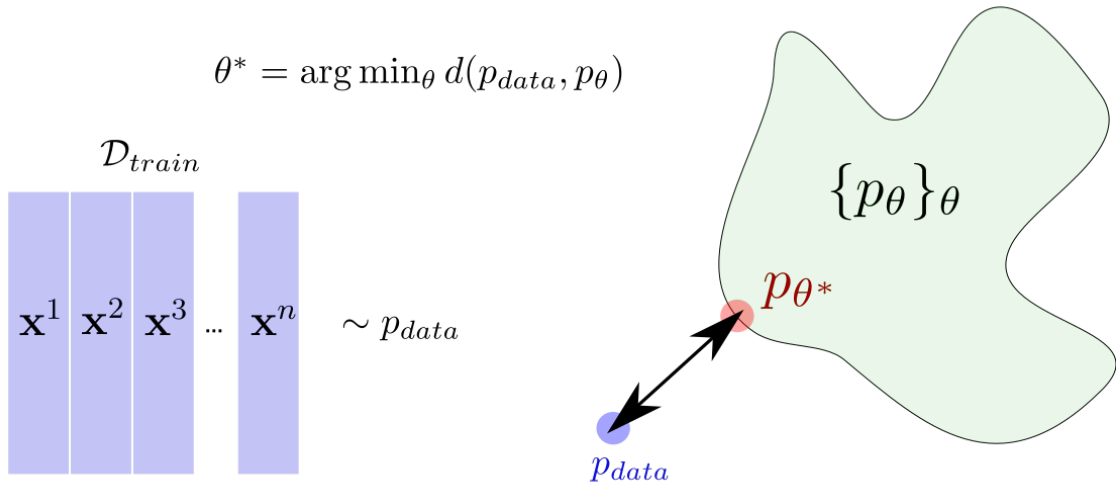
Normalizing flows

November 8, 2022



Desirable properties of any model distribution p_θ:

- Should be easy to **evaluate**, ideally with a closed form.
- Should be easy to **sample**.

Many simple distributions satisfy the above properties e.g.,

- Gaussian,
- uniform distributions.

But real data distributions are in general more complex (multi-modal).

Basic idea of **flow models**: Map **simple** distributions (easy to sample and evaluate densities) to more complex distributions with an **invertible transformation** from the latent space to the data space.

A **flow model** shares some characteristics with VAEs. Remember:

1. **Simple prior** in **the latent space**:

$$\mathbf{z} \sim \mathcal{N}(0, I) = p(\mathbf{z}).$$

2. Applies transforms

$$p(\mathbf{x}|\mathbf{z}) = \mathcal{N}(\mu_\theta(\mathbf{z}), \Sigma_\theta(\mathbf{z})).$$

$p(\mathbf{z})$ is very simple, the marginal on $\mathbf{x}$ p_θ may be quite complex.

But it is difficult to evaluate:

$$p_\theta(\mathbf{x}) = \int_{\mathbf{z}} p_\theta(\mathbf{x}|\mathbf{z})p(\mathbf{z})d\mathbf{z}$$

requires to span all the $\mathbf{z}$ that could result in high values of $p(\mathbf{x}|\mathbf{z})$, and that is why we use an auxiliary distribution $q(\mathbf{z}|\mathbf{x};\phi)$.

Another path: we would like to easily "invert" $p(\mathbf{x}|\mathbf{z})$ into $p(\mathbf{z}|\mathbf{x})$.

Idea: Use a **deterministic, invertible function of z**

$$\mathbf{x} = \mathbf{f}_\theta(\mathbf{z}),$$

with parameters θ.

1 Mapping random variables

Suppose we are in 1D. If you have access to $p_{\mathbf{z}}(\mathbf{z})$ and we suppose

$$\mathbf{x} = \mathbf{f}(\mathbf{z}),$$

with $\mathbf{f}$ monotone and invertible on some interval for $\mathbf{x}$. What is $p_{\mathbf{x}}(\mathbf{x})$ on that interval?

$$p_{\mathbf{x}}(\mathbf{x}) = p_{\mathbf{z}}(\mathbf{f}^{-1}(\mathbf{x}))|(\mathbf{f}^{-1'})(\mathbf{x})|.$$

For example, consider $\mathbf{z} \sim \mathcal{U}[0, \pi/2[$ and $\mathbf{f}(\mathbf{z}) = \tan \mathbf{z}$, then for $\mathbf{x} \in [0, +\infty[$

$$\mathbf{f}^{-1}(\mathbf{x}) = \arctan(\mathbf{x}) \text{ and } \mathbf{f}^{-1'}(\mathbf{x}) = \frac{1}{1+x^2},$$

