Deep learning

10.3. Non-volume preserving networks

François Fleuret

https://fleuret.org/dlc/

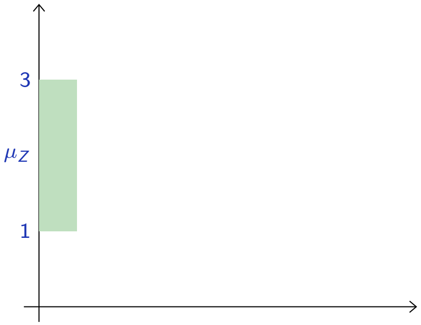
UNIVERSITÉ DE GENÈVE

A standard result of probability theory is that if f is continuous, invertible and [almost everywhere] differentiable, and $X = f^{-1}(Z)$, then

$$\forall x,\ \mu_X(x) = \mu_Z(f(x)) \left| J_f(x) \right| .$$

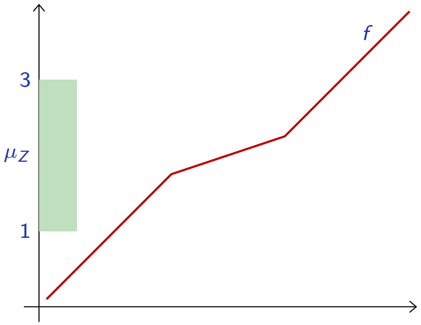
A standard result of probability theory is that if f is continuous, invertible and [almost everywhere] differentiable, and $X = f^{-1}(Z)$, then

$$\forall x,\ \mu_X(x) = \mu_Z(f(x))\,|J_f(x)|\,.$$



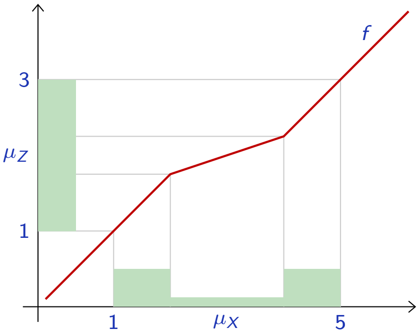
A standard result of probability theory is that if f is continuous, invertible and [almost everywhere] differentiable, and $X = f^{-1}(Z)$, then

$$\forall x, \ \mu_X(x) = \mu_Z(f(x)) \left| J_f(x) \right| .$$



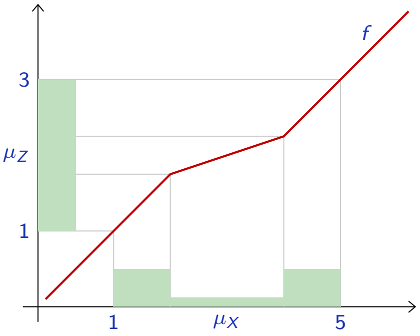
A standard result of probability theory is that if f is continuous, invertible and [almost everywhere] differentiable, and $X = f^{-1}(Z)$, then

$$\forall x, \ \mu_X(x) = \mu_Z(f(x)) \left| J_f(x) \right| .$$



A standard result of probability theory is that if f is continuous, invertible and [almost everywhere] differentiable, and $X = f^{-1}(Z)$, then

$$\forall x,\ \mu_X(x) = \mu_Z(f(x))\,|J_f(x)|\,.$$



The term $|J_f(x)|$ accounts for the local "stretching" of the space.

Since

$$\mu_X(x) = \mu_Z(f(x)) \left| J_f(x) \right|,$$

if f is a parametric function such that we can compute [and differentiate]

$$\mu_Z(f(x)) \quad \text{and} \quad \left| J_f(x) \right|,$$

given $x_1, \dots, x_N$ i.i.d $\sim \mu$, we can make μ_X fit the data by maximizing

$$\sum_n \log \mu_X(x_n)$$

Since

$$\mu_X(x) = \mu_Z(f(x)) \left| J_f(x) \right|,$$

if f is a parametric function such that we can compute [and differentiate]

$$\mu_Z(f(x)) \text{ and } \left| J_f(x) \right|,$$

given $x_1, \dots, x_N$ i.i.d $\sim \mu$, we can make μ_X fit the data by maximizing

$$\sum_n \log \mu_X(x_n) = \sum_n \log \mu_Z(f(x_n)) + \log \left| J_f(x_n) \right|.$$

Since

$$\mu_X(x) = \mu_Z(f(x))\,|J_f(x)|\,,$$

if f is a parametric function such that we can compute [and differentiate]

$$\mu_Z(f(x)) \;\text{ and }\; |J_f(x)|\,,$$

given $x_1, \dots, x_N$ i.i.d $\sim \mu$, we can make μ_X fit the data by maximizing

$$\sum_n \log \mu_X(x_n) = \sum_n \log \mu_Z(f(x_n)) + \log |J_f(x_n)|\,.$$

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

$$\log \mu_Z\left(f(x_n)\right) = -\frac{1}{2}\left(\|f(x_n)\|^2 + d \log 2\pi\right).$$

