Deep learning

10.3. Non-volume preserving networks

François Fleuret
https://fleuret.org/dlc/
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.

**Notes**

Another strategy to model high dimension data densities is to train a model to transform the data density into a fixed "normalized" one. The resulting mapping is called a normalizing flow.

Those methods rely on a standard result of probability theory. If $f$ is continuous, invertible, and [almost everywhere] differentiable, and $X = f^{-1}(Z)$, then the density $\mu_X$ of $X$ at a certain point $x$ is equal to the density of $Z$ at the image point $f(x)$ multiplied by the determinant of $f$’s Jacobian at $x$ that reflects the local expansion of the space.

This formula shows that the density of $X$ is proportional to how likely it was for $Z$ to be $f(x)$, and how much $f$ is "stretching" locally the space there.

Here we take $Z$ uniform on $[1, 3]$, and $f$ monotonically increasing with three linear pieces, the first and last ones with a slope of 1, and the middle one with a slope lesser than 1. When the slope is 1, we have the exact same density in the two domains. When the slope is smaller, the corresponding density from $Z$ is expanded on a broader domain for $X$. 
Since
\[ \mu_X(x) = \mu_Z(f(x)) |J_f(x)|, \]
if \( f \) is a parametric function such that we can compute [and differentiate] \( \mu_Z(f(x)) \) and \( |J_f(x)| \),
given \( x_1, \ldots, x_N \) i.i.d \( \sim \mu \), we can make \( \mu_X \) 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 \( Z \sim \mathcal{N}(0, I) \),
\[ \log \mu_Z(f(x_n)) = -\frac{1}{2} (\|f(x_n)\|^2 + d \log 2\pi). \]

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 $\alpha, \xi_1, \ldots, \xi_Q$.

Notes

To illustrate this on a simple example, we consider an increasing piece-wise affine model:

- $f(x) = \alpha$ for $x \leq x_{\text{min}}$,
- $[x_{\text{min}}, x_{\text{max}}]$ is split in $N$ intervals in each of which $f$ is affine and increases by $e^{\xi_q}$,
- $f$ is constant for $x \geq x_{\text{max}}$, and hence equal to $\alpha + \sum_q e^{\xi_q}$.

The quantities $N$, $x_{\text{min}}$, and $x_{\text{max}}$ will be fixed, while $\alpha$ and the $\xi_q$s are the parameters of $f$ that will be optimized during training.
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

Notes

We initialize the model so that $f$ behaves like the identity on $[x_min, x_max]$.

forward operates on a batch of values. The variable $y$ holds $f$’s values at the changes of slope, $n$ the index of the interval for each input value, and $a$ the relative position in the interval.
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 \((f'(x_1), \ldots, f'(x_N))\) with autograd.

With

\[ \Phi(x_1, \ldots, x_N) = f(x_1) + \cdots + f(x_N) \]

we have

\[ \nabla \Phi(x_1, \ldots, x_n) = (f'(x_1), \ldots, f'(x_N)) \].
\[ \mathcal{L}(f) = \frac{1}{N} \sum_n \frac{1}{2} (f(x_n)^2 + \log 2\pi) - \log f'(x_n). \]

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()

Notes

We need the derivative of \( f \) w.r.t. its input, so specify `requires_grad_()`, and compute the gradient of `output.sum()` to get \( (f'(x_1), \ldots, f'(x_N)) \).

Since PyTorch by default allows to use the autograd graph only once, we specify `retain_graph=True` when computing the \( f' \) to be able to use it a second time to compute the gradient of the loss. And since the loss depends on the \( f' \), we also state `create_graph = True`. 
Target distribution $\mu$

Resulting mapping $\hat{f}$

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

**Notes**

As a toy example, we use a mixture of two Gaussian distributions as the data distribution (top left), and train a PiecewiseLinear model with 1000 intervals to map this distribution normal distribution of zero mean and unit variance.

The empirical distribution (red curve, bottom left) is estimated by taking regularly spaced points on the interval, and computing for each of these $x$ the log of the normal density at $f(x)$, and $f'(x)$.

The bottom right image shows how $f$ is contracting or expanding the space to properly fit the normal distribution. The mapping from $x$ to $f(x)$ is depicted by the red lines. In particular, the “hole” between the two Gaussians in the mixture gets contracted to fit the normal part, which the middle of the Gaussians are expanded. The tails are just shifted.
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 $Z \sim \mathcal{N}(0, I)$ 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).