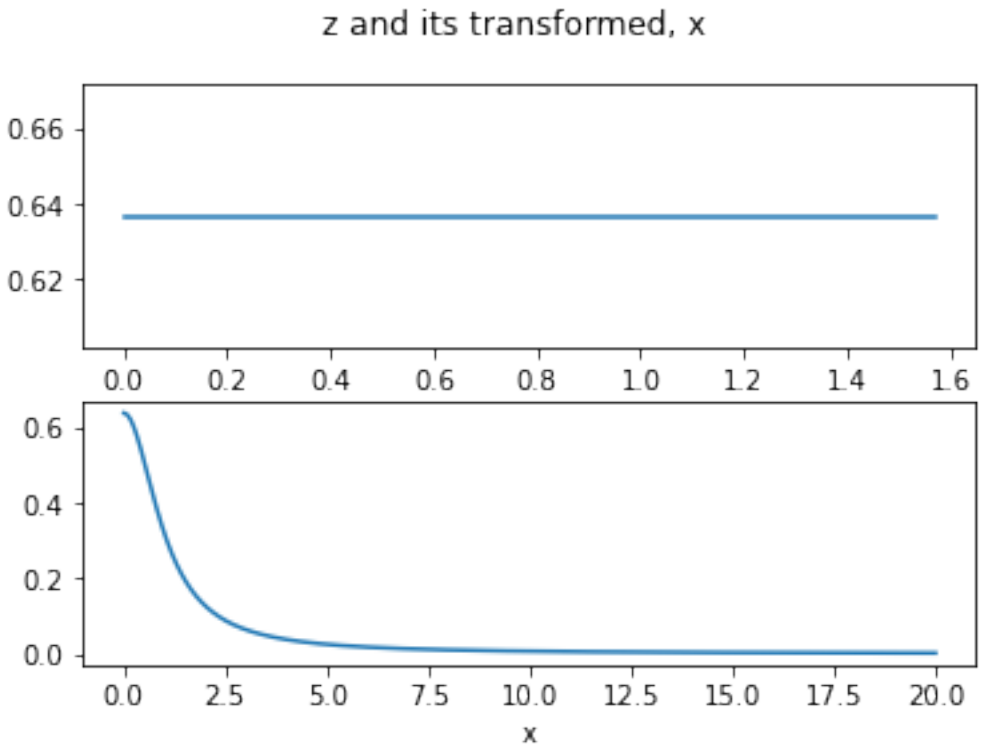
hence

$$\begin{aligned} p_{\mathbf{x}}(\mathbf{x}) &= p_{\mathbf{z}}(\mathbf{f}^{-1}(\mathbf{x}))|(\mathbf{f}^{-1'})(\mathbf{x})| && (1)\\ &= \frac{2}{\pi(1+\mathbf{x}^2)}. && (2)\\ & && (3) \end{aligned}$$

[4]:
```
import matplotlib.pyplot as plt
import numpy as np
import math

# Latent variable
z     = np.linspace(0.0, math.pi/2.0)
```

```
pdf_z = (2.0/math.pi)*np.ones(z.shape[0])

# Image x of z
x     = np.linspace(0.0, 20.0, num=1000)
pdf_x = (2.0/math.pi)/(1+x**2)

fig, (ax1, ax2) = plt.subplots(2, 1)
fig.suptitle('z and its transformed, x')

ax1.plot(z, pdf_z, '-')
ax1.set_xlabel('z')

ax2.plot(x, pdf_x, '-')
ax2.set_xlabel('x')
```

[4]: `Text(0.5, 0, 'x')`



The shape of the distribution on the data (**x**) may be **quite more complex** than the one of the prior over the latent (**z**)!

2 Normalizing flows models

Variational Inference with Normalizing Flows, Danilo Jimenez Rezende and Shakir Mohamed. Proceedings of the International Conference on International Conference on Machine Learning (ICML'15), 2015.

A **normalizing flow** is a transformation of a probability density through a **sequence of invertible mappings**.

It amounts to applying several, consecutive changes of variables from the initial distribution (typically, the latent variable), which makes the initial density 'flow' through the sequence of mappings.

At each step, we have a **normalized probability distribution**.

In a normalizing flow model, the mapping between $\mathbf{z}$ and $\mathbf{x}$, given by

$$f_\theta : \mathbb{R}^m \to \mathbb{R}^m,$$

is **deterministic and invertible** and such that $\mathbf{x} = f_\theta(\mathbf{z})$ and $\mathbf{z} = f_\theta^{-1}(\mathbf{x})$.

Compare to other latent variable models:

$$p(\mathbf{x}) = \int_{\mathbf{z}} p(\mathbf{x}|\mathbf{z})p(\mathbf{z})d\mathbf{z}.$$

Now there will be a one-to-one relation between $\mathbf{z}$ and $\mathbf{x}$ instead of a stochastic relation.

Also note some important change:

- $\mathbf{z}$ is now **by design** with the **same dimension** as $\mathbf{x}$.
- instead of using lower dimensionality in more traditional latent variable models.

Assume that we **choose a simple distribution of the latent variable**:

$$p(\mathbf{z}).$$

Using the rule of change of variables, the marginal likelihood $p(\mathbf{x})$ is given by

$$p_{\mathbf{x}}(\mathbf{x};\theta) = p_{\mathbf{z}}(\mathbf{f}_\theta^{-1}(\mathbf{x}))|\det \frac{\partial \mathbf{f}_\theta^{-1}}{\partial \mathbf{x}}|.$$

$\frac{\partial \mathbf{f}_\theta^{-1}}{\partial \mathbf{x}}$ is the **Jacobian matrix of the reciprocate function**.

This is a generalization of the 1D result we saw before.

But you can also formulate it **without the reciprocal function** being explicit:

$$p_{\mathbf{x}}(\mathbf{x};\theta) = p_{\mathbf{z}}(\mathbf{f}_\theta^{-1}(\mathbf{x})) \left|\det \frac{\partial \mathbf{f}_\theta}{\partial \mathbf{z}}\right|^{-1}, \tag{4}$$

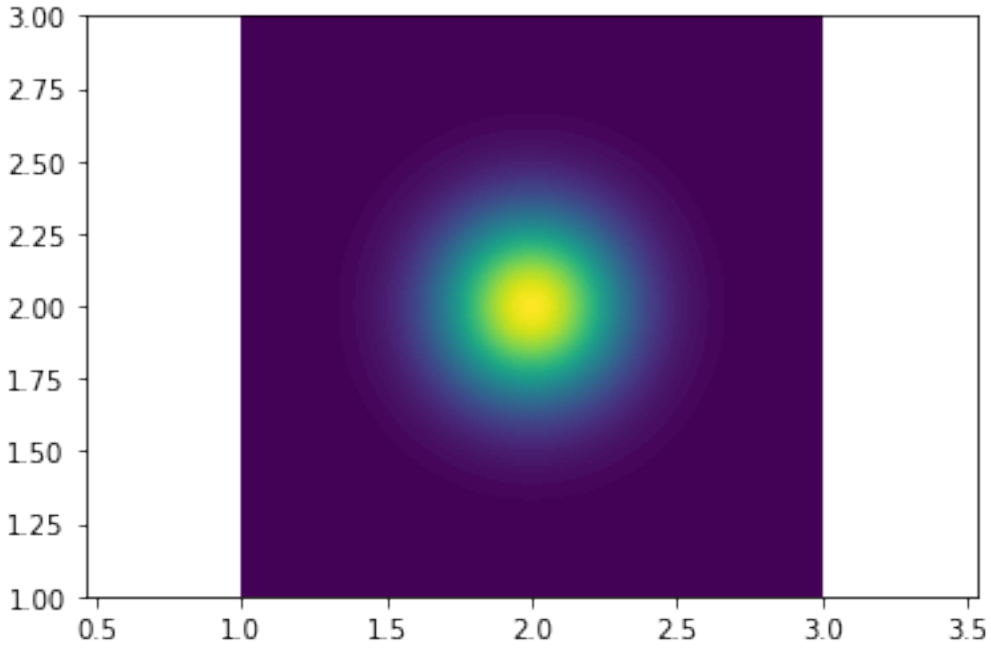
$$= p_{\mathbf{z}}(\mathbf{z}) \left|\det \frac{\partial \mathbf{f}_\theta}{\partial \mathbf{z}}\right|^{-1}. \tag{5}$$

