

Deep learning

10.1. Auto-regression

François Fleuret

<https://fleuret.org/dlc/>



UNIVERSITÉ
DE GENÈVE

Auto-regression methods model components of a signal serially, **each one conditionally to the ones already modeled**.

They rely on the chain rule from probability theory: given X_1, \dots, X_T random variables, we have

$$\forall x_1, \dots, x_T, P(X_1 = x_1, \dots, X_T = x_T) = \\ P(X_1 = x_1) P(X_2 = x_2 \mid X_1 = x_1) \dots P(X_T = x_T \mid X_1 = x_1, \dots, X_{T-1} = x_{T-1}).$$

Deep neural networks are a fitting class of models for such conditional densities when dealing with large dimension signal (Larochelle and Murray, 2011).

Given a sequence of random variables X_1, \dots, X_T on \mathbb{R} , we can represent a conditioning event of the form

$$X_{t(1)} = x_1, \dots, X_{t(N)} = x_N$$

with two tensors of dimension T : the first a Boolean mask stating which variables are conditioned, and the second the actual conditioning values.

E.g., with $T = 5$

Event	Mask tensor	Value tensor
$\{X_2 = 3\}$	$[0, 1, 0, 0, 0]$	$[0, 3, 0, 0, 0]$
$\{X_1 = 1, X_2 = 2, X_3 = 3, X_4 = 4, X_5 = 5\}$	$[1, 1, 1, 1, 1]$	$[1, 2, 3, 4, 5]$
$\{X_5 = 50, X_2 = 20\}$	$[0, 1, 0, 0, 1]$	$[0, 20, 0, 0, 50]$

In what follows, we will consider only finite distributions over C real values.

Hence we can model a conditional distribution with a mapping that maps a pair mask / known values to a distribution for the next value of the sequence:

$$f : \{0, 1\}^Q \times \mathbb{R}^Q \rightarrow \mathbb{R}^C,$$

where the C output values can be either probabilities, or as we will prefer, logits.

This can be generalized beyond categorical distributions by mapping to parameters of any distribution.

Given such a model and a sampling procedure `sample`, the generative process for a full sequence is

$$x_1 \leftarrow \text{sample}(f(\{\}))$$

$$x_2 \leftarrow \text{sample}(f(\{X_1 = x_1\}))$$

$$x_3 \leftarrow \text{sample}(f(\{X_1 = x_1, X_2 = x_2\}))$$

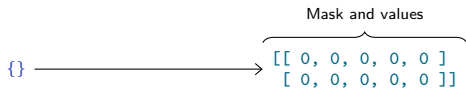
...

$$x_T \leftarrow \text{sample}(f(\{X_1 = x_1, X_2 = x_2, \dots, X_{T-1} = x_{T-1}\}))$$

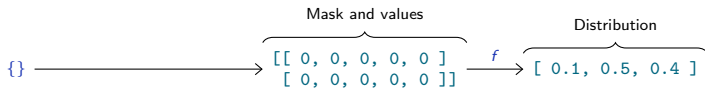
For instance, with $C = 3$ and $T = 5$, we could have:

$\{\}$

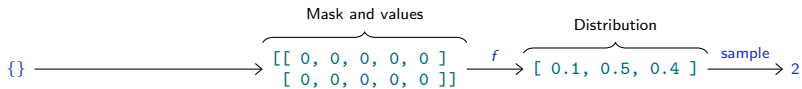
For instance, with $C = 3$ and $T = 5$, we could have:



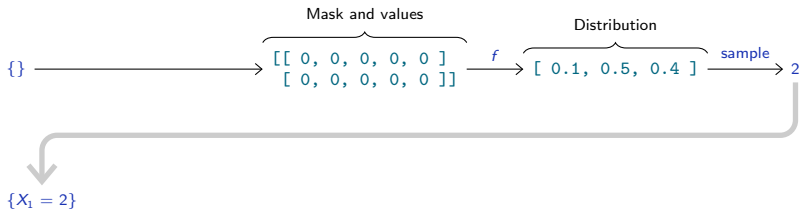
For instance, with $C = 3$ and $T = 5$, we could have:



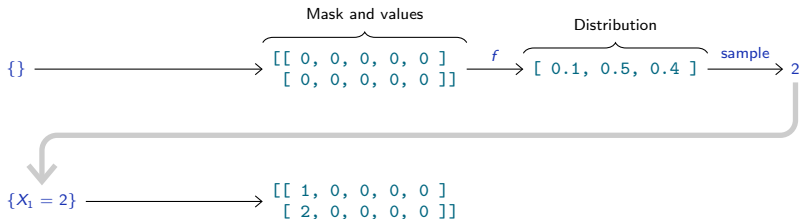
For instance, with $C = 3$ and $T = 5$, we could have:



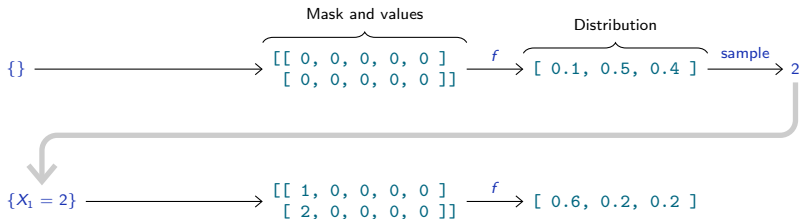
For instance, with $C = 3$ and $T = 5$, we could have:



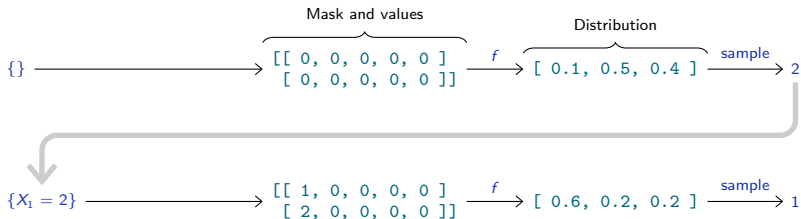
For instance, with $C = 3$ and $T = 5$, we could have:



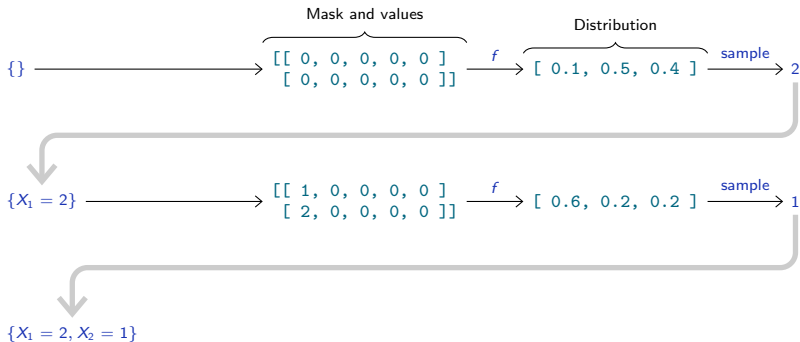
For instance, with $C = 3$ and $T = 5$, we could have:



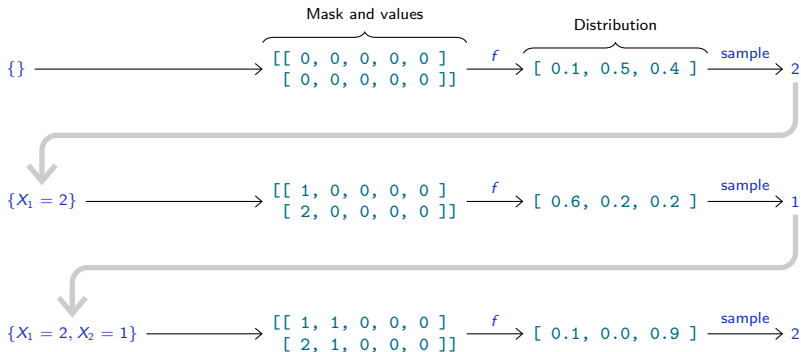
For instance, with $C = 3$ and $T = 5$, we could have:



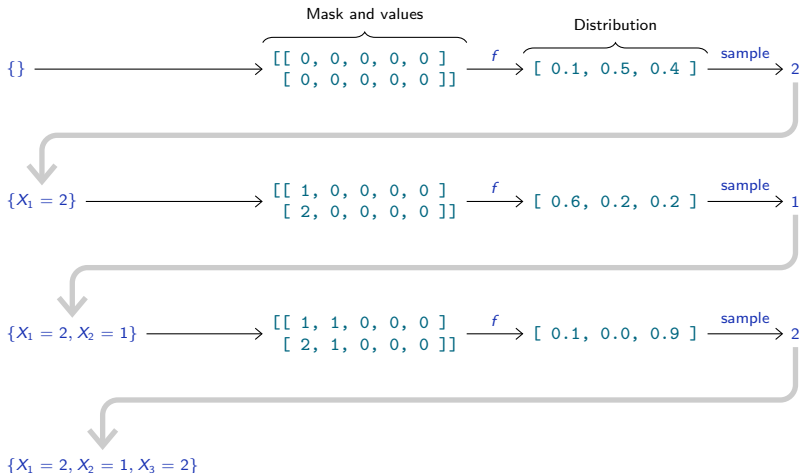
For instance, with $C = 3$ and $T = 5$, we could have:



