax) in enumerate(zip(reversed(model.layers), axs.flat[1:])):
    h = layer.inverse(h)
    ax.scatter(ops.convert_to_numpy(h)[:, 0],
               ops.convert_to_numpy(h)[:, 1], alpha=0.1)
    ax.set_title(f"After {i+1} steps of flow")

plt.show()
```



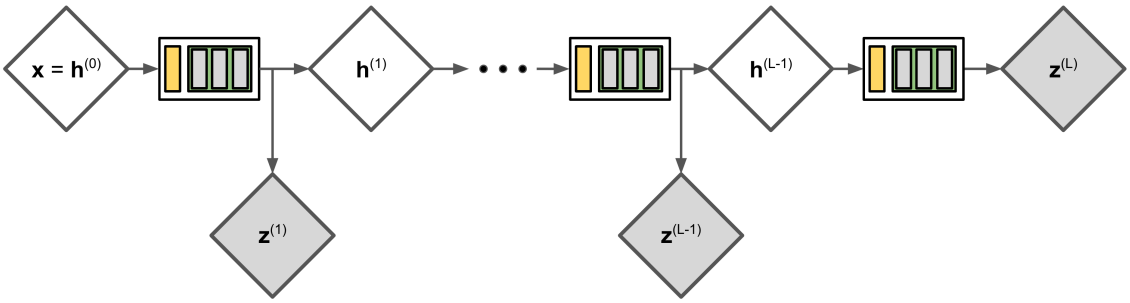
Exercise. Try re-running the two moons example again, but using a bi-modal base distribution. Is the flow able to more easily approximate the two moons distributions?

Glow

Glow (Kingma et al 2018) is short for Generative Flow, and this model builds directly from the RealNVP model. It uses the same multiscale architecture setup, where the latent variables are factored out at different levels with the same mechanism as before.

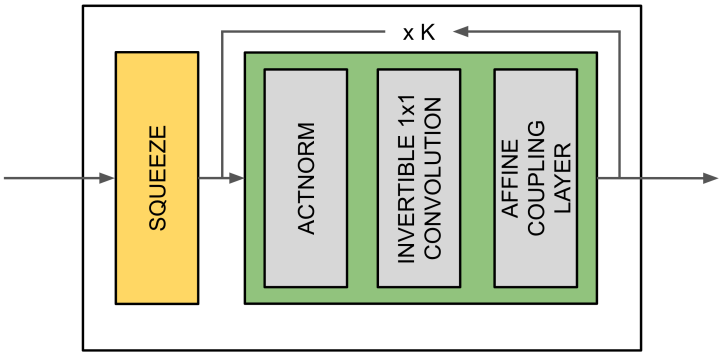
However, Glow replaces the coupling-squeeze-coupling blocks with a different computation block. The Glow model aims to both simplify the RealNVP architecture, as well as to generalise a part of it.

As we'll see, Glow still retains the affine coupling layer as the core computation of the model, and also still uses the squeeze operation, and the channel-wise masking. But it removes the checkerboard mask entirely, and also uses a different transformation instead of the alternating mask mechanism of RealNVP.



Glow uses the same multiscale architecture as in RealNVP, but replaces the coupling-squeeze-coupling block

The computation block used in the Glow model is shown in the diagram below.



The computation block used in the Glow model: a squeeze layer, followed by K steps of flow, each consisting of an actnorm layer, an invertible 1x1 convolution, and an affine coupling layer

The squeeze and affine couple layers are the same as in the RealNVP model. The affine coupling layer uses a fixed channel-wise mask. The input to the computation block is first passed through the squeeze layer, which halves each of the spatial dimensions of the input, and multiplies the size of the channel dimension by four.

This is followed by a number of steps of flow, indicated in the diagram above by a green block. A single step of flow consists of an actnorm layer, followed by an invertible 1x1 convolution layer - both of which are new to the Glow model - followed by an affine coupling layer.