The marginal $p_{\mathbf{x}}(\mathbf{x};\theta)$ can be **evaluated explicitly**!

[13]:
```
import numpy as np
import torch
import matplotlib.pyplot as plt
from scipy import stats
```

[14]:
```
# Latent space
z1         = np.linspace(1.0, 3.0, num=100)
z2         = np.linspace(1.0, 3.0, num=100)
z1_s, z2_s = np.meshgrid(z1,z2)
z_field    = np.stack([z1_s,z2_s], axis=2)

# Define a Gaussian distribution centered on 2,2
z_pdf      = stats.multivariate_normal(mean=[2.0,2.0], cov=0.05)

# Plot distribution
plt.contourf(z1_s,z2_s,z_pdf.pdf(z_field), levels=100)
plt.axis('equal')

# Check the value of the integral. Should be one!
integral   = np.trapz(np.trapz(z_pdf.pdf(z_field),z_field[:
 ↪,0,1],axis=0),z_field[0,:,0])
print(integral)
z_field_t  = torch.tensor(z_field)
```

```
0.9999842917540005
```



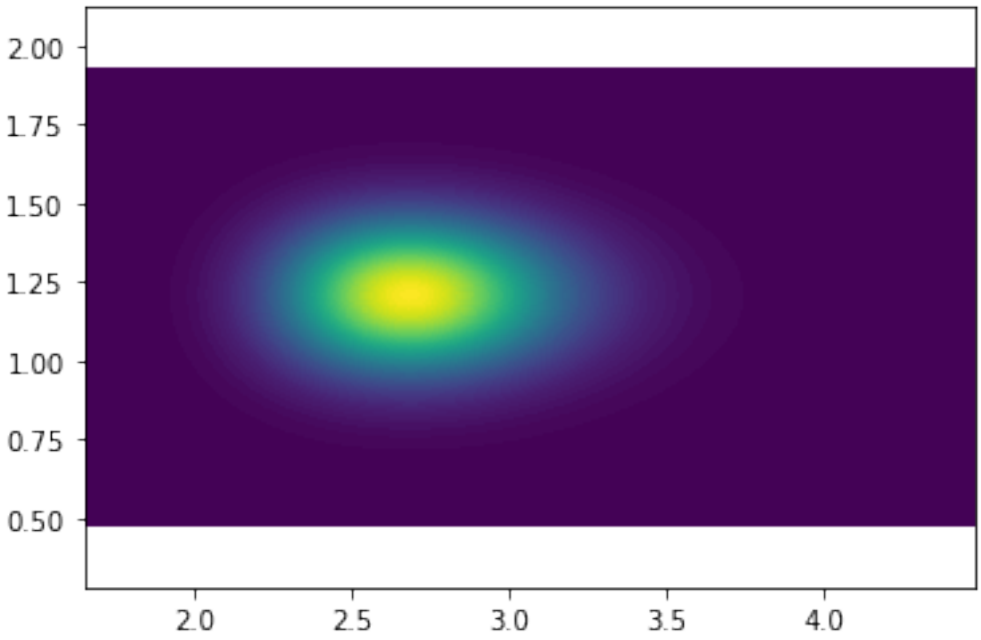
[15]:
```
# An invertible function
def f(z1, z2):
    return torch.exp(z1/2),torch.log(1.0+z2)**2

# Its inverse
def f_inv(x1,x2):
    return 2.0*torch.log(x1), torch.exp(x2**0.5)-1.0

# Evaluation on z_field_t
x_field   = np.concatenate(f(z_field_t[...,0:1],z_field_t[...,1:2]),axis=-1)
```

[16]:
```
#
def det_jacobian(x_field):
    # Computes the determinant of the jacobian f_inv(x1, x2)
    det_jac = np.zeros((x_field.shape[0], x_field.shape[1]))
    x_field_t = torch.tensor(x_field)
    x_field_t.requires_grad_(True)

    # Forward pass through f_inv
    xs = torch.cat(f_inv(x_field_t[...,0,None],x_field_t[...,1,None]),dim=-1)

    for i in range(x_field.shape[0]):
        for j in range(x_field.shape[1]):
            # Use backpropagation to evaluate the gradients
            xs[i, j, 0].backward(retain_graph=True)
            # Get the gradients
            gradx1 = x_field_t.grad[i,j].data.numpy()
            # Use backpropagation to evaluate the gradients
            xs[i, j, 1].backward(retain_graph=True)
            # Get the gradients
            gradx2 = x_field_t.grad[i,j].data.numpy()
            # Determinant is simply the product of the derivatives in that case
            det_jac[i,j] = gradx1[0]*gradx2[1]
    return det_jac
```

[17]:
```
# Use the change of variable rule to get the distribution of x=f(z)
pz = z_pdf.pdf(z_field)
px = pz * np.abs(det_jacobian(x_field))

# Plot distribution
plt.contourf(x_field[...,0],x_field[...,1],px,levels=100)
plt.axis('equal')

# Check the value of the integral
integral = np.trapz(np.trapz(px, x_field[:,0,1], axis=0),x_field[0,:,0])
```

```
print(integral)
```

```
0.9999952961139768
```



The "normalizing" in "normalizing flows" refers to that: After a **change of variables** through an **invertible transformation**, we can obtain a normalized density.