Since

$$\mu_X(x) = \mu_Z(f(x)) \, |J_f(x)| \, ,$$

if f is a parametric function such that we can compute [and differentiate]

$$\mu_Z(f(x)) \quad \text{and} \quad |J_f(x)| \, ,$$

given $x_1, \dots, x_N$ i.i.d $\sim \mu$, we can make μ_X fit the data by maximizing

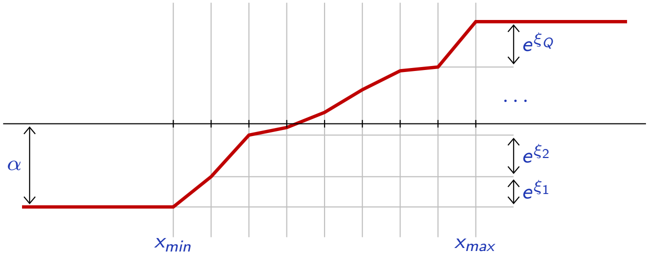
$$\sum_n \log \mu_X(x_n) = \sum_n \log \mu_Z(f(x_n)) + \log |J_f(x_n)| \, .$$

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

$$\log \mu_Z\left(f(x_n)\right) = -\frac{1}{2}\left(\|f(x_n)\|^2 + d \log 2\pi\right) .$$

We aim at $f(X) \sim \mathcal{N}(0, I)$, hence at f **normalizing** the distribution.

Consider an increasing piece-wise linear mapping with parameters $\alpha, \xi_1, \dots, \xi_Q$.



```
class PiecewiseLinear(nn.Module):
    def __init__(self, nb, xmin, xmax):
        super().__init__()
        self.xmin = xmin
        self.xmax = xmax
        self.nb = nb
        self.alpha = nn.Parameter(torch.tensor([xmin], dtype = torch.float))
        mu = math.log((xmax - xmin) / nb)
        self.xi = nn.Parameter(torch.empty(nb + 1).normal_(mu, 1e-4))

    def forward(self, x):
        y = self.alpha + self.xi.exp().cumsum(0)
        u = self.nb * (x - self.xmin) / (self.xmax - self.xmin)
        n = u.long().clamp(0, self.nb - 1)
        a = (u - n).clamp(0, 1)
        x = (1 - a) * y[n] + a * y[n + 1]
        return x
```

For $f : \mathbb{R} \to \mathbb{R}$ increasing, we have

$$|J_f(x_n)| = f'(x_n)$$

so we should minimize

$$\sum_n \frac{1}{2}\left(f(x_n)^2 + \log 2\pi\right) - \log f'(x_n).$$

To work with batches of samples, we have to compute $(f'(x_1), \dots, f'(x_N))$ with autograd.