Notes

In the toy example of the first part, computing $|J_f|$ could be done with autograd since we were in 1d. This does not scale to real-world high dimension signals.
Remember that if \( f \) is a composition

\[
f = f^{(K)} \circ \cdots \circ f^{(1)}
\]

we have

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

hence

\[
\log |J_f(x)| = \sum_{k=1}^{K} \log \left| J_{f^{(k)}} \left( f^{(k-1)} \circ \cdots \circ f^{(1)}(x) \right) \right|.
\]
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
\[
s : \mathbb{R}^d \rightarrow \mathbb{R}^d \\
t : \mathbb{R}^d \rightarrow \mathbb{R}^d,
\]
we define a [fully connected] coupling layer as the transformation
\[
c : \mathbb{R}^d \rightarrow \mathbb{R}^d \\
x \mapsto b \odot x + (1 - b) \odot \left( x \odot \exp(s(b \odot x)) + t(b \odot x) \right)
\]
where $\exp$ is component-wise, and $\odot$ is the Hadamard component-wise product.

For clarity in what follows, $b$ has all 1s first, follows by 0s, but this is not required.
\[
b = (1, 1, \ldots, 1, 0, 0, \ldots, 0)_{\Delta d - \Delta d}
\]

Notes

The quantities $t$ and $s$ stand respectively for translation and scale.

Such a “coupling layer” keeps the components for which the corresponding $b_i$ is 1 unchanged, modifies the other components in an invertible way that only depends on the unchanged ones.
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 $y = c(x)$

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

\[
J_c(x) = \begin{pmatrix}
1 & & & & (0) \\
& \ddots & & & \\
& & 1 & & \\
& & & \exp(s_{\Delta+1}(x \odot b)) & \\
(\neq 0) & & & & \ddots \\
& & & & \exp(s_d(x \odot b))
\end{pmatrix}
\]

and we have

\[
\log |J_c(x)| = \sum_{i; b_i = 0} s_i(x \odot b)
\]

\[
= \sum_i ((1 - b) \odot s (x \odot b))_i.
\]

Notes

Remember that for the sake of simplicity we make the assumption that all the 1s in \( b \) appear consecutively first.

For any pair \( i, j \) such that \( b_i = 1, b_j = 1 \), we have

\[
\frac{\partial c_i}{\partial x_j} = \frac{\partial x_i}{\partial x_j} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}
\]

which gives us a diagonal of ones in the top left part of the Jacobian.

For any pair \( i, j \) such that \( b_i = 0, b_j = 0 \), we have

\[
\frac{\partial c_i}{\partial x_j} = \frac{\partial x_i}{\partial x_j} \left( \begin{array}{c} x_i e^{s_i(b \odot x)} + t_i(b \odot x) \\ \text{constant w.r.t. } x_j \end{array} \right)
\]

\[
= \begin{cases} \exp(s_i(b \odot x)) & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}
\]

so the bottom right part of the Jacobian is a diagonal with terms \( \exp(s_i(x \odot b)) \).
```python
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 * 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 $f^{(k)}$, $k = 1, \ldots, K$ coupling layers,

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

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

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

with

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

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

---

**Notes**

Since all coupling layers are invertible, $f^{-1}$ can be computed. So at generation time, we draw a sample $Z \sim \mathcal{N}(0, I)$ and compute $X = f^{-1}(Z)$.
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.

\[
(x, u) \mapsto (f(x), u + |J_f(x)|)
\]
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().__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
```

**Notes**

Masks \( b \) are made in such a way that \( b^{(2n)} \) are generated at random, while \( b^{(2n+1)} = 1 - b^{(2n)} \).

It assures that all the components of the input are changed. This is one out of many strategies that can be used to ensure that the components are modified.
And the log-proba of individual samples of a batch

```python
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 = NVPPNet(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 $\mu_X$ with

```
z = torch.empty(nb_generated_samples, 2).normal_()
x = model.invert(z)
```
Notes

We test this model on 2d synthetic distributions. On each graph:

- the blue dots are sampled training points,
- the blue lines are circles deformed by the “true” \( f \),
- the red lines are circles deformed by the trained model,
- The red dots are points sampled according to the trained model.

The true density at the bottom-right is discontinuous. It is obtained by sampling according to a normal, and adding 1 if the \( x \) coordinate is positive, and \(-1\) otherwise. This is challenging for the model which is continuous.
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.](image)

(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

- **× 2 stages**
  - **× 3** checkerboard coupling layers,
  - a squeezing layer,
  - **× 3** channel coupling layers,
  - a factor-out layer.

- **× 1 stage**
  - **× 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,

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

We project the resulting manifold back into the data space by computing $g(z)$. Results are shown 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)
References