The transformations $\mathbf{f}_\theta(\mathbf{z})$ are chosen in general as **very simple** (so that the Jacobian is not too difficult to evluate). However, a simple transformation does not modify that much the original transformation.

But: another idea from the authors is to **pile up a whole chain of transformations** $\mathbf{f}_{k,\theta}$, and it allows to "reach" quite complex distributions.

Flow: Sequence of **invertible transformations** composed with each other,

$$\mathbf{z}_i = \mathbf{f}_{i,\theta} \circ ... \circ \mathbf{f}_{1,\theta}(\mathbf{z}_0) \triangleq \mathbf{f}_\theta(\mathbf{z}_0).$$

At the beginning, $\mathbf{z}_0$ is chosen with a **simple distribution** (Gaussian or uniform).

Then a sequence of M invertible transformations is applied to succesive $\mathbf{z}_i$. At the end:

$$\mathbf{x} = \mathbf{z}_M.$$

Now we saw above:

$$p_{\mathbf{z}_1}(\mathbf{z}_1;\theta) = p_{\mathbf{z}_0}(\mathbf{f}_{1,\theta}^{-1}(\mathbf{z}_1)) \left|\det \frac{\partial \mathbf{f}_{1,\theta}}{\partial \mathbf{z}_0}\right|^{-1}$$

then

$$\begin{aligned} p_{\mathbf{z}_2}(\mathbf{z}_2;\theta) &= p_{\mathbf{z}_1}(\mathbf{f}_{2,\theta}^{-1}(\mathbf{z}_2)) \left|\det \frac{\partial \mathbf{f}_{2,\theta}}{\partial \mathbf{z}_1}\right|^{-1} && (6) \\ &= p_{\mathbf{z}_0}(\mathbf{f}_{1,\theta}^{-1} \circ \mathbf{f}_{2,\theta}^{-1}(\mathbf{z}_2)) \left|\det \frac{\partial \mathbf{f}_{1,\theta}}{\partial \mathbf{z}_0}\right|^{-1} \left|\det \frac{\partial \mathbf{f}_{2,\theta}}{\partial \mathbf{z}_1}\right|^{-1} && (7) \end{aligned}$$

And by recurrence

$$p_{\mathbf{x}}(\mathbf{x};\theta) = p_{\mathbf{z}_m}(\mathbf{z}_m;\theta) = p_{\mathbf{z}_0}(\mathbf{f}_\theta^{-1}(\mathbf{x})) \prod_{m=1} \left|\det \frac{\partial \mathbf{f}_{m,\theta}}{\partial \mathbf{z}_{m-1}}\right|^{-1}.$$

Consider the family of

$$f_\theta(\mathbf{z}) = \mathbf{z} + \mathbf{u}h(\mathbf{w}^T\mathbf{z} + b)$$

parameterized by $\theta \triangleq \mathbf{u}, \mathbf{w}, b \in \mathbb{R}^m \times \mathbb{R}^m \times \mathbb{R}$.

They are referred to as **planar flows**.

The logarithm of the determinant of the Jacobian

$$\begin{aligned} \psi(\mathbf{z}) &\triangleq h'(\mathbf{w}^T\mathbf{z} + b)\mathbf{w} && (8) \\ |\det \frac{\partial f}{\partial \mathbf{z}}| &= |\det(\mathbf{I} + \mathbf{u}\psi(\mathbf{z})^T)| = |1 + \mathbf{u}^T\psi(\mathbf{z})| && (9) \end{aligned}$$

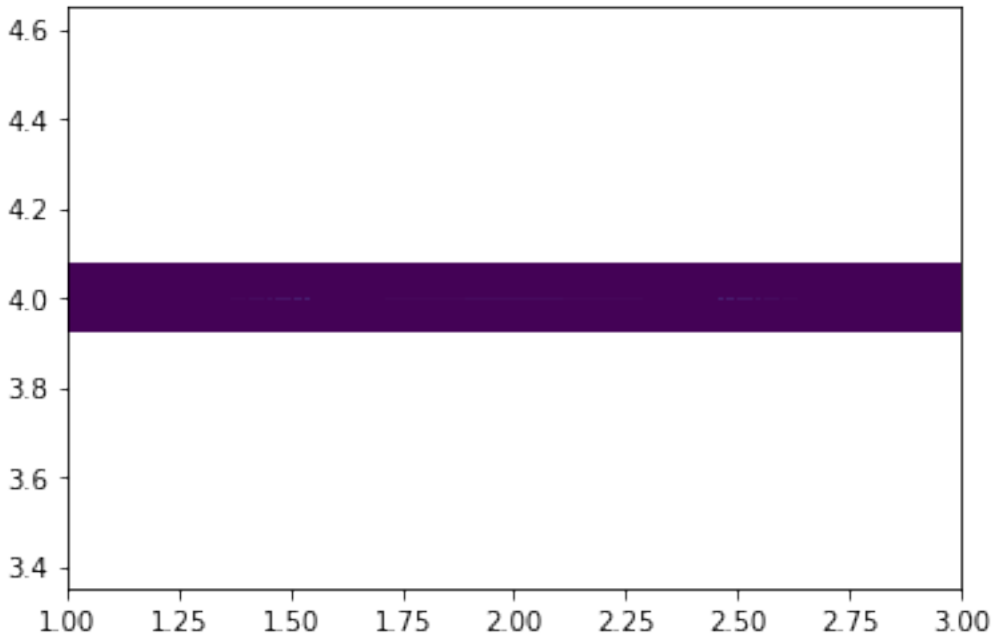
with $h'(a) = h(a)(1 - h(a))$ (property of sigmoid function).