With

$$\Phi(x_1, \dots, x_N) = f(x_1) + \dots + f(x_N)$$

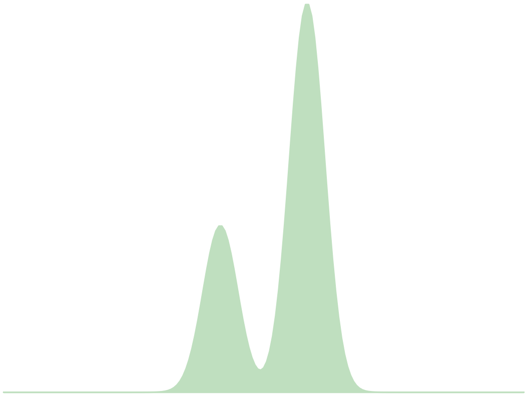
we have

$$\nabla \Phi(x_1, \dots, x_n) = \left(f'(x_1), \dots, f'(x_N)\right).$$

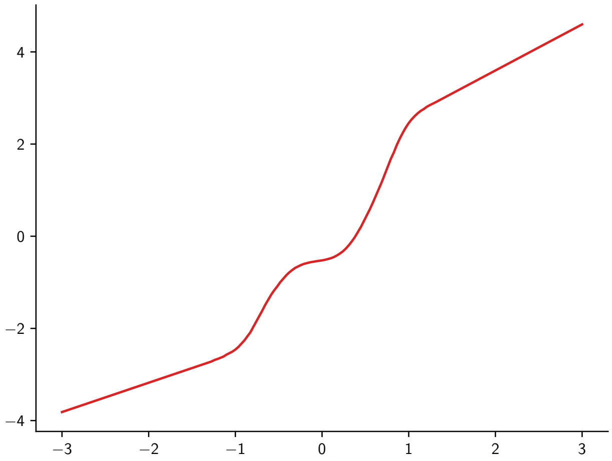
$$\mathscr{L}(f) = \frac{1}{N} \sum_n \frac{1}{2} \left(f(x_n)^2 + \log 2\pi \right) - \log f'(x_n).$$

```
for input in train_input.split(batch_size):
    input.requires_grad_()
    output = model(input)

    derivatives, = autograd.grad(
        output.sum(), input,
        retain_graph = True, create_graph = True
    )

    loss = ( 0.5 * (output**2 + math.log(2*pi)) - derivatives.log() ).mean()

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

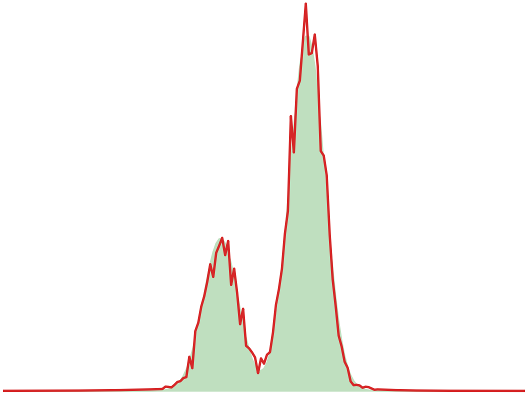
Target distribution μ.

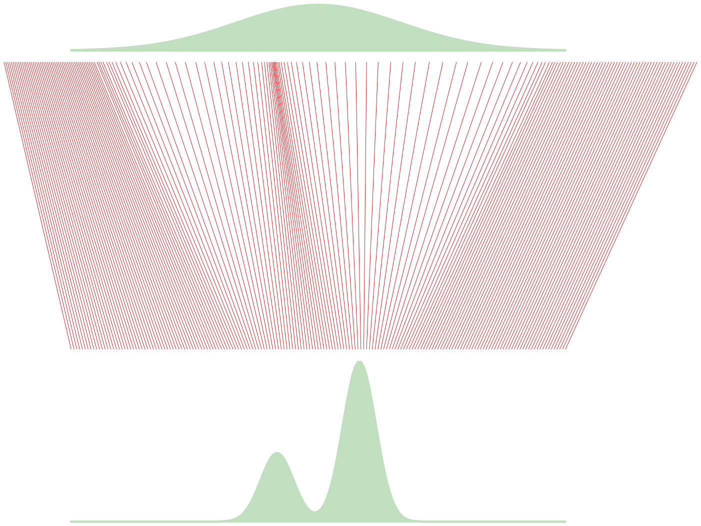


Resulting mapping $\hat{f}$.



μ_X with $X = \hat{f}^{-1}(Z)$ and $Z \sim \mathcal{N}(0, I)$.





Non-Volume Preserving networks

To apply the same idea to high dimension signals, we have to compute and differentiate $|J_f(x)|$. And to use that approach for synthesis, we can sample $Z \sim \mathscr{N}(0, I)$ and compute $f^{-1}(Z)$.

However, for standard layers:

- computing $f^{-1}(z)$ is impossible, and
- computing $|J_f(x)|$ is intractable.

To apply the same idea to high dimension signals, we have to compute and differentiate $|J_f(x)|$. And to use that approach for synthesis, we can sample $Z \sim \mathcal{N}(0, I)$ and compute $f^{-1}(Z)$.

However, for standard layers:

- computing $f^{-1}(z)$ is impossible, and
- computing $|J_f(x)|$ is intractable.

Dinh et al. (2014) introduced the **coupling layers** to address both issues.

The resulting Non-Volume Preserving network (NVP) is one form of **normalizing flow** among many techniques (Papamakarios et al., 2019).

Remember that if f is a composition

$$f = f^{(K)} \circ \dots \circ f^{(1)}$$

we have

$$J_f(x) = \prod_{k=1}^{K} J_{f^{(k)}} \left(f^{(k-1)} \circ \dots \circ f^{(1)}(x) \right),$$

hence

$$\log |J_f(x)| = \sum_{k=1}^{K} \log \left| J_{f^{(k)}} \left(f^{(k-1)} \circ \dots \circ f^{(1)}(x) \right) \right|.$$

We use here the formalism from Dinh et al. (2016).

Given a dimension d, a Boolean vector $b \in \{0,1\}^d$ and two mappings

$$s : \mathbb{R}^d \to \mathbb{R}^d$$
$$t : \mathbb{R}^d \to \mathbb{R}^d,$$

We use here the formalism from Dinh et al. (2016).

Given a dimension d, a Boolean vector $b \in \{0,1\}^d$ and two mappings