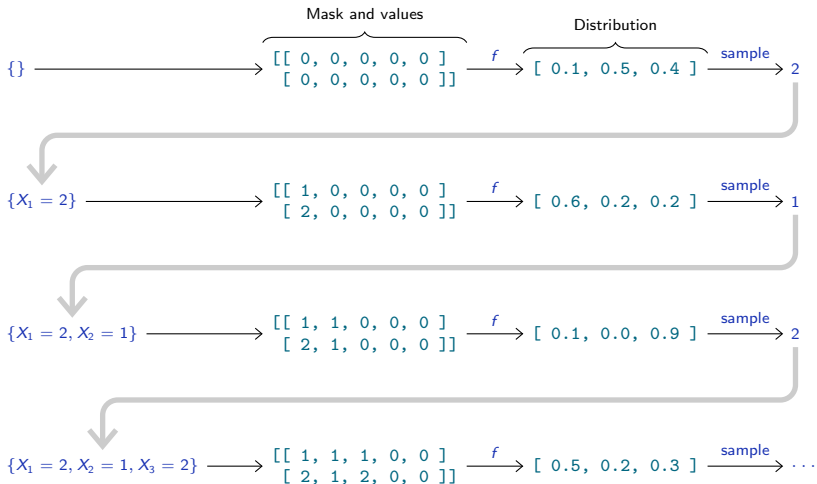
For instance, with $C = 3$ and $T = 5$, we could have:



For instance, with $C = 3$ and $T = 5$, we could have:



For instance, with $C = 3$ and $T = 5$, we could have:



The package `torch.distributions` provides the necessary tools to sample from a variety of distributions.

```
>>> l = torch.tensor([ log(0.8), log(0.1), log(0.1) ])
>>> dist = torch.distributions.categorical.Categorical(logits = l)
>>> s = dist.sample((10000,))
>>> (s.view(-1, 1) == torch.arange(3).view(1, -1)).float().mean(0)
tensor([0.8037, 0.0988, 0.0975])
```

Sampling can also be done in batch

```
>>> l = torch.tensor([[ log(0.90), log(0.10) ],
...                   [ log(0.50), log(0.50) ],
...                   [ log(0.25), log(0.75) ],
...                   [ log(0.01), log(0.99) ]])
>>> dist = torch.distributions.categorical.Categorical(logits = l)
>>> dist.sample((8,))
tensor([[0, 1, 1, 1],
        [0, 1, 1, 1],
        [0, 0, 1, 1],
        [0, 1, 0, 1],
        [1, 0, 1, 1],
        [0, 1, 1, 1],
        [0, 1, 1, 1],
        [0, 0, 1, 1]])
```


With a finite distribution and the output values interpreted as logits, training consists of maximizing the likelihood of the training samples, hence minimizing

$$\begin{aligned}\mathcal{L}(f) &= - \sum_n \sum_t \log \hat{p}(X_t = x_{n,t} \mid X_1 = x_{n,1}, \dots, X_{t-1} = x_{n,t-1}) \\ &= \sum_n \sum_t \ell \left(f((1, \dots, 1, 0, \dots, 0), (x_{n,1}, \dots, x_{n,t-1}, 0, \dots, 0)), x_{n,t} \right)\end{aligned}$$

where ℓ is the cross-entropy.

In practice, for each batch, we sample a position to predict for each sample at random, from which we build the masks, conditioning values, and target values.


Training Sequences



```
[[ 3, 1, 8, 1, 0, 3 ],  
 [ 2, 3, 0, 9, 6, 5 ],  
 [ 7, 1, 5, 7, 3, 1 ],  
 [ 6, 0, 2, 3, 1, 9 ]]
```

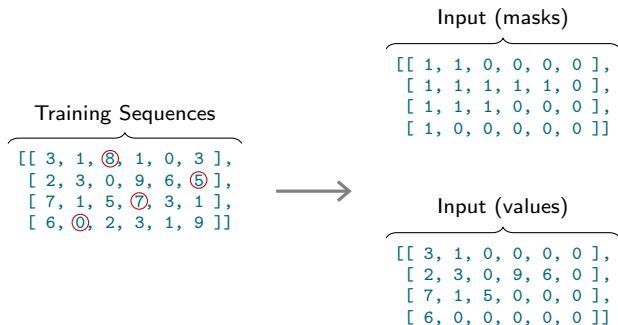
In practice, for each batch, we sample a position to predict for each sample at random, from which we build the masks, conditioning values, and target values.

Training Sequences

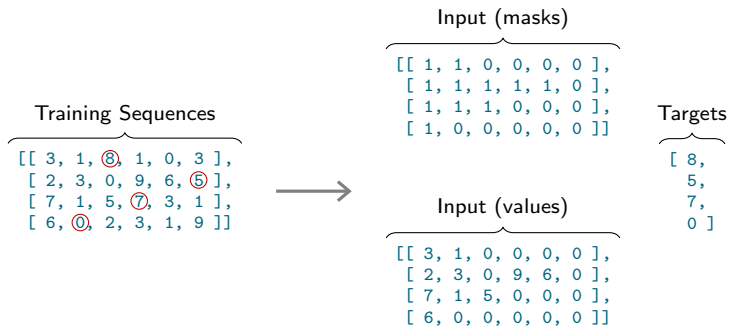


$\begin{bmatrix} [3, 1, \textcircled{8}, 1, 0, 3], \\ [2, 3, 0, 9, 6, \textcircled{5}], \\ [7, 1, 5, \textcircled{7}, 3, 1], \\ [6, \textcircled{0}, 2, 3, 1, 9] \end{bmatrix}$

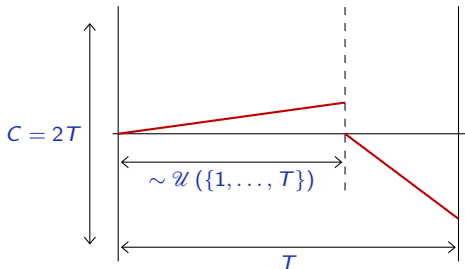
In practice, for each batch, we sample a position to predict for each sample at random, from which we build the masks, conditioning values, and target values.



In practice, for each batch, we sample a position to predict for each sample at random, from which we build the masks, conditioning values, and target values.

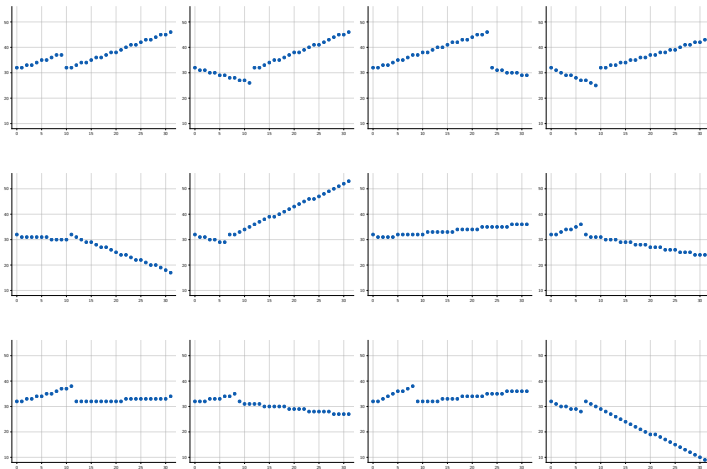


Consider a toy problem, where sequences from $\{1, \dots, C\}^T$ are split in two at a random position, and are linear in both parts, with slopes $\sim \mathcal{U}([-1, 1])$.



Values are re-centered and discretized into $2T$ values.