Within each level of the Glow model, the squeeze operation is followed by K steps of flow, before half of the neurons are factored out as latent variables as part of the multi scale architecture that we've seen before.

Activation normalisation (actnorm)

Actnorm is actually a replacement for the batch normalisation layers that are used within the RealNVP model. Recall that batchnorm is used in RealNVP in the shift and scale networks s and t that are used to parameterise the transformation inside the affine coupling layers. It's also applied to the output of the whole affine coupling layer, and this is where we need to compute the log-Jacobian determinant of the transformation, shown in the following table.

Transformation	log-Jacobian determinant
$\hat{h}_j^{(k)} = \frac{h_j^{(k)} - \mu_{jm}^{(k)}}{\sqrt{\left(\sigma_{jm}^{(k)}\right)^2 + \epsilon}}$	
$z_j^{(k)} = \gamma_j^{(k)} \hat{h}_j^{(k)} + \beta_j^{(k)}$	$-\frac{1}{2} \sum_j \log\left(\left(\sigma_{jm}^{(k)}\right)^2 + \epsilon\right)$ | $\sum_j \log\left\lvert\gamma_j^{(k)}\right\rvert$

Batch normalisation transformations and log-Jacobian determinants

The potential problem with batch normalisation is that it relies on minibatch statistics being reasonable estimates of the dataset statistics. If the minibatch statistics are high variance, then this can impact performance. In the case of the original Glow implementation, practical computing constraints meant that the the authors needed to reduce the minibatch size to one, and batchnorm then became less effective.

The solution in the Glow architecture is to introduce actnorm, or activation normalisation, as an alternative to batchnorm. The transformation of actnorm is a simple affine transformation per feature, parameterised by scale and shift parameters gamma and beta:

$$z_j^{(k)} = \gamma_j^{(k)} h_j^{(k)} + \beta_j^{(k)},$$

where the subscript j is indexing the feature dimension and k indexes the layer.

Actnorm does not compute minibatch statistics, which means that the transformation is more stable, but also less flexible, because it's not aware of the statistics of the given input minibatch, and so it's not able to normalise the input activations towards a target mean and variance.

However, the parameters $\gamma_j^{(k)}$ and $\beta_j^{(k)}$ are initialised based on the statistics of a sample minibatch. In particular, they're initialised such that the output activations of the layer have zero mean and unit variance on the given sample minibatch. This is an example of data-dependent initialisation, and is trying to initialise these parameters in a good place for the particular dataset, even though the post-activations will drift away from zero mean and unit variance during training.

Clearly this transformation is trivially invertible so long as all the gammas are nonzero, and in practice these are parameterised to ensure this.

The log-Jacobian determinant is given by

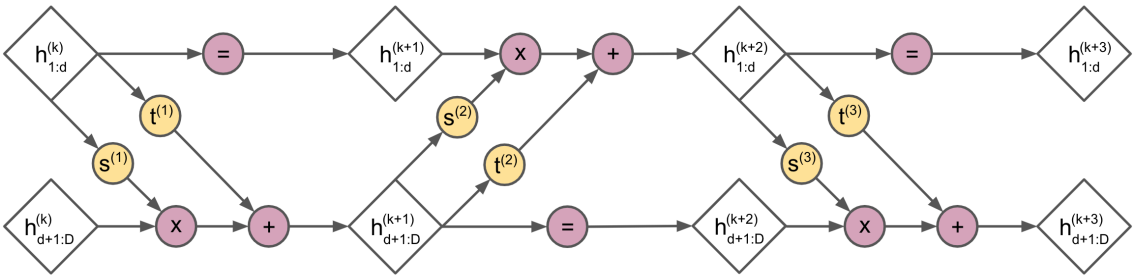
$$\sum_j \log\left|\gamma_j^{(k)}\right|,$$

which is a straightforward and efficient computation.

