9.4. Non-volume preserving networks

François Fleuret
https://fleuret.org/ammi-2018/
Sat Nov 10 16:47:12 UTC 2018
A standard result of probability theory is that if $f$ is continuous, invertible and [almost everywhere] differentiable, then

$$\forall x, \quad p_f^{-1}(Z)(x) = p_Z(f(x)) |J_f(x)| .$$
A standard result of probability theory is that if $f$ is continuous, invertible and [almost everywhere] differentiable, then

$$\forall x, \ p^{f^{-1}(Z)}(x) = p^Z(f(x)) |J_f(x)| .$$
A standard result of probability theory is that if $f$ is continuous, invertible and [almost everywhere] differentiable, then

$$\forall x, \ p_f^{-1}(Z)(x) = p_Z(f(x)) |J_f(x)| .$$
A standard result of probability theory is that if \( f \) is continuous, invertible and [almost everywhere] differentiable, then

\[
\forall x, \quad p^{f^{-1}}(Z)(x) = p^Z(f(x)) |J_f(x)|.
\]
From this equality, if $f$ is a parametric function such that we can compute [and differentiate]

$$p^Z(f(x))$$

and

$$|J_f(x)|$$

then, we can make the distribution of $f^{-1}(Z)$ fits the data by optimizing

$$\sum_n \log p^{f^{-1}(Z)}(x_n)$$
From this equality, if $f$ is a parametric function such that we can compute [and differentiate]

$$p^Z(f(x))$$

and

$$|J_f(x)|$$

then, we can make the distribution of $f^{-1}(Z)$ fits the data by optimizing

$$\sum_n \log p^{f^{-1}(Z)}(x_n) = \sum_n \log \left( p^Z(f(x_n)) |J_f(x_n)| \right).$$
From this equality, if $f$ is a parametric function such that we can compute [and differentiate]

$$p^Z(f(x))$$

and

$$|J_f(x)|$$

then, we can make the distribution of $f^{-1}(Z)$ fits the data by optimizing

$$\sum_n \log p^{f^{-1}(Z)}(x_n) = \sum_n \log \left(p^Z(f(x_n)) |J_f(x_n)| \right).$$

If we are able to do so, then we can synthesize a new $X$ by sampling $Z \sim \mathcal{N}(0, 1)$ and computing $f^{-1}(Z)$. 
If $Z \sim \mathcal{N}(0, I)$,

$$\log p^Z (f(x_n)) = -\frac{1}{2} (\|f(x_n)\|^2 + d \log 2\pi).$$
If $Z \sim \mathcal{N}(0, I)$,

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

And remember that if $f$ is is a composition of functions

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

we have

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

so

$$\log |J_f(x)| = \sum_{k=1}^{K} \log \left| J_{f^{(k)}} \left( f^{(k-1)} \circ \ldots \circ f^{(1)}(x) \right) \right|. $$
If $f^{(k)}$ are standard layers computing $f^{-1}(z)$ is impossible, and computing $|J_f(x)|$ is intractable.
If \( f^{(k)} \) are 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 an example of a Normalizing flow (Rezende and Mohamed, 2015).
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

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

we define a fully connected coupling layer as the transformation

$$c : \mathbb{R}^d \rightarrow \mathbb{R}^d \times \mapsto b \odot x + (1 - b) \odot (x \odot \exp(s(b \odot x)) + t(b \odot x))$$

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

$$c : \mathbb{R}^d \to \mathbb{R}^d$$
$$x \mapsto b \odot x + (1 - b) \odot \left( x \odot \exp(s(b \odot x)) + t(b \odot x) \right)$$

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

\[ c(x) = b \odot x + (1 - b) \odot \left( x \odot \exp(s(b \odot x)) + t(b \odot x) \right) \]

can be understood as: forward \( b \odot x \) unchanged,
The expression

\[
c(x) = b \odot x + (1 - b) \odot \left( x \odot \exp(s(b \odot x)) + t(b \odot x) \right)
\]

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 $y = c(x)$

$$x = b \odot y + (1 - b) \odot \left( y - t(b \odot y) \right) \odot \exp(-s(b \odot y)).$$
The consequence is that $c$ is invertible, and if $y = c(x)$

$$x = b \odot y + (1 - b) \odot \left( y - t(b \odot y) \right) \odot \exp(-s(b \odot y)).$$
The second property of this mapping is the simplicity of its Jacobian determinant.
The second property of this mapping is the simplicity of its Jacobian determinant. Since

\[ c_i(x) = b_i \odot x_i + (1 - b_i) \odot \left( x_i \odot \exp(s_i(b \odot x)) + t_i(b \odot x) \right) \]

we have, \( \forall i, j, x, \)

\[ b_i = 1 \implies c_i(x) = x_i \]

\[ \implies \frac{\partial c_i}{\partial x_j} = 1\{i=j\} \]
The second property of this mapping is the simplicity of its Jacobian determinant. Since

\[ c_i(x) = b_i \odot x_i + (1 - b_i) \odot \left( x_i \odot \exp(s_i(b \odot x)) + t_i(b \odot x) \right) \]

we have, \( \forall i, j, x, \)

\[ b_i = 1 \implies c_i(x) = x_i \]

\[ \implies \frac{\partial c_i}{\partial x_j} = 1_{\{i=j\}} \]

and

\[ b_i = 0 \implies c_i(x) = x_i \exp(s_i(b \odot x)) + t_i(b \odot x) \]