Some train sequences



Model

```
class Net(nn.Module):
    def __init__(self, nb_values):
        super().__init__()

        self.features = nn.Sequential(
            nn.Conv1d(2, 32, kernel_size = 5),
            nn.ReLU(),
            nn.MaxPool1d(2),
            nn.Conv1d(32, 64, kernel_size = 5),
            nn.ReLU(),
            nn.MaxPool1d(2),
            nn.ReLU(),
        )

        self.fc = nn.Sequential(
            nn.Linear(320, 200),
            nn.ReLU(),
            nn.Linear(200, nb_values)
        )

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x
```

Training loop

```
for sequences in train_sequences.split(batch_size):
    nb = sequences.size(0)

    # Select a random index in each sequence, this is our targets
    idx = torch.randint(len, (nb, 1), device = sequences.device)
    targets = sequences.gather(1, idx).view(-1)

    # Create masks and values accordingly
    tics = torch.arange(len, device = sequences.device).view(1, -1).expand(nb, -1)
    masks = (tics < idx.expand(-1, len)).float()
    values = (sequences.float() - mean) / std * masks

    # Make masks and values one-channel and concatenate them along
    # channels to make the input
    input = torch.cat((masks.unsqueeze(1), values.unsqueeze(1)), 1)

    # Compute the loss and make the gradient step
    output = model(input)
    loss = cross_entropy(output, targets)

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

Synthesis

```
nb = 25
generated = torch.zeros(nb, len, device = device, dtype = torch.int64)
tics = torch.arange(len, device = device).view(1, -1).expand(nb, -1)

for t in range(len):
    masks = (tics < t).float()
    values = (generated.float() - mean) / std * masks
    input = torch.cat((masks.unsqueeze(1), values.unsqueeze(1)), 1)
    output = model(input)
    dist = torch.distributions.categorical.Categorical(logits = output)
    generated[:, t] = dist.sample()
```

Some generated sequences

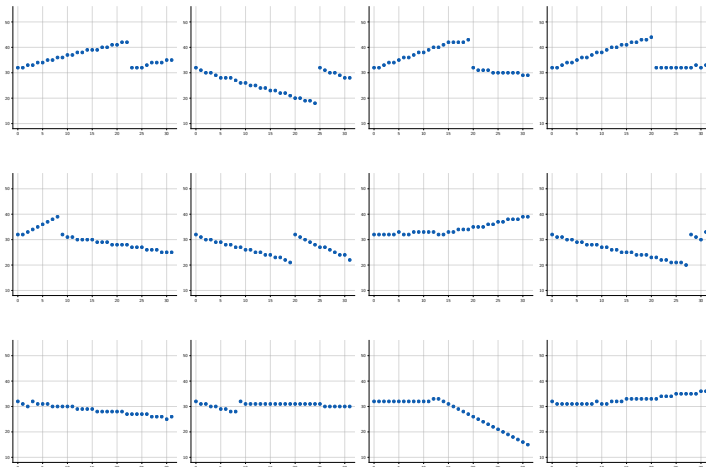
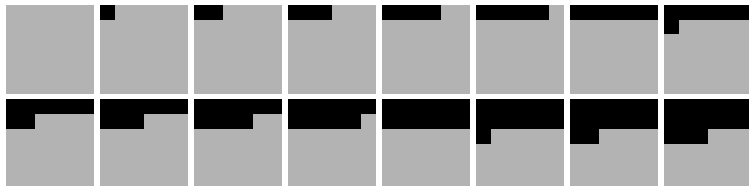


Image auto-regression

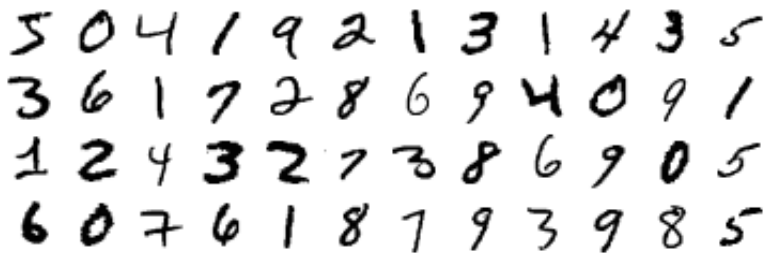
The exact same auto-regressive approach generalizes to any tensor shape, as long as a visiting order of the coefficients is provided.

For instance, for images, we can visit pixels in the “raster scan order” corresponding to the standard mapping in memory, top-to-bottom, left-to-right.

```
image_masks = torch.empty(16, 1, 6, 6)
for k in range(image_masks.size(0)):
    sequence_mask = torch.arange(1 * 6 * 6) < k
    image_masks[k] = sequence_mask.float().view(1, 6, 6)
```



Some of the MNIST train images



We define two functions to serialize the image tensors into sequences

```
def seq2tensor(s):  
    return s.reshape(-1, 1, 28, 28)  
  
def tensor2seq(s):  
    return s.reshape(-1, 28 * 28)
```

Model

```
class LeNetMNIST(nn.Module):
    def __init__(self, nb_classes):
        super().__init__()

        self.features = nn.Sequential(
            nn.Conv2d(1, 32, kernel_size = 3),
            nn.MaxPool2d(kernel_size = 2),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size = 5),
            nn.ReLU(),
        )

        self.fc = nn.Sequential(
            nn.Linear(64 * 81, 512),
            nn.ReLU(),
            nn.Linear(512, nb_classes)
        )

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x
```

Training loop

```
for data in train_input.split(args.batch_size):
    # Make 1d sequences from the images
    sequences = tensor2seq(data)
    nb, len = sequences.size(0), sequences.size(1)

    # Select a random index in each sequence, this is our targets
    idx = torch.randint(len, (nb, 1), device = device)
    targets = sequences.gather(1, idx).view(-1)

    # Create masks and values accordingly
    tics = torch.arange(len, device = device).view(1, -1).expand(nb, -1)
    masks = seq2tensor((tics < idx.expand(-1, len)).float())
    values = (data.float() - mu) / std * masks

    # Make the input, set the mask and values as two channels
    input = torch.cat((masks, values), 1)

    # Compute the loss and make the gradient step
    output = model(input)
    loss = cross_entropy(output, targets)

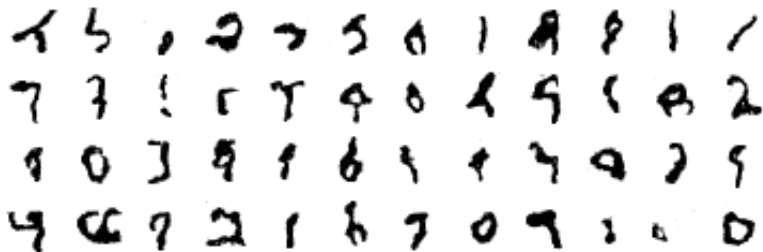
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Synthesis

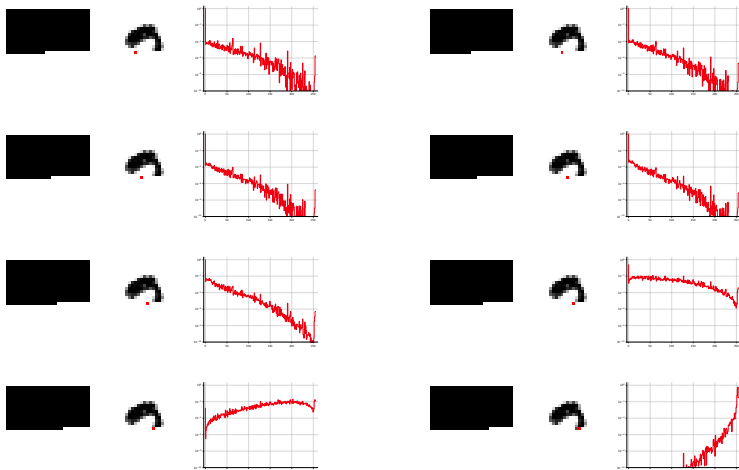
```
nb = 48
generated = torch.zeros((nb,) + train_input.shape[1:],
                        device = device, dtype = torch.int64)
sequences = tensor2seq(generated)
tics = torch.arange(sequences.size(1), device = device).view(1, -1).expand(nb, -1)

for t in range(sequences.size(1)):
    masks = seq2tensor((tics < t).float())
    values = (seq2tensor(sequences).float() - mu) / std * masks
    input = torch.cat((masks, values), 1)
    output = model(input)
    dist = torch.distributions.categorical.Categorical(logits = output)
    sequences[:, t] = dist.sample()
```

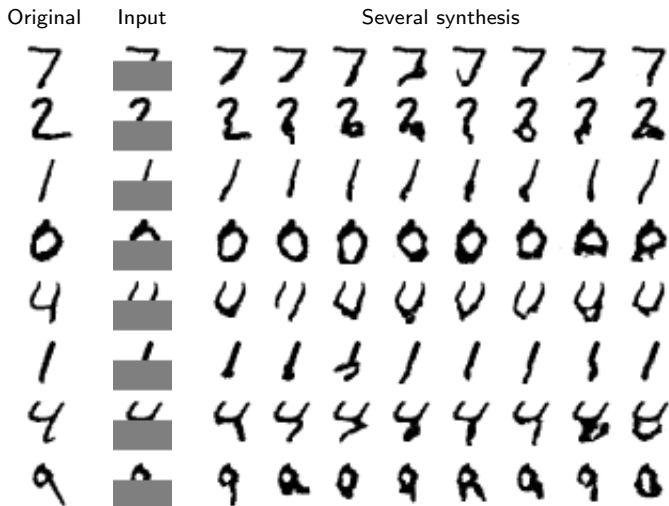
Some generated images



Masks, generated pixels so far, and posterior on the next pixel to generate (red dot), as predicted by the model (logscale). White is 0 and black is 255.



The same generative process can be used for in-painting, by starting the process with available pixel values.



Some remarks:

- The index ordering for the sampling is a design decision. It can be fixed during train and test, or be adaptive.
- Even when there is a clear metric structure on the value space, best results are obtained with cross-entropy over a discretization of it.

This is due in large part to the ability of categorical distributions and cross-entropy to deal with exotic posteriors, in particular multi-modal.