$$\begin{aligned} s &: \mathbb{R}^d \to \mathbb{R}^d \\ t &: \mathbb{R}^d \to \mathbb{R}^d, \end{aligned}$$

we define a [fully connected] coupling layer as the transformation

$$\begin{aligned} c : \mathbb{R}^d &\to \mathbb{R}^d \\ x &\mapsto b \odot x + (1 - b) \odot \Big(x \odot \exp(s(b \odot x)) + t(b \odot x) \Big) \end{aligned}$$

where $\exp$ is component-wise, and $\odot$ is the Hadamard component-wise product.

We use here the formalism from Dinh et al. (2016).

Given a dimension d, a Boolean vector $b \in \{0,1\}^d$ and two mappings

$$s : \mathbb{R}^d \to \mathbb{R}^d$$
$$t : \mathbb{R}^d \to \mathbb{R}^d,$$

we define a [fully connected] coupling layer as the transformation

$$\begin{aligned} c :\ & \mathbb{R}^d \to \mathbb{R}^d \\ & x \mapsto b \odot x + (1-b) \odot \Big(x \odot \exp(s(b \odot x)) + t(b \odot x) \Big) \end{aligned}$$

where $\exp$ is component-wise, and $\odot$ is the Hadamard component-wise product.

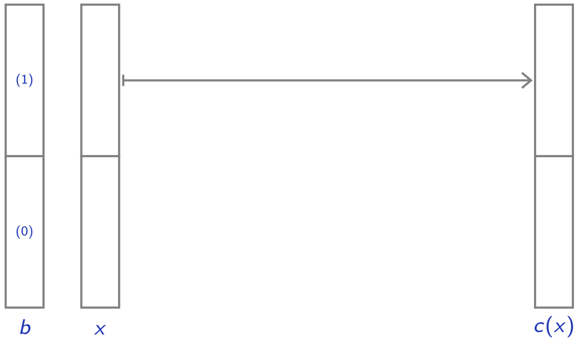
For clarity in what follows, b has all 1s first, follows by 0s, but this is not required.

$$b = (\underbrace{1, 1, \dots, 1}_{\Delta}, \underbrace{0, 0, \dots, 0}_{d-\Delta})$$

The expression

$$c(x) = b \odot x + (1 - b) \odot \Big(x \odot \exp(s(b \odot x)) + t(b \odot x)\Big)$$

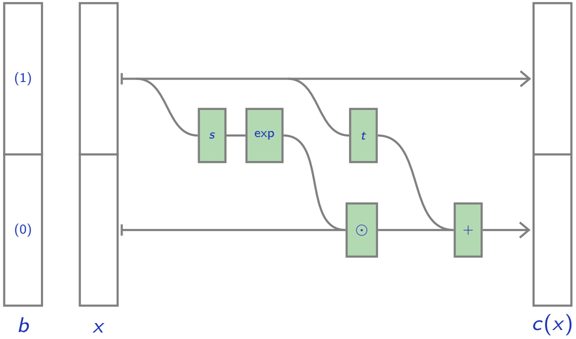
can be understood as: forward $b \odot x$ unchanged,



The expression

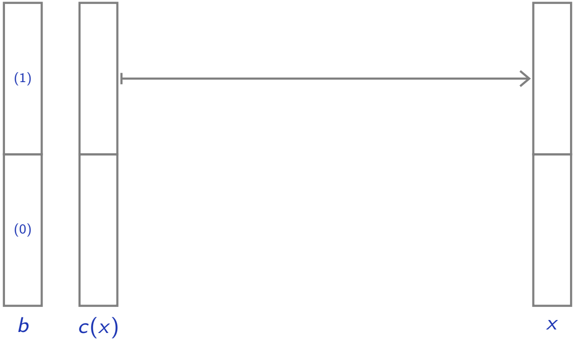
$$c(x) = b \odot x + (1 - b) \odot \Big(x \odot \exp(s(b \odot x)) + t(b \odot x)\Big)$$

can be understood as: forward $b \odot x$ unchanged, and apply to $(1 - b) \odot x$ an invertible transformation parametrized by $b \odot x$.



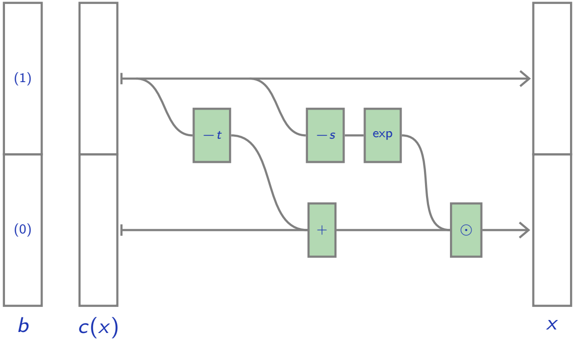
The consequence is that c is invertible, and if $y = c(x)$

$$x = b \odot y + (1-b) \odot \Big(y - t(b \odot y)\Big) \odot \exp(-s(b \odot y)).$$



The consequence is that c is invertible, and if $y = c(x)$

$$x = b \odot y + (1 - b) \odot \Big(y - t(b \odot y)\Big) \odot \exp(-s(b \odot y)).$$



The second property of this mapping is the simplicity of its Jacobian.

$$J_c(x) = \left(\begin{array}{ccc|ccc} 1 & & & & & \\ & \ddots & & & (0) & \\ & & 1 & & & \\ \hline & & & \exp(s_{\Delta+1}(x \odot b)) & & \\ & (\neq 0) & & & \ddots & \\ & & & & & \exp(s_d(x \odot b)) \end{array}\right)$$

and we have

$$\begin{aligned} \log |J_c(x)| &= \sum_{i:b_i=0} s_i(x \odot b) \\ &= \sum_i \left((1-b) \odot s(x \odot b)\right)_i . \end{aligned}$$