Note that the expressions above assume rank-one inputs, so that in particular $\mathbf{h}^{(k)}$ would be a vector. When using the Glow model on image inputs, the inputs $\mathbf{h}^{(k)}$ will be 3-dimensional tensors with shape (h, w, c), and the parameters $\gamma_j^{(k)} \in \mathbb{R}^c$ and $\beta_j^{(k)} \in \mathbb{R}^c$ are shared across every spatial location. In this case, the log-Jacobian determinant is multiplied by hw.

Invertible 1x1 convolution

Recall the following affine coupling layer sequence in RealNVP, that consists of alternating binary channel-wise masks.



Three affine coupling layers, with alternating channel masks in between layers

The forward transformation in one of the affine coupling layers can be written as

$$\mathbf{z} = b \odot \mathbf{x} + (1 - b) \odot (\mathbf{x} \odot \exp(s(b \odot \mathbf{x})) + t(b \odot \mathbf{x})),$$

where b is the channel mask, that zeros out half of the neurons in the channel dimension. RealNVP composes three affine coupling layers together like this, reversing the binary mask each time.

Instead, we could equivalently think of using a fixed binary mask, and permuting the neurons in the layer in the channel dimension after each affine coupling layer. The permutation matrix $W \in \mathbb{R}^{c \times c}$ would have the following block structure.

$$W = \begin{bmatrix} \mathbf{0} & \mathbf{I} \\ \mathbf{I} & \mathbf{0} \end{bmatrix}$$

This same permutation matrix would then be applied at each spatial location. This is a special case of a convolutional layer, where the convolutional kernel has spatial dimensions of 1 by 1, where there is no bias term in the convolutional layer, and where the number of output channels is the same as the number of input channels.

So an equivalent formulation to the sequence of affine coupling layers with an alternating binary channel mask that we looked at before in RealNVP, is where we instead use a 1x1 convolution operation.

The main contribution of the Glow model is to generalise the permutation of the channel dimensions by using a learned 1x1 convolution. This convolutional layer will have the same number of input and output channels, and doesn't use a bias term.

So then we can write the transformation as a 2-dimensional convolution as follows:

$$\mathbf{h}^{(k+1)} = \mathrm{conv2d}(\mathbf{h}^{(k)}; \mathbf{W}^{(k)}),$$

where $\mathbf{W}^{(k)}$ is the convolutional kernel. The log Jacobian determinant will then be

$$h \cdot w \cdot \log \left| \det(\mathbf{W}^{(k)}) \right|,$$

where h and w are the sizes of the height and width dimensions respectively.

The problem here is that it can be expensive to compute the determinant of the convolutional kernel $\mathbf{W}^{(k)}$. This computation scales as $\mathcal{O}(c^3)$, where c is the number of channels.

A solution to this problem is to use the LU decomposition of the $c \times c$ matrix $\mathbf{W}^{(k)}$:

$$\mathbf{W}^{(k)} = \mathbf{P}^{(k)} \mathbf{L}^{(k)} \left(\mathbf{U}^{(k)} + \mathrm{diag}\left(\mathbf{s}^{(k)} \right) \right).$$

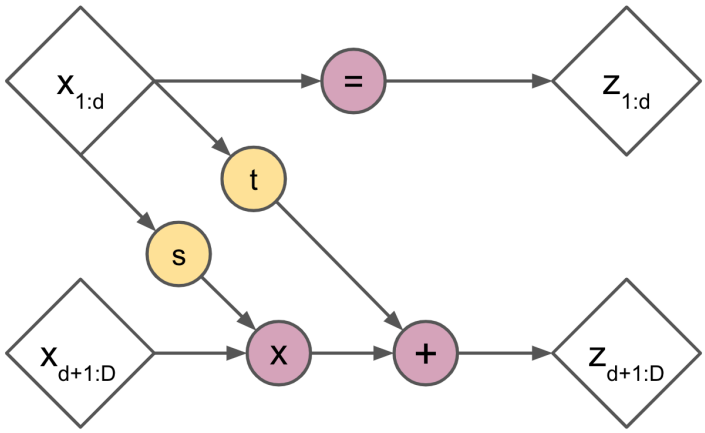
In the above, $\mathbf{P}^{(k)}$ is some permutation matrix, $\mathbf{L}^{(k)}$ is lower triangular with ones on the diagonal, $\mathbf{U}^{(k)}$ is upper triangular, and we have separated out the diagonal elements so that $\mathbf{U}^{(k)}$ is upper triangular with zeros on the diagonal, and the diagonal elements are written as the diagonalisation of a vector $\mathbf{s}^{(k)}$.