- The cross entropy for a sample is $\ell_n = -\log \hat{p}(y_n)$ hence $e^{\ell_n} = \frac{1}{\hat{p}(y_n)}$.

If the predicted posterior was uniform on N values, this loss value would correspond to $N = e^{\ell_n}$. This is the **perplexity** and is often monitored as a more intuitive quantity.

References

- H. Larochelle and I. Murray. **The neural autoregressive distribution estimator**. In International Conference on Artificial Intelligence and Statistics (AISTATS), pages 29–37, 2011.

Deep learning

10.2. Causal convolutions

François Fleuret

<https://fleuret.org/dlc/>



UNIVERSITÉ
DE GENÈVE

If we use an autoregressive model with a **masked input** as we saw in lecture 10.1. “Auto-regression”

$$f : \{0, 1\}^T \times \mathbb{R}^T \rightarrow \mathbb{R}^C$$

the input differs from a position to another.

During training, even though the full sequence is known, common computation is lost.

Instead of predicting [the distribution of] one component, the model could make a prediction at every position of the sequence, that is

$$f : \mathbb{R}^T \rightarrow \mathbb{R}^{T \times C}.$$

It can be used for synthesis with

$$\begin{aligned}x_1 &\leftarrow \text{sample}(f_1(0, \dots, 0)) \\x_2 &\leftarrow \text{sample}(f_2(x_1, 0, \dots, 0)) \\x_3 &\leftarrow \text{sample}(f_3(x_1, x_2, 0, \dots, 0)) \\&\dots \\x_T &\leftarrow \text{sample}(f_T(x_1, x_2, \dots, x_{T-1}, 0))\end{aligned}$$

where the 0s simply fill in for unknown values, and the mask is not needed.

If additionally, the model is such that “future values” do not influence the prediction at a certain time, that is

$$\forall t, x_1, \dots, x_t, \alpha_1, \dots, \alpha_{T-t}, \beta_1, \dots, \beta_{T-t}, \\ f_{t+1}(x_1, \dots, x_t, \alpha_1, \dots, \alpha_{T-t}) = f_{t+1}(x_1, \dots, x_t, \beta_1, \dots, \beta_{T-t})$$

then, we have in particular

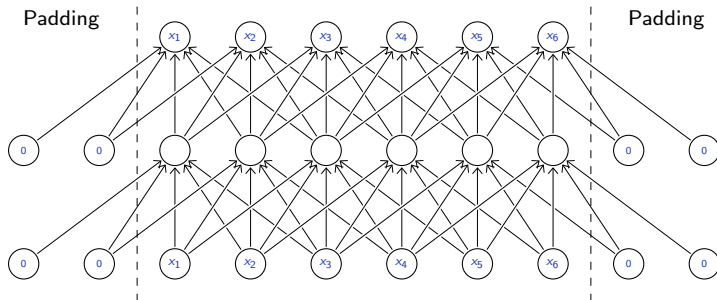
$$\begin{aligned} f_1(0, \dots, 0) &= f_1(x_1, \dots, x_T) \\ f_2(x_1, 0, \dots, 0) &= f_2(x_1, \dots, x_T) \\ f_3(x_1, x_2, 0, \dots, 0) &= f_3(x_1, \dots, x_T) \\ &\dots \\ f_T(x_1, x_2, \dots, x_{T-1}, 0) &= f_T(x_1, \dots, x_T) \end{aligned}$$

This provides a tremendous computational advantage during training, since all the $f_t(x_1, \dots, x_T)$ can be computed with a single forward pass:

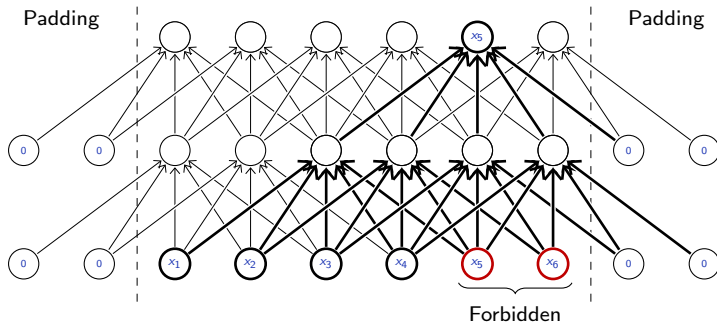
$$\begin{aligned}\ell(f, x) &= \sum_t \ell(f_t(x_1, \dots, x_{t-1}, 0, \dots, 0), x_t) \\ &= \sum_t \ell(\underbrace{f_t(x_1, \dots, x_T)}_{f \text{ is computed once}}, x_t).\end{aligned}$$

Such models are referred to as **causal**, since the future cannot affect the past.

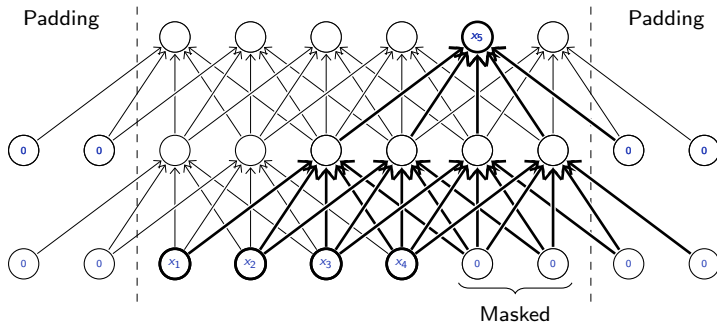
We can illustrate this with convolutional models. Standard convolutions let information flow “to the past,” and masked input was a way to condition only on already generated values.



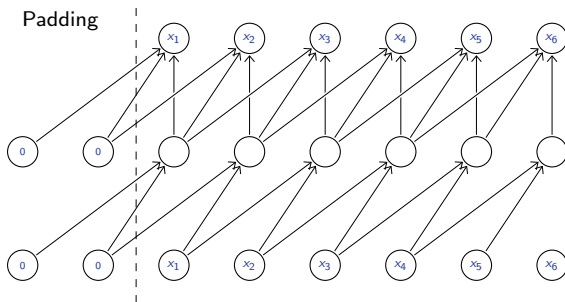
We can illustrate this with convolutional models. Standard convolutions let information flow “to the past,” and masked input was a way to condition only on already generated values.



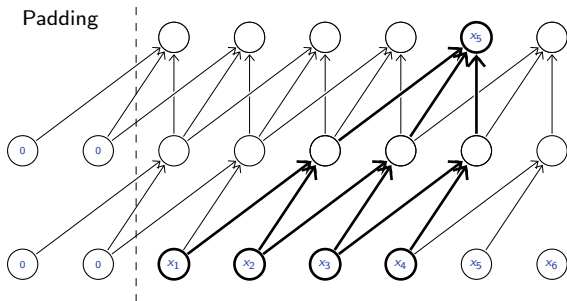
We can illustrate this with convolutional models. Standard convolutions let information flow “to the past,” and masked input was a way to condition only on already generated values.



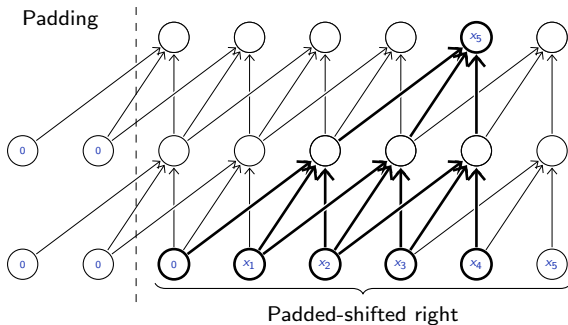
Such a model can be made **causal** with convolutions that let information flow only to the future, combined with a first convolution that hides the present.



Such a model can be made **causal** with convolutions that let information flow only to the future, combined with a first convolution that hides the present.



Another option for the first layer is to shift the input by one entry to hide the present.



PyTorch's convolutional layers do not accept asymmetric padding, but we can do it with `F.pad`, which even accepts negative padding to remove entries.

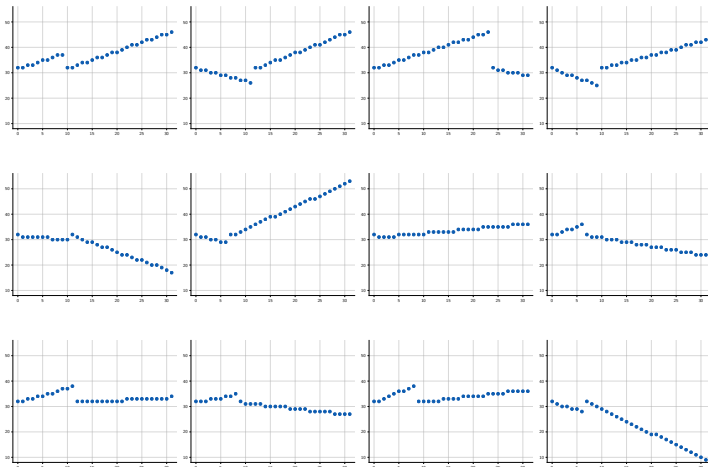
For a n -dim tensor, the padding specification is

$$(start_n, end_n, start_{n-1}, end_{n-1}, \dots, start_{n-k}, end_{n-k})$$

```
>>> x = torch.randint(10, (2, 1, 5))
>>> x
tensor([[[1, 6, 3, 9, 1]],
        [[4, 8, 2, 2, 9]]])
>>> F.pad(x, (-1, 1))
tensor([[[6, 3, 9, 1, 0]],
        [[8, 2, 2, 9, 0]]])
>>> F.pad(x, (0, 0, 2, 0))
tensor([[[0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0],
        [1, 6, 3, 9, 1]],
        [[0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0],
        [4, 8, 2, 2, 9]]])
```

Similar processing can be achieved with the modules `nn.ConstantPad1d`, `nn.ConstantPad2d`, or `nn.ConstantPad3d`.