```
dim = 6

x = torch.randn(1, dim).requires_grad_()
b = torch.zeros(1, dim)
b[:, :dim//2] = 1.0

s = nn.Sequential(nn.Linear(dim, dim), nn.Tanh())
t = nn.Sequential(nn.Linear(dim, dim), nn.Tanh())

c = b * x + (1 - b) * (x * torch.exp(s(b * x)) + t(b * x))

# Flexing a bit
j = torch.cat([autograd.grad(c_k, x, retain_graph=True)[0] for c_k in c[0]])

print(j)
```

```
dim = 6

x = torch.randn(1, dim).requires_grad_()
b = torch.zeros(1, dim)
b[:, :dim//2] = 1.0

s = nn.Sequential(nn.Linear(dim, dim), nn.Tanh())
t = nn.Sequential(nn.Linear(dim, dim), nn.Tanh())

c = b * x + (1 - b) * (x * torch.exp(s(b * x)) + t(b * x))

# Flexing a bit
j = torch.cat([autograd.grad(c_k, x, retain_graph=True)[0] for c_k in c[0]])

print(j)
```

prints

```
tensor([[ 1.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000],
        [ 0.0000,  1.0000,  0.0000,  0.0000,  0.0000,  0.0000],
        [ 0.0000,  0.0000,  1.0000,  0.0000,  0.0000,  0.0000],
        [ 0.4001, -0.3774, -0.9410,  1.0074,  0.0000,  0.0000],
        [-0.1756,  0.0409,  0.0808,  0.0000,  1.2412,  0.0000],
        [ 0.0875, -0.3724, -0.1542,  0.0000,  0.0000,  0.6186]])
```

To recap, with $f^{(k)}, k = 1, \dots, K$ coupling layers,

$$f = f^{(K)} \circ \dots \circ f^{(1)},$$

and $x_n^{(0)} = x_n$ and $x_n^{(k)} = f^{(k)}\left(x_n^{(k-1)}\right)$,

To recap, with $f^{(k)}, k = 1, \dots, K$ coupling layers,

$$f = f^{(K)} \circ \dots \circ f^{(1)},$$

and $x_n^{(0)} = x_n$ and $x_n^{(k)} = f^{(k)}\left(x_n^{(k-1)}\right)$, we train by minimizing

$$\mathscr{L}(f) = -\sum_n -\frac{1}{2}\left(\left\|x_n^{(K)}\right\|^2 + d \log 2\pi\right) + \sum_{k=1}^{K} \log\left|J_{f^{(k)}}\left(x_n^{(k-1)}\right)\right|,$$

with

$$\log\left|J_{f^{(k)}}(x)\right| = \sum_i \left(\left(1 - b^{(k)}\right) \odot s^{(k)}\left(x \odot b^{(k)}\right)\right)_i.$$

To recap, with $f^{(k)}, k = 1, \dots, K$ coupling layers,

$$f = f^{(K)} \circ \dots \circ f^{(1)},$$

and $x_n^{(0)} = x_n$ and $x_n^{(k)} = f^{(k)}\left(x_n^{(k-1)}\right)$, we train by minimizing

$$\mathscr{L}(f) = -\sum_n -\frac{1}{2}\left(\left\|x_n^{(K)}\right\|^2 + d \log 2\pi\right) + \sum_{k=1}^{K} \log \left| J_{f^{(k)}}\left(x_n^{(k-1)}\right)\right|,$$

with

$$\log \left| J_{f^{(k)}}(x)\right| = \sum_i \left(\left(1 - b^{(k)}\right) \odot s^{(k)}\left(x \odot b^{(k)}\right)\right)_i.$$

And to sample we just need to generate $Z \sim \mathscr{N}(0, I)$ and compute X.

A coupling layer can be implemented with

```
class NVPCouplingLayer(nn.Module):
    def __init__(self, map_s, map_t, b):
        super().__init__()
        self.map_s = map_s
        self.map_t = map_t
        self.register_buffer('b', b.unsqueeze(0))

    def forward(self, x, ldj): # ldj for log det Jacobian
        s, t = self.map_s(self.b * x), self.map_t(self.b * x)
        ldj = ldj + ((1 - self.b) * s).sum(1)
        y = self.b * x + (1 - self.b) * (torch.exp(s) * x + t)
        return y, ldj

    def invert(self, y):
        s, t = self.map_s(self.b * y), self.map_t(self.b * y)
        return self.b * y + (1 - self.b) * (torch.exp(-s) * (y - t))
```

The `forward` here computes both the image of x and the update on the accumulated determinant of the Jacobian, i.e.

$$(x, u) \mapsto (f(x), u + \log |J_f(x)|) .$$

We can then define a complete network with one-hidden layer tanh MLPs for the s and t mappings

```
class NVPNet(nn.Module):
    def __init__(self, dim, hidden_dim, depth):
        super().__init__()
        b = torch.empty(dim)
        self.layers = nn.ModuleList()
        for d in range(depth):
            if d%2 == 0:
                i = torch.randperm(b.numel())[0:b.numel() // 2]
                b.zero_()[i] = 1
            else:
                b = 1 - b
            map_s = nn.Sequential(nn.Linear(dim, hidden_dim), nn.Tanh(),
                                  nn.Linear(hidden_dim, dim))
            map_t = nn.Sequential(nn.Linear(dim, hidden_dim), nn.Tanh(),
                                  nn.Linear(hidden_dim, dim))
            self.layers.append(NVPCouplingLayer(map_s, map_t, b.clone()))

    def forward(self, x, ldj):
        for m in self.layers: x, ldj = m(x, ldj)
        return x, ldj

    def invert(self, y):
        for m in reversed(self.layers): y = m.invert(y)
        return y
```

And the log-proba of individual samples of a batch