[32]:
```
w1 = 0.0
w2 =-1.0
u1 = 0.0
u2 = 4.0
b  = 2.0

def planar_f(z1, z2):
    h = torch.sigmoid(w1*z1+w2*z2+b)
    return z1+u1*h,z2+u2*h

def planar_detlogjac(z1, z2):
    h = torch.sigmoid(w1*z1+w2*z2+b)
    return 1.0/np.abs(1.0+h*(1.0-h)*(u1*w1+u2*w2))
```

```
# Evaluation on z_field _t
x_field = np.concatenate(planar_f(z_field_t[..., 0:1],z_field_t[...,1:
↪2]),axis=-1)
```

[33]:
```
# Use the change of variable rule to get the distribution of x=f(z)
pz     = z_pdf.pdf(z_field)
detlog = planar_detlogjac(z_field_t[...,0],z_field_t[...,1])
px     = torch.mul(torch.Tensor(pz),detlog)

# Plot x distribution
plt.contourf(x_field[...,0],x_field[...,1],px,levels=100)
plt.axis('equal')

# Check the value of the integral.
integral = np.trapz(np.trapz(px, x_field[:,0,1],axis=0),x_field[0,:,0])
print(integral)
```

```
1.1157221577991936
```



Note that not any choice of parameters will give us an invertible mapping!

Since our support for $\mathbf{z}$ may be the whole $\mathbb{R}^m$, we must ensure invertibility on $\mathbb{R}^m$.

We have seen above that:

$$|\det \frac{\partial f}{\partial \mathbf{z}}| = |1 + \mathbf{u}^T \psi(\mathbf{z})| = |1 + \mathbf{u}^T h'(\mathbf{w}^T \mathbf{z} + b)\mathbf{w}|.$$

We know that h' is the derivative of the sigmoid function, so it is always **strictly positive** and **strictly inferior to 1**.

Hence a **sufficient condition** for invertibility is:

$$\mathbf{u}^T\mathbf{w} \geq -1$$

since it will imply

$$\mathbf{u}^T\mathbf{w}h'(\mathbf{w}^T\mathbf{z} + b) > -1,$$

and

$$|\det \frac{\partial f}{\partial \mathbf{z}}| > 0.$$

Another family of transformations proposed by the authors is what they call a **radial flow** which performs the distortions around a reference point $\mathbf{z}_0$.

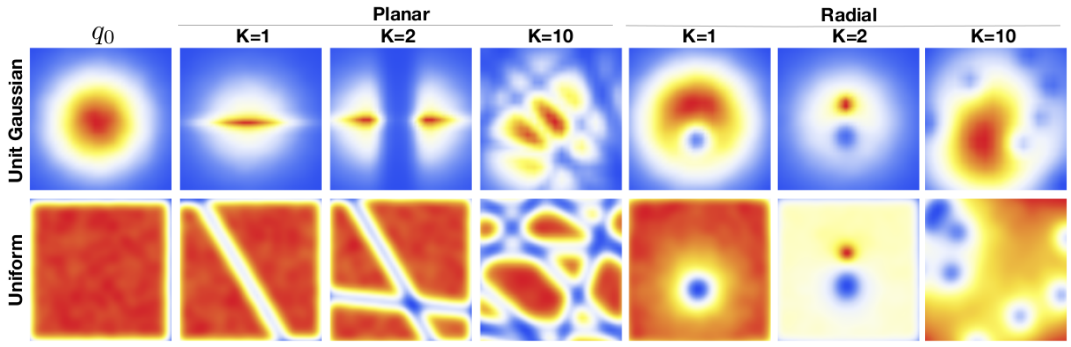
It is parameterized as:

$$f(\mathbf{z}) = \mathbf{z} + \beta h(\alpha, r)(\mathbf{z} - \mathbf{z}_0),$$

with $r = |\mathbf{z} - \mathbf{z}_0|$ and $h(\alpha, r) = \frac{1}{\alpha + r}$.

One can show that:

$$|\det \frac{\partial f}{\partial \mathbf{z}}| = [1 + \beta h(\alpha, r)]^{d-1}[1 + \beta h(\alpha, r) + \beta h'(\alpha, r)r].$$



From: Variational Inference with Normalizing Flows. Danilo Jimenez Rezende and Shakir Mohamed. Proc. of ICML, 2015

2.1 Training

The training of the normalizing flow network is done through **maximizing the likelihood** over the dataset $\mathcal{D}_{train}$

$$\theta^* = \arg\max_\theta \prod_{\mathbf{x}^k \in \mathcal{D}_{train}} p_{\mathbf{x}}(\mathbf{x}^k; \theta). \tag{10}$$

where $p_{\mathbf{x}}(\mathbf{x}; \theta)$ is the density that we build through normalizing flows, with parameters θ.

Remember:

$$p_{\mathbf{x}}(\mathbf{x}; \theta) = p_{\mathbf{z}_0}(\mathbf{f}_\theta^{-1}(\mathbf{x})) \left| \det \frac{\partial \mathbf{f}_\theta^{-1}}{\partial \mathbf{x}} \right|.$$

By minimizing the minus logarithm of this likelihood

$$\theta^* = \arg\min_\theta - \sum_{\mathbf{x}_k \in \mathbf{D}} \log p_{\mathbf{x}}(\mathbf{x}_k; \theta) = \arg\min_\theta - \sum_{\mathbf{x}_k \in \mathbf{D}} (\log p_{\mathbf{z}_0}(\mathbf{f}_\theta^{-1}(\mathbf{x})) + \log |\det(\frac{\partial \mathbf{f}_\theta^{-1}}{\partial \mathbf{x}})|). \tag{11}$$

Then:

$$|\det \frac{\partial \mathbf{f}_\theta^{-1}}{\partial \mathbf{x}}| = \prod_{m=1} \left| \det \frac{\partial \mathbf{f}_{m,\theta}^{-1}}{\partial \mathbf{z}_m} \right| = \prod_{m=1} \left| \det \frac{\partial \mathbf{f}_{m,\theta}}{\partial \mathbf{z}_{m-1}} \right|^{-1}.$$

Finally:

$$\theta^* = \arg\min_\theta \sum_{\mathbf{x}_k \in \mathbf{D}} (-\log p_{\mathbf{z}_0}(\mathbf{f}_\theta^{-1}(\mathbf{x}))) + \sum_{m=1} \log \left| \det \frac{\partial \mathbf{f}_{m,\theta}}{\partial \mathbf{z}_{m-1}} \right|. \tag{12}$$

This is an **exact likelihood evaluation** which is quite better than most other methods!