Here some train sequences as in lecture 10.1. “Auto-regression”.



Model

```
class NetToy1d(nn.Module):
    def __init__(self, nb_classes, ks = 2, nc = 32):
        super().__init__()
        self.pad = (ks - 1, 0)
        self.conv0 = nn.Conv1d(1, nc, kernel_size = 1)
        self.conv1 = nn.Conv1d(nc, nc, kernel_size = ks)
        self.conv2 = nn.Conv1d(nc, nc, kernel_size = ks)
        self.conv3 = nn.Conv1d(nc, nc, kernel_size = ks)
        self.conv4 = nn.Conv1d(nc, nc, kernel_size = ks)
        self.conv5 = nn.Conv1d(nc, nb_classes, kernel_size = 1)

    def forward(self, x):
        x = F.relu(self.conv0(F.pad(x, (1, -1))))
        x = F.relu(self.conv1(F.pad(x, self.pad)))
        x = F.relu(self.conv2(F.pad(x, self.pad)))
        x = F.relu(self.conv3(F.pad(x, self.pad)))
        x = F.relu(self.conv4(F.pad(x, self.pad)))
        x = self.conv5(x)
        return x.permute(0, 2, 1).contiguous()
```

Training loop

```
for sequences in train_input.split(args.batch_size):  
    input = (sequences - mean)/std  
  
    output = model(input)  
  
    loss = cross_entropy(  
        output.view(-1, output.size(-1)),  
        sequences.view(-1)  
    )  
  
    optimizer.zero_grad()  
    loss.backward()  
    optimizer.step()
```

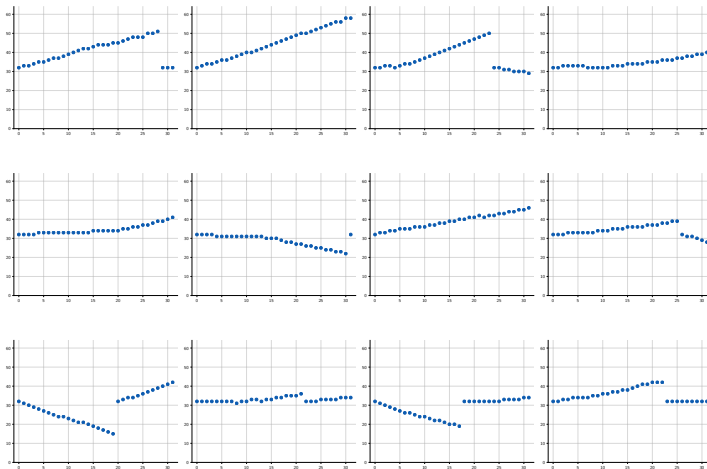

Synthesis

```
generated = train_input.new_zeros((48,) + train_input.size()[1:])

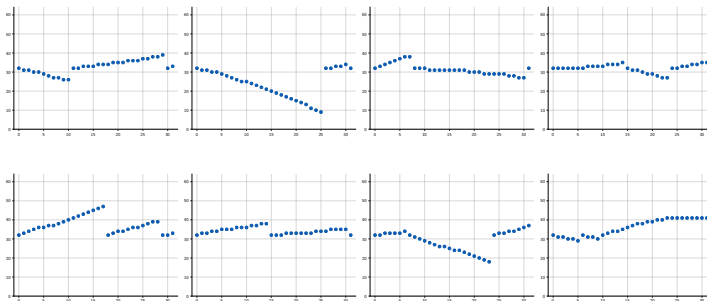
flat = generated.view(generated.size(0), -1)

for t in range(flat.size(1)):
    input = (generated.float() - mean) / std
    output = model(input)
    logits = output.view(flat.size() + (-1,))[:, t]
    dist = torch.distributions.categorical.Categorical(logits = logits)
    flat[:, t] = dist.sample()
```

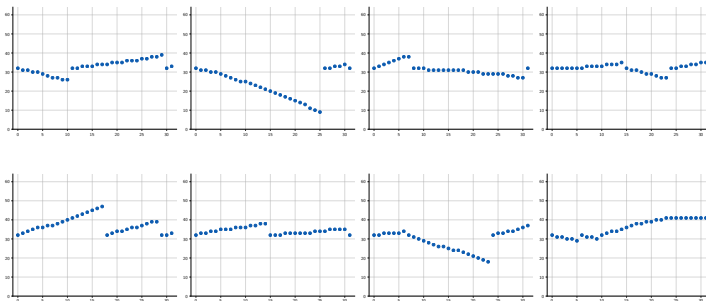
Some generated sequences



The global structure may not be properly generated.



The global structure may not be properly generated.



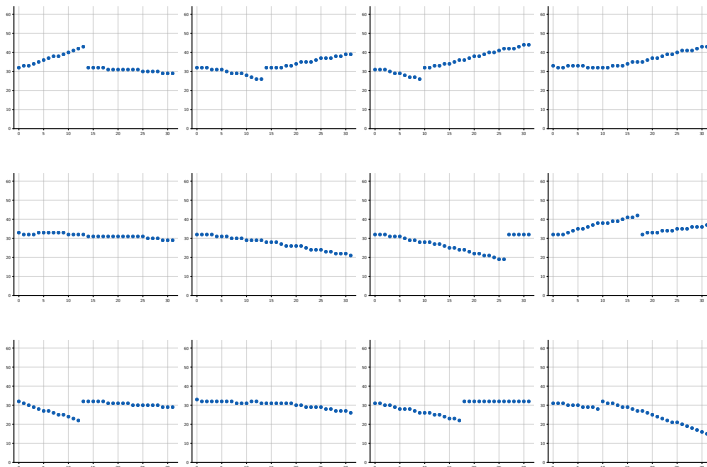
This can be fixed with **dilated convolutions** to have a larger context.

Model

```
class NetToy1dWithDilation(nn.Module):
    def __init__(self, nb_classes, ks = 2, nc = 32):
        super().__init__()
        self.conv0 = nn.Conv1d(1, nc, kernel_size = 1)
        self.pad1 = ((ks-1) * 2, 0)
        self.conv1 = nn.Conv1d(nc, nc, kernel_size = ks, dilation = 2)
        self.pad2 = ((ks-1) * 4, 0)
        self.conv2 = nn.Conv1d(nc, nc, kernel_size = ks, dilation = 4)
        self.pad3 = ((ks-1) * 8, 0)
        self.conv3 = nn.Conv1d(nc, nc, kernel_size = ks, dilation = 8)
        self.pad4 = ((ks-1) * 16, 0)
        self.conv4 = nn.Conv1d(nc, nc, kernel_size = ks, dilation = 16)
        self.conv5 = nn.Conv1d(nc, nb_classes, kernel_size = 1)

    def forward(self, x):
        x = F.relu(self.conv0(F.pad(x, (1, -1))))
        x = F.relu(self.conv1(F.pad(x, self.pad1)))
        x = F.relu(self.conv2(F.pad(x, self.pad2)))
        x = F.relu(self.conv3(F.pad(x, self.pad3)))
        x = F.relu(self.conv4(F.pad(x, self.pad4)))
        x = self.conv5(x)
        return x.permute(0, 2, 1).contiguous()
```

Some generated sequences



The WaveNet model proposed by Oord et al. (2016a) for voice synthesis relies in large part on such an architecture.

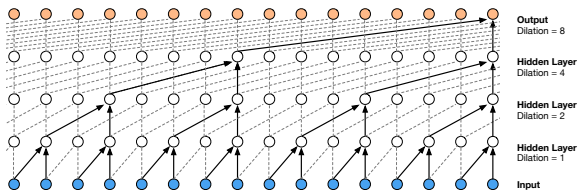


Figure 3: Visualization of a stack of *dilated* causal convolutional layers.