```
def LogProba(x, ldj):
    log_p = - 0.5 * (x**2 + math.log(2*pi)).sum(1) + ldj
    return log_p
```

Training is achieved by maximizing the mean log-proba

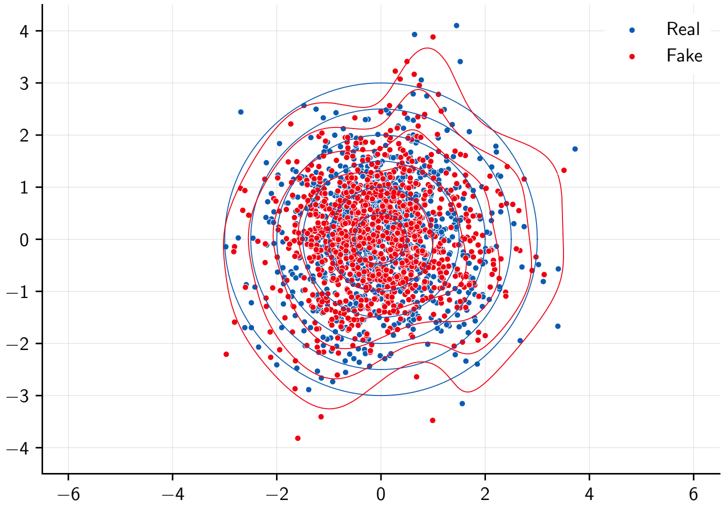
```
batch_size = 100

model = NVPNet(dim = 2, hidden_dim = 2, depth = 4)
optimizer = optim.Adam(model.parameters(), lr = 1e-2)

for e in range(args.nb_epochs):

    for input in train_input.split(batch_size):
        output, ldj = model(input, 0)
        loss = - LogProba(output, ldj).mean()
        model.zero_grad()
        loss.backward()
        optimizer.step()
```

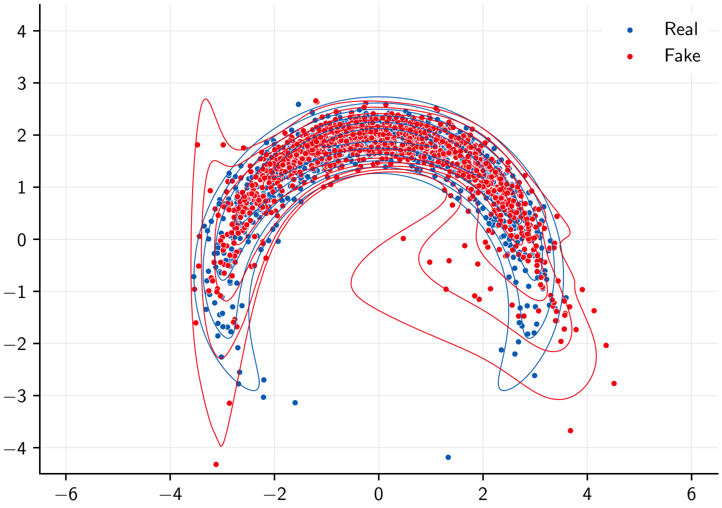
Training is achieved by maximizing the mean log-proba

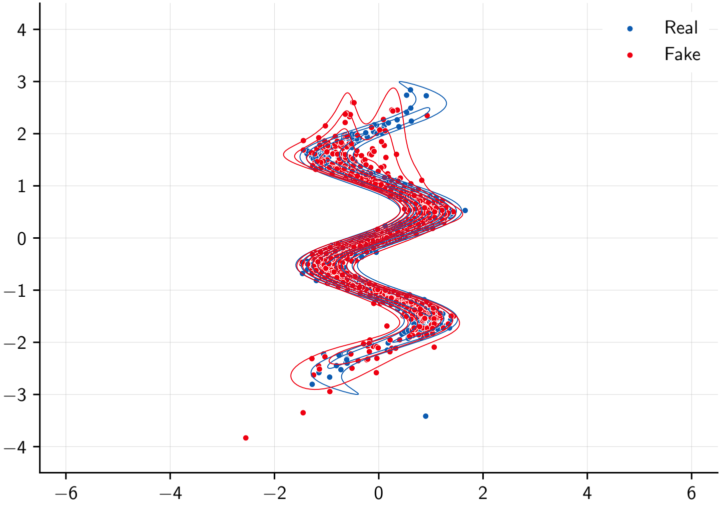
```
batch_size = 100

model = NVPNet(dim = 2, hidden_dim = 2, depth = 4)
optimizer = optim.Adam(model.parameters(), lr = 1e-2)

for e in range(args.nb_epochs):