[34]:
```
import torch.nn as nn
```

[35]:
```
# Inspired from:
# https://www.ritchievink.com/blog/2019/10/11/
 ↪sculpting-distributions-with-normalizing-flows/
class Planar(nn.Module):
    def __init__(self, size=1, init_sigma=0.01):
        super().__init__()
        # Random initialization of u, w
        self.u = nn.Parameter(torch.randn(1, size).normal_(0, init_sigma))
        self.w = nn.Parameter(torch.randn(1, size).normal_(0, init_sigma))
        self.b = nn.Parameter(torch.zeros(1))

    @property
    def normalized_u(self):
```

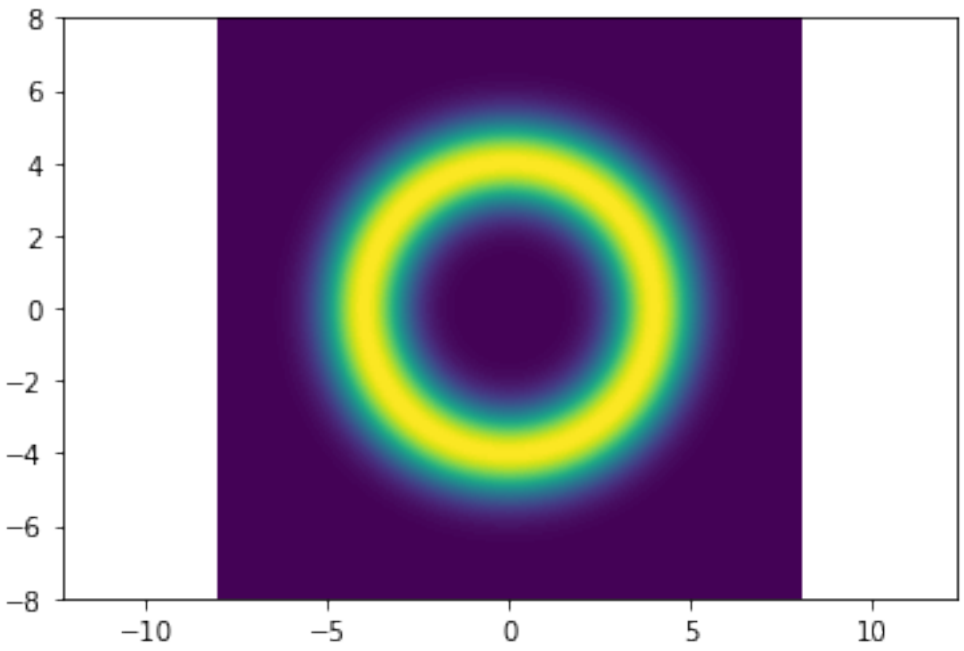
```
        """
        A trick to ensure invertibility condition. u^tw>=-1
        See Appendix A.1
        Rezende et al. Variational Inference with Normalizing Flows
        https://arxiv.org/pdf/1505.05770.pdf
        """
        # Apply softplus
        def m(x):
            return -1 + torch.log(1 + torch.exp(x))
        wtu      = torch.matmul(self.w, self.u.t())
        w_div_w2 = self.w / torch.norm(self.w)
        return self.u + (m(wtu) - wtu) * w_div_w2

    def psi(self, z):
        return self.h_prime(z @ self.w.t() + self.b) @ self.w

    # Non-linear function
    def h(self, x):
        return torch.tanh(x)

    # Derivative of h
    def h_prime(self, z):
        return 1 - torch.tanh(z) ** 2

    # Forward pass
    def forward(self, z):
        # In the sequence: On the first call, z alone, then (z',sum of log det␣
↪jac)
        if isinstance(z, tuple):
            z, accumulating_ldj = z
        else:
            z, accumulating_ldj = z, 0
        psi = self.psi(z)

        # Trick proposed by Rezende et al. to ensure that f is invertible
        u = self.normalized_u

        # Determinant of Jacobian
        det = (1 + psi @ u.t())

        # log |det Jac|
        logdetjac = torch.log(torch.abs(det) + 1e-6)

        # f(z) = z +  u h( w^T z+b)
        fz = z + (u * self.h(z @ self.w.t() + self.b))
```

```
        # Returns the function value and the accumulated values of the log␣
↪determinants of the Jacobian
        return fz, logdetjac + accumulating_ldj
```

[36]:
```
# A target density we will try to approximate
def target_density(z):
    z1, z2 = z[..., 0], z[..., 1]
    norm   = (z1**2 + z2**2)**0.5
    u      = 0.5 * ((norm - 4) / 0.8) ** 2
    return torch.exp(-u)
```

[37]:
```
z1         = np.linspace(-8.0, 8.0, num=100)
z2         = np.linspace(-8.0, 8.0, num=100)
z1_s, z2_s = np.meshgrid(z1,z2)
z_field    = np.stack([z1_s,z2_s], axis=2)

# Plot the target density
plt.contourf(z1_s, z2_s, target_density(torch.Tensor(z_field)),levels=100)
plt.axis('equal')
```

[37]:
```
(-8.0, 8.0, -8.0, 8.0)
```



[38]:
```
from torch.distributions import Normal

def det_loss(mu, log_var, z_0, z_k, ldj, beta):
```

```
    # Note that I assume uniform prior here.
    # So P(z) is constant and not modelled in this loss function
    batch_size = z_0.size(0)

    # Here we evaluat log q_{z_0}(z_0)
    mlog_qz0 = Normal(mu, torch.exp(0.5 * log_var)).log_prob(z_0)

    # Qzk = Qz0 + sum(log det jac)
    log_qzk = mlog_qz0.sum() - ldj.sum()

    # P(x|z)
    nll = -torch.log(target_density(z_k) + 1e-7).sum() * beta
    return (log_qzk + nll) / batch_size
```