(Oord et al., 2016a)

Causal convolutions for images

The same mechanism can be implemented for images, using causal convolution:

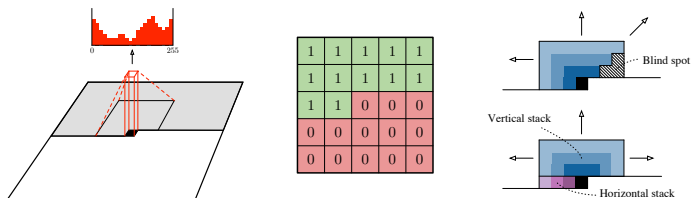


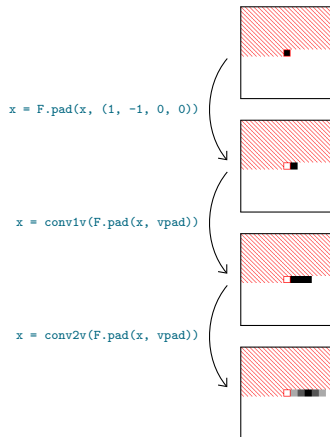
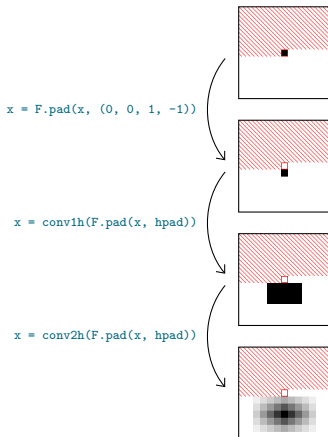
Figure 1: **Left:** A visualization of the PixelCNN that maps a neighborhood of pixels to prediction for the next pixel. To generate pixel x_i the model can only condition on the previously generated pixels x_1, \dots, x_{i-1} . **Middle:** an example matrix that is used to mask the 5x5 filters to make sure the model cannot read pixels below (or strictly to the right) of the current pixel to make its predictions. **Right:** Top: PixelCNNs have a *blind spot* in the receptive field that can not be used to make predictions. Bottom: Two convolutional stacks (blue and purple) allow to capture the whole receptive field.

(Oord et al., 2016b)

```

ks = 5
hpad = (ks//2, ks//2, ks//2, 0)
conv1h = nn.Conv2d(1, 1, kernel_size = (ks//2+1, ks))
conv2h = nn.Conv2d(1, 1, kernel_size = (ks//2+1, ks))
vpad = (ks//2, 0, 0, 0)
conv1v = nn.Conv2d(1, 1, kernel_size = (1, ks//2+1))
conv2v = nn.Conv2d(1, 1, kernel_size = (1, ks//2+1))

```



```

class PixelCNN(nn.Module):
    def __init__(self, nb_classes, in_channels = 1, ks = 5):
        super().__init__()

        self.hpad = (ks//2, ks//2, ks//2, 0)
        self.vpad = (ks//2, 0, 0, 0)

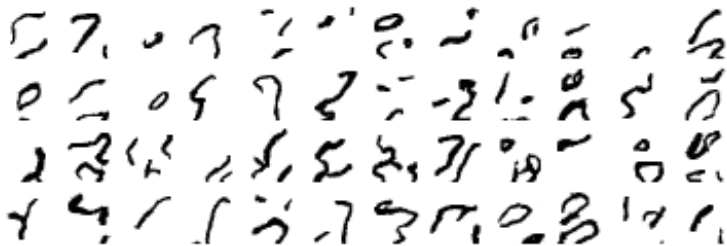
        self.conv1h = nn.Conv2d(in_channels, 32, kernel_size = (ks//2+1, ks))
        self.conv2h = nn.Conv2d(32, 64, kernel_size = (ks//2+1, ks))
        self.conv1v = nn.Conv2d(in_channels, 32, kernel_size = (1, ks//2+1))
        self.conv2v = nn.Conv2d(32, 64, kernel_size = (1, ks//2+1))
        self.final1 = nn.Conv2d(128, 128, kernel_size = 1)
        self.final2 = nn.Conv2d(128, nb_classes, kernel_size = 1)

    def forward(self, x):
        xh = F.pad(x, (0, 0, 1, -1))
        xv = F.pad(x, (1, -1, 0, 0))
        xh = F.relu(self.conv1h(F.pad(xh, self.hpad)))
        xv = F.relu(self.conv1v(F.pad(xv, self.vpad)))
        xh = F.relu(self.conv2h(F.pad(xh, self.hpad)))
        xv = F.relu(self.conv2v(F.pad(xv, self.vpad)))
        x = F.relu(self.final1(torch.cat((xh, xv), 1)))
        x = self.final2(x)

    return x.permute(0, 2, 3, 1).contiguous()

```

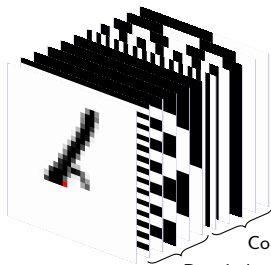
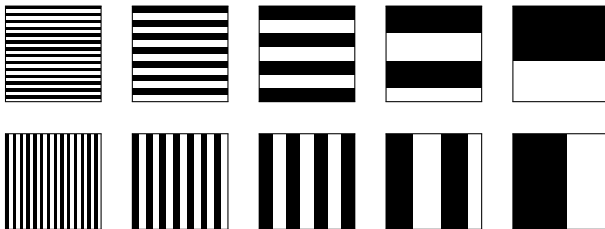
Some generated images



Such a fully convolutional model has no way to make the prediction position-dependent, which results here in local consistency, but fragmentation.

A classical fix is to supplement the input with a **positional encoding**, that is a multi-channel input that provides full information about the location.

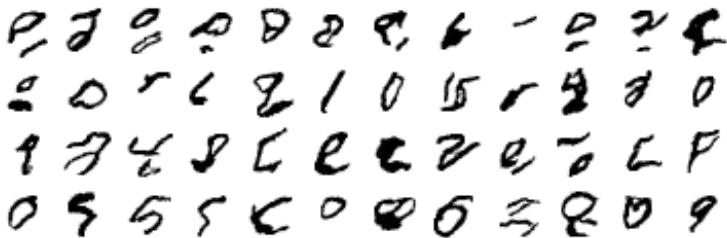
Here with a resolution of 28×28 we can encode the positions with 5 Boolean channels per coordinate.



Input tensor with
positional encoding

Column index encoding
Row index encoding

Some generated images



References

- A. Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu. **WaveNet: A generative model for raw audio**. CoRR, abs/1609.03499, 2016a.
- A. Oord, N. Kalchbrenner, O. Vinyals, L. Espeholt, A. Graves, and K. Kavukcuoglu. **Conditional image generation with PixelCNN decoders**. CoRR, abs/1606.05328, 2016b.

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

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

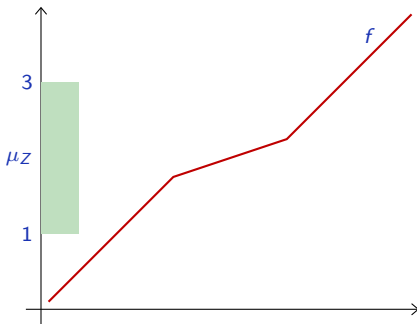
A standard result of probability theory is that if f is continuous, invertible and [almost everywhere] differentiable, and $X = f^{-1}(Z)$, then

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



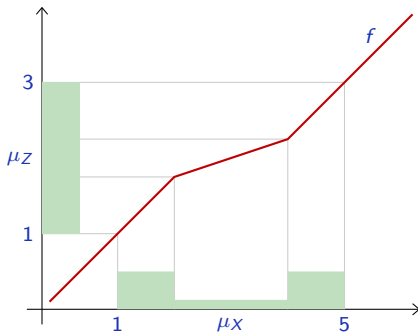
A standard result of probability theory is that if f is continuous, invertible and [almost everywhere] differentiable, and $X = f^{-1}(Z)$, then

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



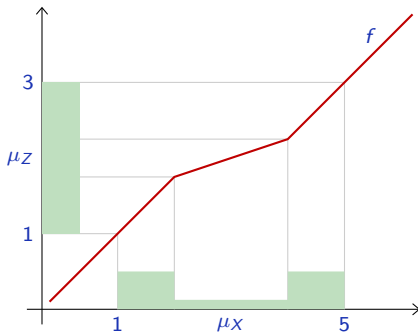
A standard result of probability theory is that if f is continuous, invertible and [almost everywhere] differentiable, and $X = f^{-1}(Z)$, then

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



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

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



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

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

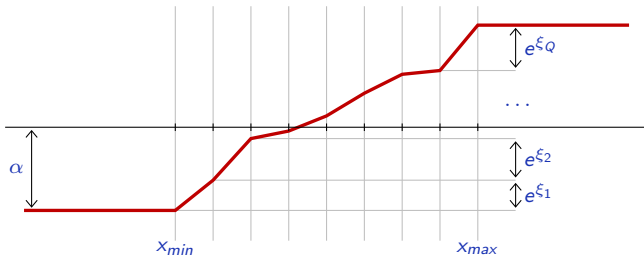
$$\sum_n \log \mu_X(x_n) = \sum_n \log \mu_Z(f(x_n)) + \log |J_f(x_n)|.$$

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

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

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 $\alpha, \xi_1, \dots, \xi_Q$.




```

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

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

so we should minimize

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

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

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

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