Hence

\[ \frac{\partial c_i}{\partial x_j} \text{ can be non-zero only if } i = j, \text{ or } (1 - b_i) b_j = 1. \]
The second property of this mapping is the simplicity of its Jacobian determinant. Since

\[ c_i(x) = b_i \odot x_i + (1 - b_i) \odot \left( x_i \odot \exp(s_i(b \odot x)) + t_i(b \odot x) \right) \]

we have, \( \forall i, j, x, \)

\[ b_i = 1 \Rightarrow c_i(x) = x_i \]

\[ \Rightarrow \frac{\partial c_i}{\partial x_j} = 1_{\{i=j\}} \]

and

\[ b_i = 0 \Rightarrow c_i(x) = x_i \exp(s_i(b \odot x)) + t_i(b \odot x) \]

\[ \Rightarrow \frac{\partial c_i}{\partial x_j} = \left( 1_{\{i=j\}} + x_i \frac{\partial s_i(b \odot x)}{\partial x_j} \right) \exp(s_i(b \odot x)) + \frac{\partial t_i(b \odot x)}{\partial x_j} \]

\[ \Rightarrow \frac{\partial c_i}{\partial x_j} = 1_{\{i=j\}} \exp(s_i(b \odot x)) + b_j \left( \begin{array}{c} \ldots \end{array} \right). \]
The second property of this mapping is the simplicity of its Jacobian determinant. Since

\[ c_i(x) = b_i \odot x_i + (1 - b_i) \odot \left( x_i \odot \exp(s_i(b \odot x)) + t_i(b \odot x) \right) \]

we have, \( \forall i, j, x, \)

\[ b_i = 1 \Rightarrow c_i(x) = x_i \]
\[ \Rightarrow \frac{\partial c_i}{\partial x_j} = 1_{\{i=j\}} \]

and

\[ b_i = 0 \Rightarrow c_i(x) = x_i \exp(s_i(b \odot x)) + t_i(b \odot x) \]
\[ \Rightarrow \frac{\partial c_i}{\partial x_j} = \left( 1_{\{i=j\}} + x_i \frac{\partial s_i(b \odot x)}{\partial x_j} \right) \exp(s_i(b \odot x)) + \frac{\partial t_i(b \odot x)}{\partial x_j} \]
\[ = 1_{\{i=j\}} \exp(s_i(b \odot x)) + b_j \left( \ldots \right). \]

Hence \( \frac{\partial c_i}{\partial x_j} \) can be non-zero only if \( i = j \), or \( (1 - b_i)b_j = 1 \).
If we re-order both the rows and columns of the Jacobian to put first the non-zeros entries of $b$, and then the zeros, it becomes lower triangular

$$J_c(x) = \begin{pmatrix} 1 & \cdots & \cdots & \cdots & \cdots & (0) \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots \\ (\neq 0) & \ddots & \ddots & \ddots & \ddots & \ddots \end{pmatrix} \begin{pmatrix} \exp(s_k(x \odot b)) \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \exp(s_{k'}(x \odot b)) \end{pmatrix}$$

its determinant remains unchanged, and we have

$$\log |J_{f(k)}(x)| = \sum_{i: b_i = 0} s_i(x \odot b)$$

$$= \sum_i ((1 - b) \odot s(x \odot b))_i.$$
dim = 6

x = torch.empty(1, dim).normal_().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 * s(b * x).exp() + t(b * x))

j = torch.cat([torch.autograd.grad(c_k, x, retain_graph=True)[0] for c_k in c[0]])
print(j)
dim = 6

x = torch.empty(1, dim).normal_().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 * s(b * x).exp() + t(b * x))

j = torch.cat([torch.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],
        [ 0.6073,  0.3393,  0.0282,  0.9897,  0.0000,  0.0000],
        [-0.3223, -0.0081,  0.1224,  0.0000,  1.4373,  0.0000],
        [ 0.0301,  0.3185,  0.0665,  0.0000,  0.0000,  1.0295]]])
To recap, with $f^{(k)}$, $k = 1, \ldots, K$ coupling layers,

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

and $x_n^{(0)} = x_n$ and $x_n^{(k)} = f^{(k)}(x_n^{(k-1)})$. 

And to sample we just need to generate $Z \sim \mathcal{N}(0, I)$ and compute $f^{-1}(Z)$. 