[39]:
```
class Flow(nn.Module):
    def __init__(self, dimension=2, K=10):
        super().__init__()
        self.K      = K
        # Concatenation of K planar functions
        self.flow   = nn.Sequential(*[ Planar(dimension) for _ in range(self.K)])
        # Distribution of z0. Close to (0,1)
        self.mu     = nn.Parameter(torch.randn(dimension,).normal_(0, 0.01))
        self.log_var= nn.Parameter(torch.randn(dimension,).normal_(1, 0.01))

    def forward(self, dimension):
        # Generate z0
        std = torch.exp(0.5 * self.log_var)
        eps = torch.randn(dimension)
        z0  = self.mu + eps * std
        # Apply the flow on z0
        zk, logdetjac = self.flow(z0)
        return z0, zk, logdetjac, self.mu, self.log_var
```

[40]:
```
def train_flow(flow, dimension, epochs=1000):
    optim = torch.optim.Adam(flow.parameters(), lr=1e-2)

    for i in range(epochs):
        # Forward pass
        z0, zk, ldj, mu, log_var = flow(dimension=dimension)
        loss = det_loss(mu=mu,
                        log_var=log_var,
                        z_0=z0,
                        z_k=zk,
                        ldj=ldj,
                        beta=1)
        loss.backward()
        optim.step()
```

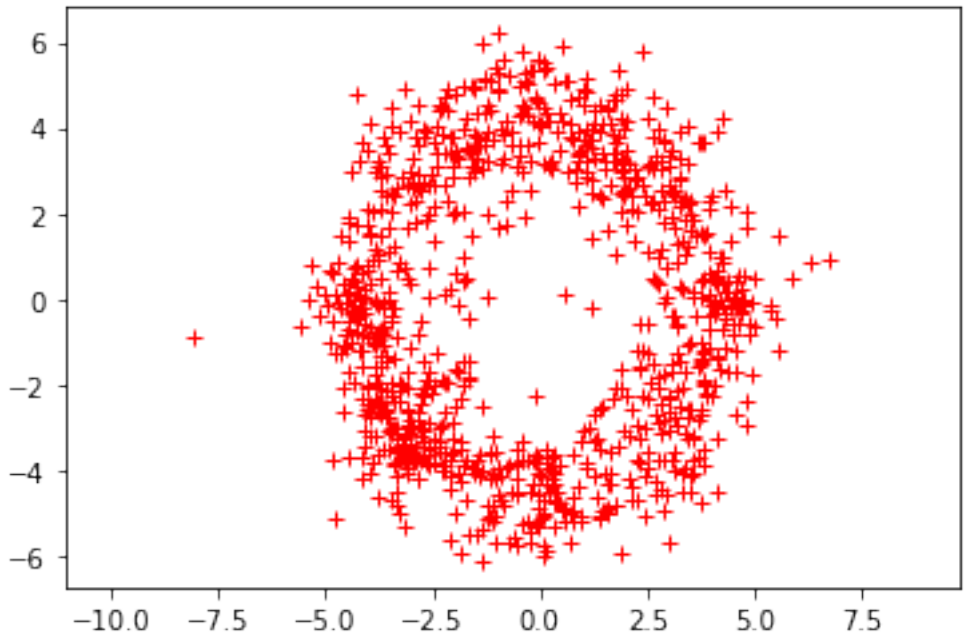
```
        optim.zero_grad()
```

[41]:
```
flow = Flow(dimension=2,K=20)
```

[42]:
```
train_flow(flow=flow, dimension=2, epochs=15000)
```

[43]:
```
# Let us test our normalizing flow
xs = []
for i in range(1000):
    z0, zk, __, __, __ = flow(dimension=2)
    xs.append(zk.detach().numpy())

xs = np.concatenate(xs)
plt.plot(xs[:,0],xs[:,1],'r+')
plt.axis('equal')
```

[43]:
```
(-8.803603339195252, 7.519385027885437, -6.730954360961914, 6.8212635040283205)
```



2.2 Sampling and inference

Sampling is particularly easy:

$$\mathbf{z} \sim p_{\mathbf{z}}(\mathbf{z}), \tag{13}$$

$$\mathbf{x} = \mathbf{f}_\theta(\mathbf{z}). \tag{14}$$

It requires an efficient evaluation of $\mathbf{f}_\theta$.

Latent representation is inferred by applying the inverse transformation, we do not need $q(\mathbf{z}|\mathbf{x};\phi)$

$$\mathbf{z} = \mathbf{f}_\theta^{-1}(\mathbf{x}). \tag{15}$$

Likelihood evalution:

- You gets some data $\mathbf{x}$.
- Apply the inverse transformation

$$\mathbf{z} = \mathbf{f}_\theta^{-1}(\mathbf{x}). \tag{16}$$

- Evaluates $p_\mathbf{z}(\mathbf{z}) \left|\det \frac{\partial \mathbf{f}_\theta}{\partial \mathbf{z}}\right|^{-1}$.

It requires an efficient evaluation of $\mathbf{f}_\theta^{-1}$ and a simple form for $p_\mathbf{z}$.

As seen above, anothr critical point is the **evalation of the determinants of the $m \times m$ Jacobian matrices**.

A priori $O(m^3)$: **not very efficient**!

Idea: Use tranformations that ensure some structure in the Jacobian so that the determinant is not that heavy to compute. Ideas?

Suppose $\mathbf{x}_i = \mathbf{f}_i(\mathbf{z})$ does **only depend on the $\mathbf{z}_j$ with $j \geq i$.**

Then the Jacobian

$$\mathbf{J} = \begin{pmatrix} \frac{\partial \mathbf{f}_1}{\partial \mathbf{z}_1} & 0 & \dots & 0 \\ \frac{\partial \mathbf{f}_1}{\partial \mathbf{z}_2} & \frac{\partial \mathbf{f}_2}{\partial \mathbf{z}_2} & \dots & 0 \\ \dots & \dots & \dots & \dots \\ \frac{\partial \mathbf{f}_1}{\partial \mathbf{z}_2} & \frac{\partial \mathbf{f}_2}{\partial \mathbf{z}_2} & \dots & \frac{\partial \mathbf{f}_d}{\partial \mathbf{z}_d} \end{pmatrix}$$

has lower **triangular structure** and the determinant can be computed in linear time:

$$\det \mathbf{J} = \prod_{i=1}^{d} \frac{\partial \mathbf{f}_i}{\partial \mathbf{z}_i}.$$

NICE: Non-linear independent components estimation, Laurent Dinh, David Krueger, Yoshua Bengio. ArXiv 2014.