```

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

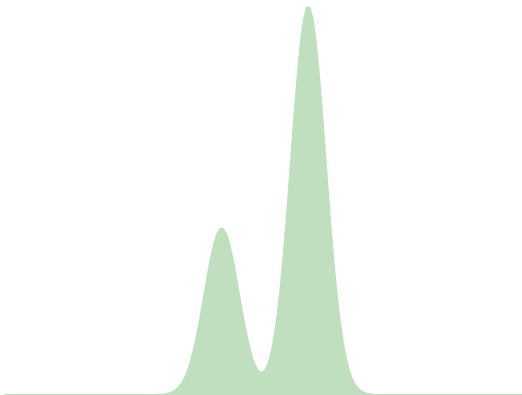
    derivatives, = autograd.grad(
        output.sum(), input,
        retain_graph = True, create_graph = True
    )

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

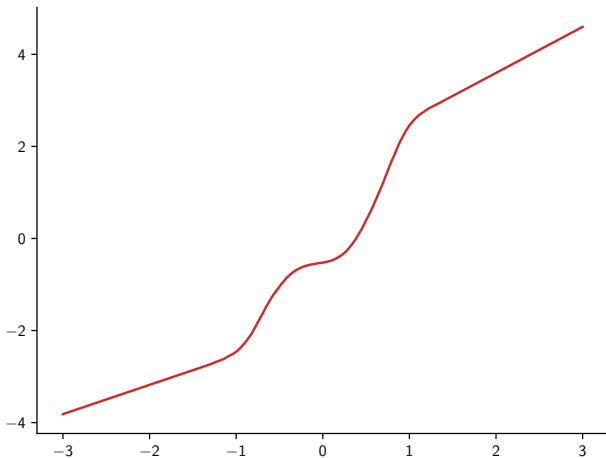
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

```

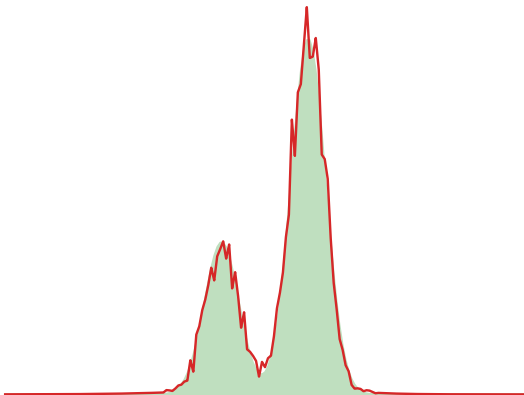
Target distribution μ .

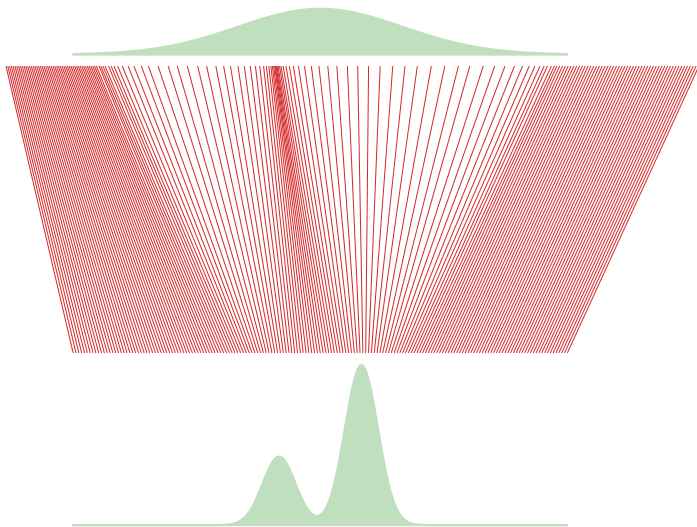


Resulting mapping \hat{f} .



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

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

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

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

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

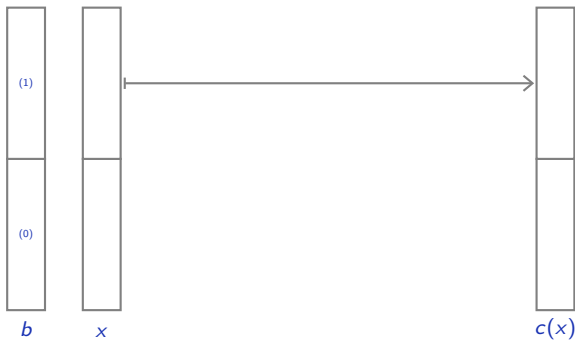
For clarity in what follows, b has all 1s first, followed by 0s, but this is not required.

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

The expression

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

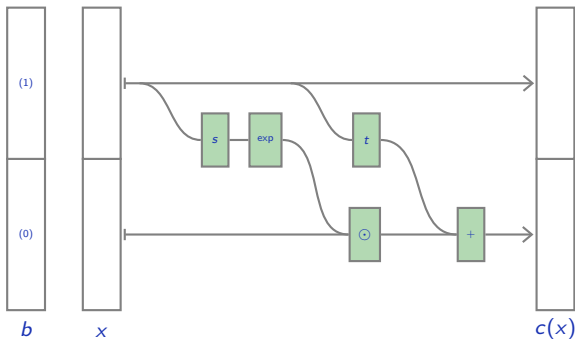
can be understood as: forward $b \odot x$ unchanged,



The expression

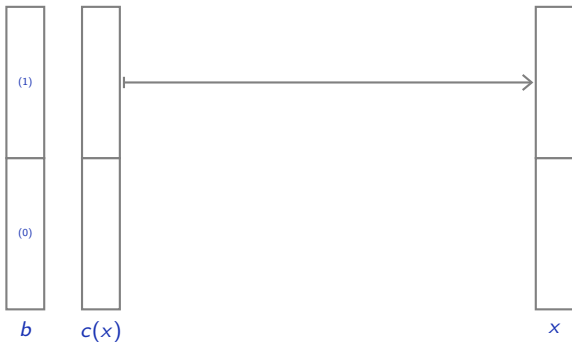
$$c(x) = b \odot x + (1 - b) \odot \left(x \odot \exp(s(b \odot x)) + t(b \odot x) \right)$$

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



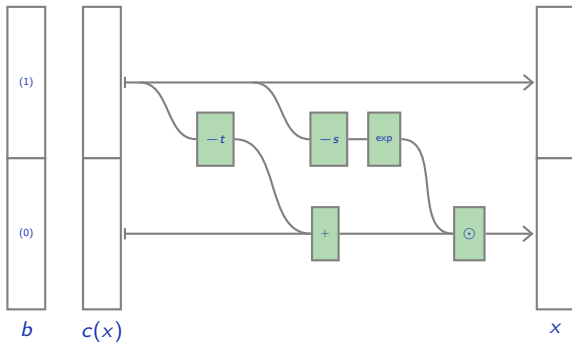
The consequence is that c is invertible, and if $y = c(x)$

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



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.

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

and we have

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

```

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)

```

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

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

with

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

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

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

```
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

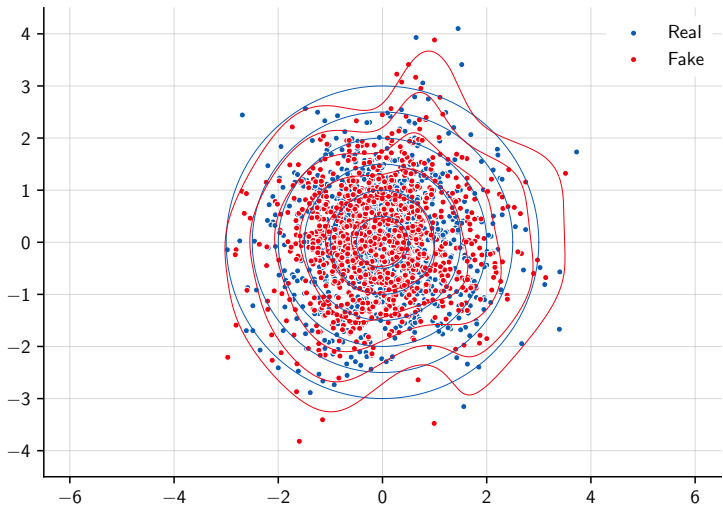
model = NVPNet(dim = 2, hidden_dim = 2, depth = 4)
optimizer = optim.Adam(model.parameters(), lr = 1e-2)

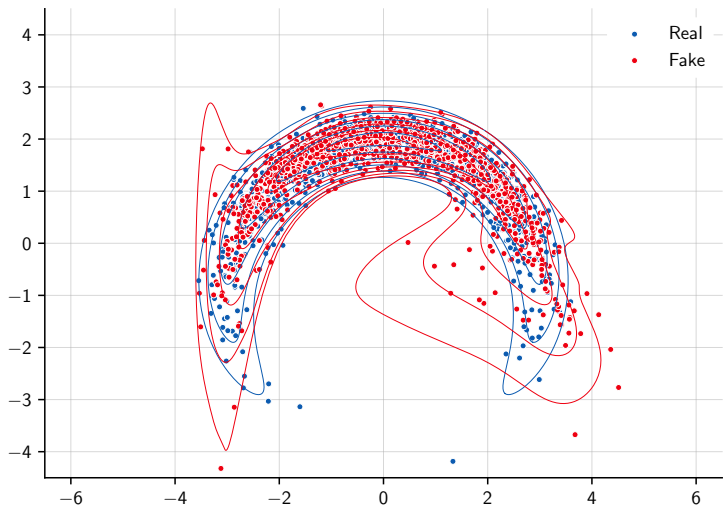
for e in range(args.nb_epochs):