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

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

and $x_n^{(0)} = x_n$ and $x_n^{(k)} = f^{(k)}(x_n^{(k-1)})$, we train by maximizing

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

with

$$\log \left| J_{f^{(k)}}(x) \right| = \sum_i \left( (1 - b^{(k)}) \circ s^{(k)}(x \circ b^{(k)}) \right)_i.$$
To recap, with $f^{(k)}$, $k = 1, \ldots, K$ coupling layers,

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

and $x_n^{(0)} = x_n$ and $x_n^{(k)} = f^{(k)} (x_n^{(k-1)})$, we train by maximizing

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

with

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

And to sample we just need to generate $Z \sim \mathcal{N}(0, I)$ and compute $f^{-1}(Z)$. 
A coupling layer can be implemented with

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

    def forward(self, x_and_logdetjac):
        x, logdetjac = x_and_logdetjac
        s, t = self.map_s(self.b * x), self.map_t(self.b * x)
        logdetjac += ((1 - self.b) * s).sum(1)
        y = self.b * x + (1 - self.b) * (torch.exp(s) * x + t)
        return (y, logdetjac)

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

The **forward** here computes both the image of \( x \) and the update on the accumulated determinant of the Jacobian, \( i.e. \)

\[
(x, u) \mapsto (f(x), u + |J_f(x)|).
\]
We can then define a complete network with one-hidden layer tanh MLPs for the \( s \) and \( t \) mappings

```python
class NVPNet(nn.Module):
    def __init__(self, dim, hidden_dim, depth):
        super(NVPNet, self).__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))

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

    def invert(self, x):
        for m in reversed(self.layers):
            x = m.invert(x)
        return x
```
And the log-proba of individual samples of a batch

```python
def LogProba(x_and_logdetjac):
    (x, logdetjac) = x_and_logdetjac
    log_p = logdetjac - 0.5 * x.pow(2).add(math.log(2 * math.pi)).sum(1)
    return log_p
```
Training is achieved by maximizing the mean log-proba

```python
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):
    acc_loss = 0

    for b in range(0, nb_train_samples, batch_size):
        output = model((input[b:b+batch_size], 0))
        loss = - LogProba(output).mean()
        acc_loss = acc_loss + loss.item()
        model.zero_grad()
        loss.backward()
        optimizer.step()
```

Finally, we can sample according to $p_{f-1}(Z)$ with $z = \text{torch.empty(nb_train_samples, 2).normal()}$ $x = \text{model.invert(z)}$
Training is achieved by maximizing the mean log-proba

```python
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):
    acc_loss = 0

    for b in range(0, nb_train_samples, batch_size):
        output = model((input[b:b+batch_size], 0))
        loss = - LogProba(output).mean()
        acc_loss = acc_loss + loss.item()
        model.zero_grad()
        loss.backward()
        optimizer.step()
```

Finally, we can sample according to $p_{f^{-1}}(Z)$ with

```python
z = torch.empty(nb_train_samples, 2).normal_()
x = model.invert(z)
```
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).

We also illustrate the smooth semantically consistent meaning of our latent variables. In the latent space, we define a manifold based on four validation examples $z^{(1)}, z^{(2)}, z^{(3)}, z^{(4)}$, and parametrized by two parameters $\phi$ and $\phi'$ by:

$$z = \cos(\phi) \left( \cos(\phi') z^{(1)} + \sin(\phi') z^{(2)} \right) + \sin(\phi) \left( \cos(\phi') z^{(3)} + \sin(\phi') z^{(4)} \right).$$

(19)

We project the resulting manifold back into the data space by computing $g(z)$. Results are shown in Figure 6. We observe that the model seems to have organized the latent space with a notion of meaning that goes well beyond pixel space interpolation. More visualization are shown in the Appendix.

5 Discussion and conclusion

In this paper, we have defined a class of invertible functions with tractable Jacobian determinant, enabling exact and tractable log-likelihood evaluation, inference, and sampling. We have shown that this class of generative model achieves competitive performances, both in terms of sample quality and log-likelihood. Many avenues exist to further improve the functional form of the transformations, for instance by exploiting the latest advances in dilated convolutions [69] and residual networks architectures [60].

This paper presented a technique bridging the gap between auto-regressive models, variational autoencoders, and generative adversarial networks. Like auto-regressive models, it allows tractable and exact log-likelihood evaluation for training. It allows however a much more flexible functional form, similar to that in the generative model of variational autoencoders. This allows for fast and exact sampling from the model distribution. Like GANs, and unlike variational autoencoders, our technique does not require the use of a fixed form reconstruction cost, and instead defines a cost in terms of higher level features, generating sharper images. Finally, unlike both variational autoencoders and GANs, our technique is able to learn a semantically meaningful latent space which is as high dimensional as the input space. This may make the algorithm particularly well suited to semi-supervised learning tasks, as we hope to explore in future work.

(Dinh et al., 2016)
The end
References