In practical implementations, we can directly parameterise the convolutional kernel as in the above expression. The log Jacobian determinant can now be written as h times w times the sum of the log of the absolute values of the elements of the vector $\mathbf{s}^{(k)}$. This computation now scales as $\mathcal{O}(c)$, so is much more efficient.

The Glow paper initialises the convolutional kernel $\mathbf{W}^{(k)}$ as a random rotation matrix, and computes the LU decomposition in order to initialise the parameterisation you see above. This will set the permutation matrix $\mathbf{P}^{(k)}$, which remains fixed throughout the training. The matrices $\mathbf{L}^{(k)}$ and $\mathbf{U}^{(k)}$ as well as the vector $\mathbf{s}^{(k)}$ are all learned during training.

Affine Coupling Layer

The third layer in a single step of flow is the affine coupling layer with a (fixed) binary channel-wise mask b. The forward transformation and the log-Jacobian determinant are shown again below.



Transformation	log-Jacobian determinant
$\mathbf{z} = b \odot \mathbf{x} + (1 - b) \odot (\mathbf{x} \odot \exp(s(b \odot \mathbf{x})) + t(b \odot \mathbf{x}))$	$\sum_j \log\lvert s_j\rvert$

The actnorm, invertible 1x1 convolution and affine coupling layers are the three layers that make up a single step of flow of the Glow model. Within each level of the Glow architecture, the inputs are first passed through the squeeze layer, and through K steps of flow, before half of the neurons are factored out as latent variables. The remaining neurons continue on for further processing. This is the same multi scale architecture we saw in the RealNVP model.

References

- Abdelhamed, A., Brubaker, M. A. & Brown, M. S. (2019), "Noise flow: Noise modeling with conditional normalizing flows", in *Proceedings of the IEEE International Conference on Computer Vision*, 3165–3173.
- Dinh, L., Krueger, D. & Bengio, Y. (2015),"NICE: Non-linear Independent Components Estimation", in *3rd International Conference on Learning Representations, (ICLR)*, San Diego, CA, USA, May 7-9, 2015.
- Dinh, L., Sohl-Dickstein, J. & Bengio, S. (2017), "Density estimation using Real NVP", in *5th International Conference on Learning Representations, (ICLR)*, Toulon, France, April 24-26, 2017.
- Ho, J., Chen, X., Srinivas, A., Duan, Y., & Abbeel, P. (2019), "Flow++: Improving flow-based generative models with variational dequantization and architecture design", in *Proceedings of the 36th International Conference on Machine Learning, ICML*.
- Kingma, D. P. & Dhariwal, P. (2018), "Glow: Generative Flow with Invertible 1x1 Convolutions", in *Advances in Neural Information Processing Systems*, **31**, 10215--10224.
- Kumar, M., Babaeizadeh, M., Erhan, D., Finn, C., Levine, S., Dinh, L. & Kingma, D. (2019), "VideoFlow: A Flow-Based Generative Model for Video", in *Workshop on Invertible Neural Nets and Normalizing Flows*, ICML, 2019.
- Prenger, R., Valle, R., & Catanzaro, B. (2019), "Waveglow: A flow-based generative network for speech synthesis", in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, IEEE, 3617-3621.
- Rezende, D. & Mohamed, S. (2015), "Variational Inference with Normalizing Flows", in *Proceedings of Machine Learning Research*, **37**, 1530-1538.