    for input in train_input.split(batch_size):
        output, ldj = model(input, 0)
        loss = - LogProba(output, ldj).mean()
        model.zero_grad()
        loss.backward()
        optimizer.step()
```

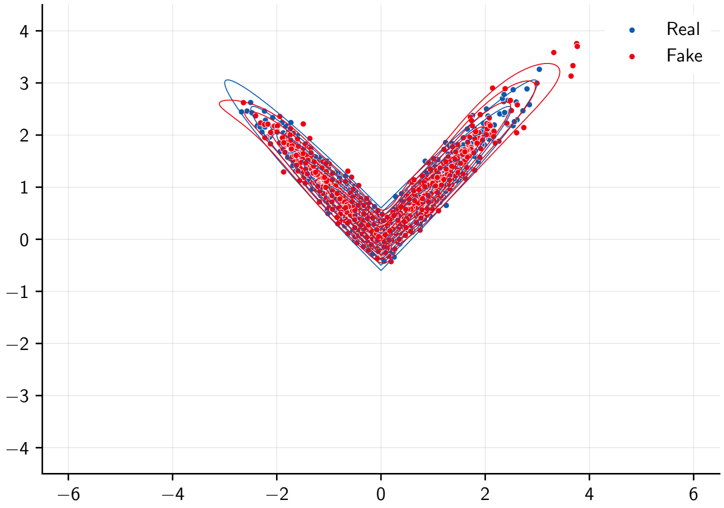
Finally, we can sample according to μ_X with