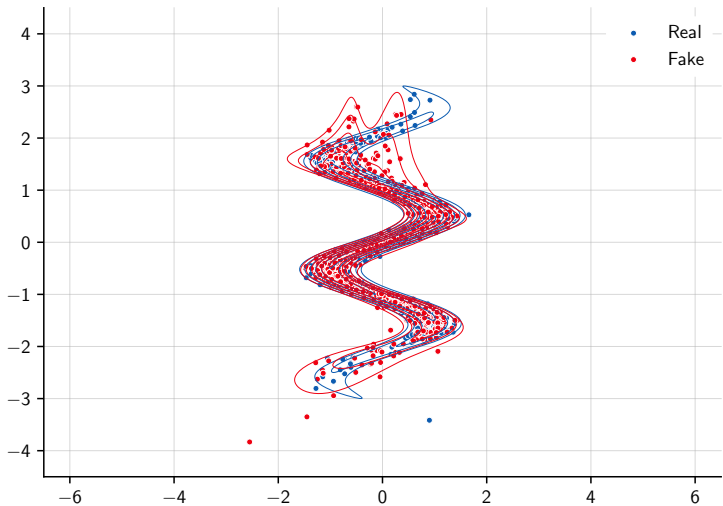
    for input in train_input.split(batch_size):
        output, ldj = model(input, 0)
        loss = - LogProba(output, ldj).mean()
        model.zero_grad()
        loss.backward()
        optimizer.step()
```

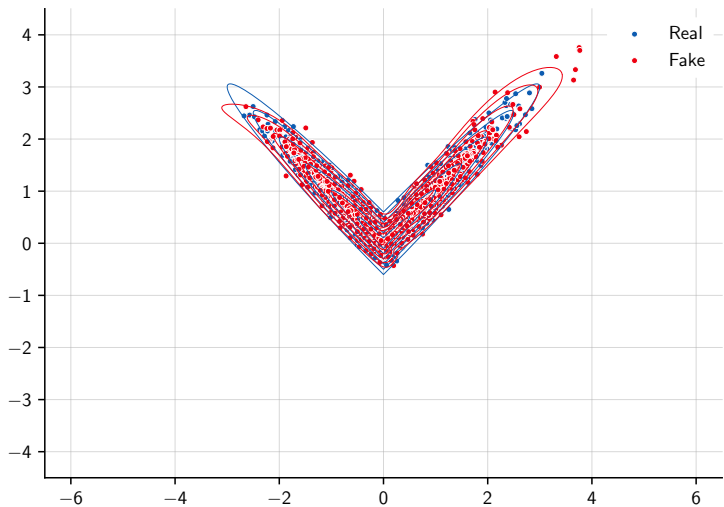
Finally, we can sample according to μ_X with

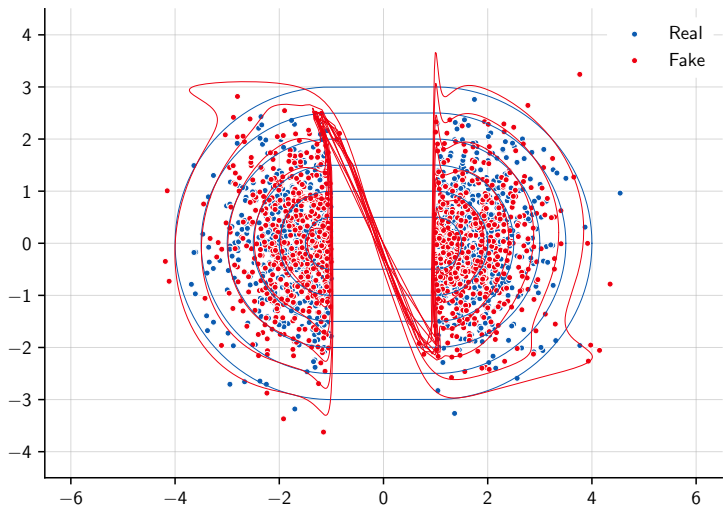
```
z = torch.randn(nb_generated_samples, 2)
x = model.invert(z)
```











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

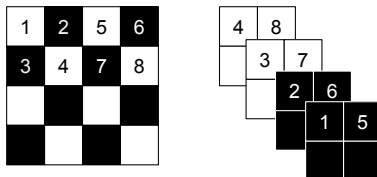
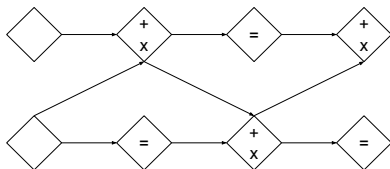


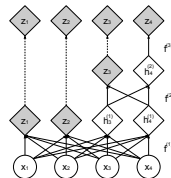
Figure 3: Masking schemes for affine coupling layers. On the left, a spatial checkerboard pattern mask. On the right, a channel-wise masking. The squeezing operation reduces the $4 \times 4 \times 1$ tensor (on the left) into a $2 \times 2 \times 4$ tensor (on the right). Before the squeezing operation,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

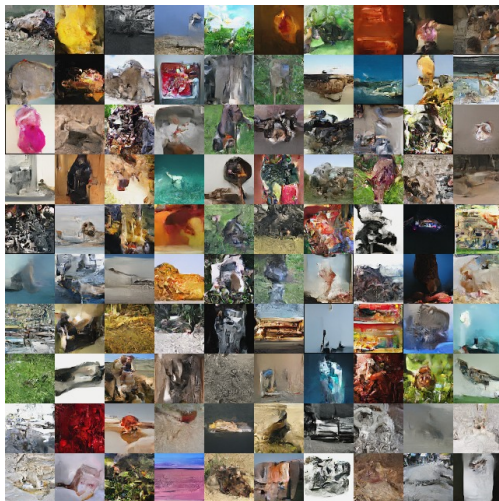


Figure 7: Samples from a model trained on *Imagenet* (64×64).

(Dinh et al., 2016)

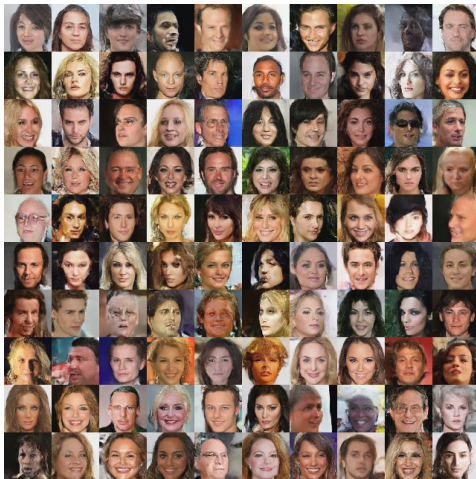


Figure 8: Samples from a model trained on *CelebA*.

(Dinh et al., 2016)

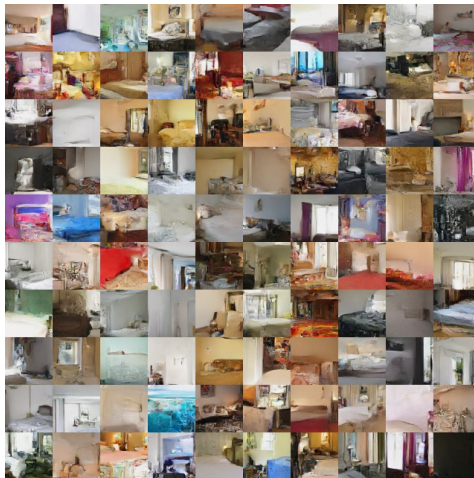


Figure 9: Samples from a model trained on *LSUN* (bedroom category).

(Dinh et al., 2016)



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

(Dinh et al., 2016)

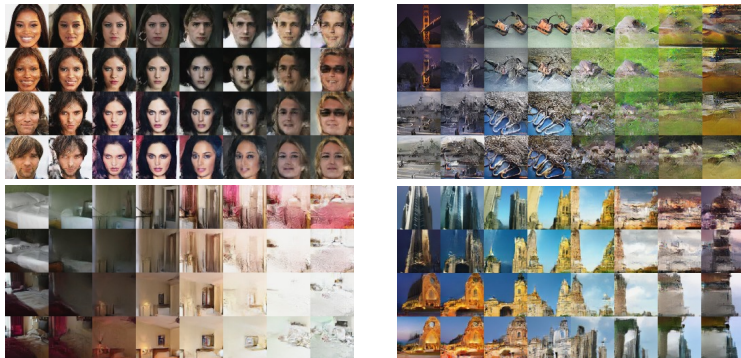


Figure 6: Manifold generated from four examples in the dataset. Clockwise from top left: CelebA, Imagenet (64×64), LSUN (tower), LSUN (bedroom).

(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.