Partition the variables z into two disjoint subsets

- $\mathbf{z}_{1:k}$
- $\mathbf{z}_{k+1:m}$

for some $1 \leq k < m$.

The forward mapping $f_\theta : \mathbf{z} \to \mathbf{x}$ is chosen as:

$$\mathbf{x}_{1:k} = \mathbf{z}_{1:k} \tag{17}$$
$$\mathbf{x}_{k+1:m} = \mathbf{z}_{k+1:m} + m_\theta(\mathbf{z}_{1:k}) \tag{18}$$

($m_\theta(.)$ is a **neural network with parameters** θ, with k input units, $m-k$ output units)

Inverse mapping is trivial:

$$\mathbf{z}_{1:k} = \mathbf{x}_{1:k} \tag{19}$$
$$\mathbf{z}_{k+1:m} = \mathbf{x}_{k+1:m} - m_\theta(\mathbf{x}_{1:k}) \tag{20}$$

Jacobian of forward mapping:

$$\mathbf{J} = \frac{\partial f_\theta}{\partial \mathbf{z}} \tag{21}$$
$$= \begin{pmatrix} \mathbf{I}_k & \mathbf{0} \\ \frac{\partial m_\theta}{\partial \mathbf{z}_{1:k}} & \mathbf{I}_{m-k} \end{pmatrix} \tag{22}$$

from which you deduce

$$\det(\mathbf{J}) = 1.$$

This is called a **volume preserving transformation**.

These are called **additive coupling layers** and they are **composed together** (with different, arbitrary partitions of the indices in each layer)

Final layer is **anisotropic scaling**:

$$\mathbf{x}_i = s_i \mathbf{z}_i \text{ at dimension } i.$$

The **reciprocal function** is trivial

$$\mathbf{z}_i = \frac{1}{s_i} \mathbf{x}_i \text{ at dimension } i.$$

The determinant of the Jacobian:

$$\prod_{i=1}^{m} s_i.$$

(a) Model trained on MNIST

(b) Model trained on TFD

(c) Model trained on SVHN

(d) Model trained on CIFAR-10

They apply it to inpaiting with $\mathbf{x}_O$ the observed data and $\mathbf{x}_H$ the hidden data with is optimized, with the normalizing flow parameter being kept as constant:

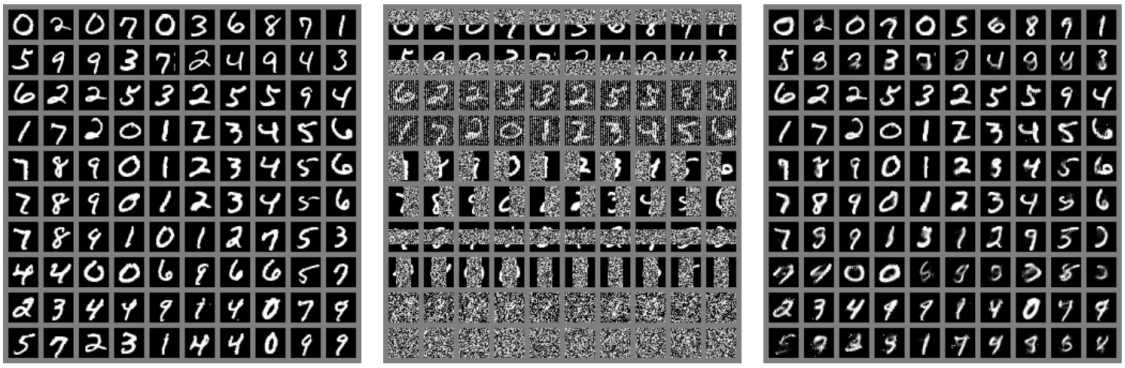
$$\mathbf{x}_{H,k+1} = \mathbf{x}_{H,k} + \alpha_k \frac{\partial \log p_{\mathbf{x}}(\mathbf{x}_O, \mathbf{x}_{H,k})}{\partial \mathbf{x}_{H,k}}.$$

with

$$p_{\mathbf{x}}(\mathbf{x}_O, \mathbf{x}_H) = p_{\mathbf{z}}(\mathbf{z}) \left| \det \frac{\partial \mathbf{f}_\theta}{\partial \mathbf{z}} \right|^{-1}.$$

with:

$$\mathbf{z} = \mathbf{f}_\theta^{-1}(\mathbf{x}_O, \mathbf{x}_H). \tag{23}$$



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

Density estimation using Real NVP.Laurent Dinh, Jascha Sohl-Dickstein, Samy Bengio. ICLR 2017

The forward mapping is a bit more complex:

$$\mathbf{x}_{1:k} = \mathbf{z}_{1:k} \tag{24}$$

$$\mathbf{x}_{k+1:m} = \mathbf{z}_{k+1:m} \odot \exp(\alpha_\theta(\mathbf{z}_{1:k})) + \mu_\theta(\mathbf{z}_{1:k}) \tag{25}$$

$\mu_\theta(\mathbf{z}_{1:k})$ and $\alpha_\theta(\mathbf{z}_{1:k})$ are neural networks.

Reciprocal function:

$$\mathbf{z}_{1:k} = \mathbf{x}_{1:k} \tag{26}$$

$$\mathbf{z}_{k+1:m} = (\mathbf{x}_{k+1:m} - \mu_\theta(\mathbf{z}_{1:k})) \odot \exp(-\alpha_\theta(\mathbf{z}_{1:k}) \tag{27}$$

and the Jacobian is

$$\mathbf{J} = \begin{pmatrix} \mathbf{I}_k & \mathbf{0} \\ \frac{\partial \mathbf{x}_{k+1:m}}{\partial \mathbf{z}_{1:k}} & diag\, \exp(\alpha_\theta(\mathbf{z}_{1:k})) \end{pmatrix} \tag{28}$$

We have

$$\det \mathbf{J} = \exp\left(\sum_j \alpha_\theta(\mathbf{z}_{1:k})_j\right).$$

Called **non-volume preserving transformation** since determinant is < 1 or > 1.