```
z = torch.randn(nb_generated_samples, 2)
x = model.invert(z)
```

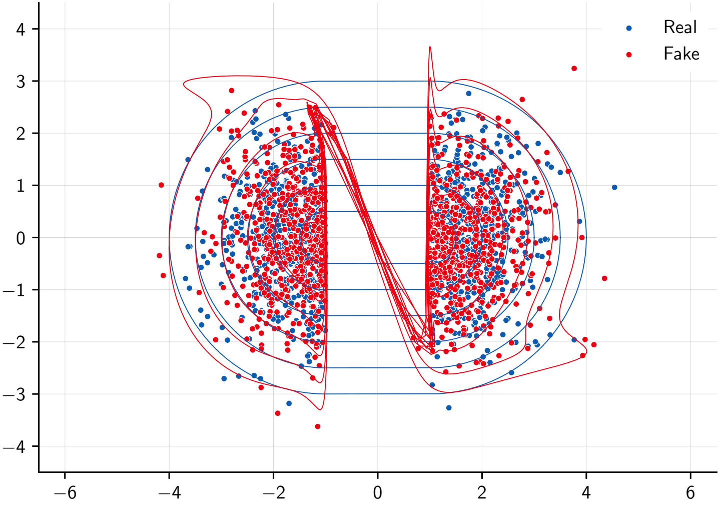

Real
Fake
4
3
2
1
0
−1
−2
−3
−4
−6
−4
−2
0
2
4
6


4
3
2
1
0
−1
−2
−3
−4
−6
−4
−2
0
2
4
6
● Real
● Fake


Real
Fake
4
3
2
1
0
−1
−2
−3
−4
−6
−4
−2
0
2
4
6



Real
Fake
4
3
2
1
0
−1
−2
−3
−4
−6
−4
−2
0
2
4
6


Real
Fake
4
3
2
1
0
−1
−2
−3
−4
−6
−4
−2
0
2
4
6

Dinh et al. (2016) apply this approach to convolutional layers by using *b*s consistent with the activation map structure, and reducing the map size while increasing the number of channels.

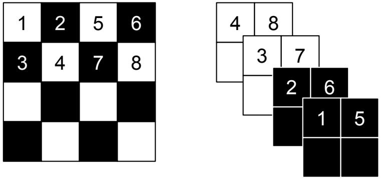
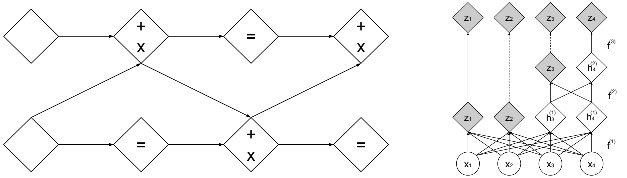


Figure 3: Masking schemes for affine coupling layers. On the left, a spatial checkerboard pattern mask. On the right, a channel-wise masking. The squeezing operation reduces the $4 \times 4 \times 1$ tensor (on the left) into a $2 \times 2 \times 4$ tensor (on the right). Before the squeezing operation,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

(Dinh et al., 2016)

They combine these layers by alternating masks, and branching out half of the channels at certain points to forward them unchanged.



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

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

Figure 4: Composition schemes for affine coupling layers.

(Dinh et al., 2016)

The structure for generating images consists of

- $\times 2$ stages
  - $\times 3$ checkerboard coupling layers,
  - a squeezing layer,
  - $\times 3$ channel coupling layers,
  - a factor-out layer.
- $\times 1$ stage
  - $\times 4$ checkerboard coupling layers
  - a factor-out layer.

The s and t mappings get more complex in the later layers.

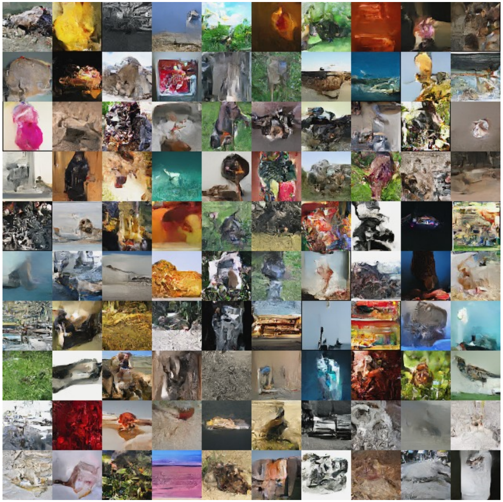


Figure 7: Samples from a model trained on *Imagenet* (64×64).

(Dinh et al., 2016)

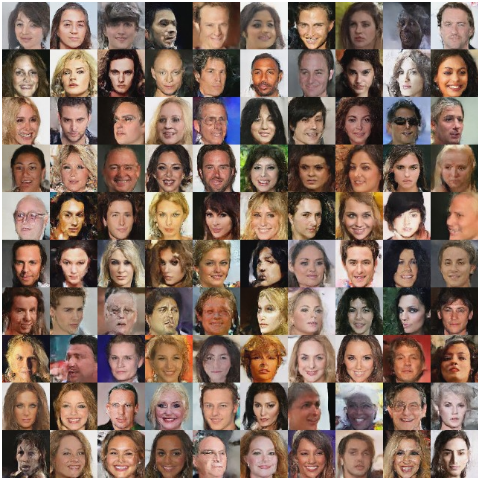


Figure 8: Samples from a model trained on *CelebA*.

(Dinh et al., 2016)

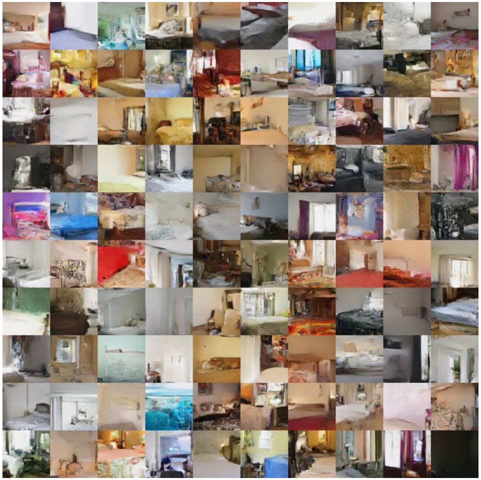


Figure 9: Samples from a model trained on *LSUN* (*bedroom* category).

(Dinh et al., 2016)

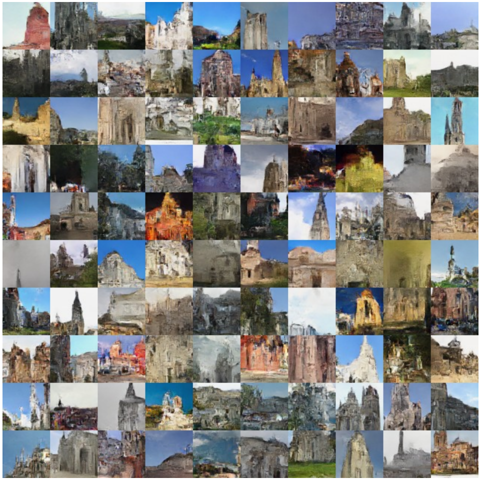


Figure 10: Samples from a model trained on *LSUN* (*church outdoor* category).

(Dinh et al., 2016)

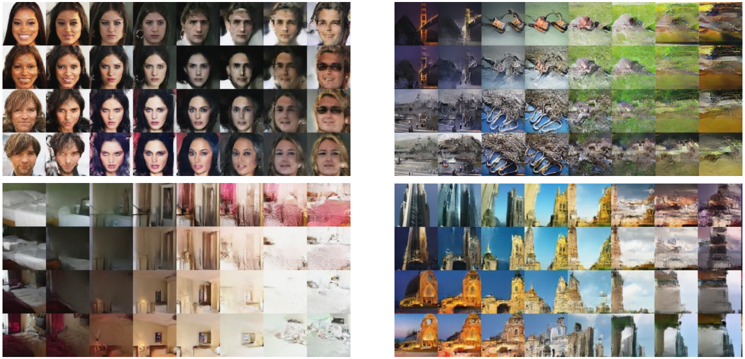


Figure 6: Manifold generated from four examples in the dataset. Clockwise from top left: CelebA, Imagenet (64×64), LSUN (tower), LSUN (bedroom).

(Dinh et al., 2016)

The End

References

L. Dinh, D. Krueger, and Y. Bengio. **NICE: non-linear independent components estimation**. CoRR, abs/1410.8516, 2014.

L. Dinh, J. Sohl-Dickstein, and S. Bengio. **Density estimation using real NVP**. CoRR, abs/1605.08803, 2016.

G. Papamakarios, E. Nalisnick, D. Rezende, S. Mohamed, and B. Lakshminarayanan. **Normalizing flows for probabilistic modeling and inference**. CoRR, abs/1912.02762, 2019.