Deep learning

10.1. Auto-regression

François Fleuret

https://fleuret.org/dlc/

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 $X_1, \ldots, X_T$ random variables, we have

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

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 $X_1, \ldots, X_T$ on $\mathbb{R}$, we can represent a conditioning event of the form

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

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:

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

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.

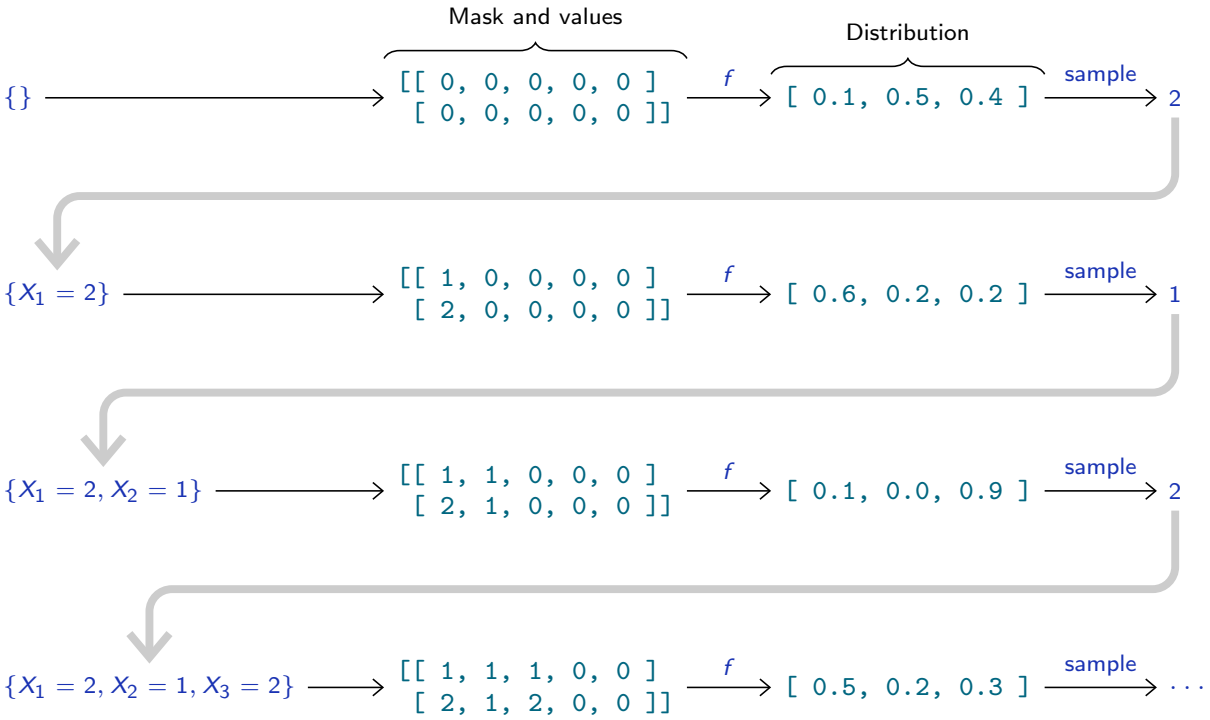
Given such a model and a sampling procedure sample, the generative process for a full sequence is

$$
\begin{aligned}
x_1 &\leftarrow \text{sample}\,(f(\{\})) \\
x_2 &\leftarrow \text{sample}\,(f(\{X_1 = x_1\})) \\
x_3 &\leftarrow \text{sample}\,(f(\{X_1 = x_1, X_2 = x_2\})) \\
&\dots \\
x_T &\leftarrow \text{sample}\,(f(\{X_1 = x_1, X_2 = x_2, \dots, X_{T-1} = x_{T-1}\}))
\end{aligned}
$$

Notes

A sampling procedure takes as input the probabilities (or logits) output by the model (a tensor in $\mathbb{R}^C$) and outputs a value sampled randomly according to the provided probabilities or logits.

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]])
```


Notes

In the batch case, the sampler is parameterized by a tensor of size

$$M_1 \times \cdots \times M_K \times C,$$

that represents

$$M_1 \times \cdots \times M_K$$

vectors of logits over C classes.
The sampling itself takes as input a shape $(N_1, \ldots, N_L)$ and returns a tensor of size

$$N_1 \times \cdots \times N_L \times M_1 \times \cdots \times M_K$$

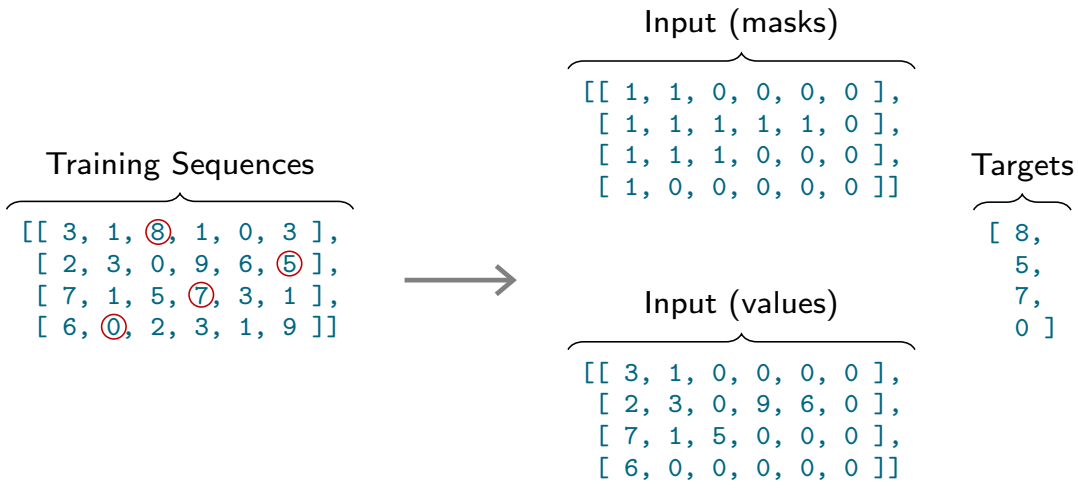
of values in $\{0, \ldots, C-1\}$.

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

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

where ℓ is the cross-entropy.

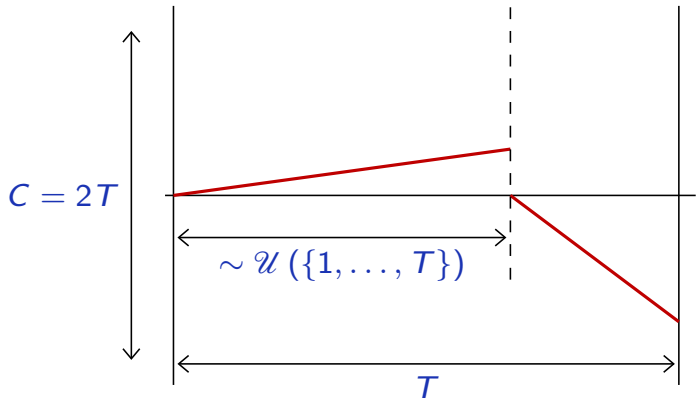
In practice, for each batch, we sample a position to predict for each sample at random, from which we build the masks, conditioning values, and target values.



Notes

The training can be done with mini-batches which are generated as follows:
We start from a tensor of training sequences, and pick the position of the value to predict at random in each, depicted with the red circles.
We create a mask and a value tensor with 1s and values in each sequence up to the value before the value to predict and zeros after.
We create the target vector with the values to predict for each sequence.

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

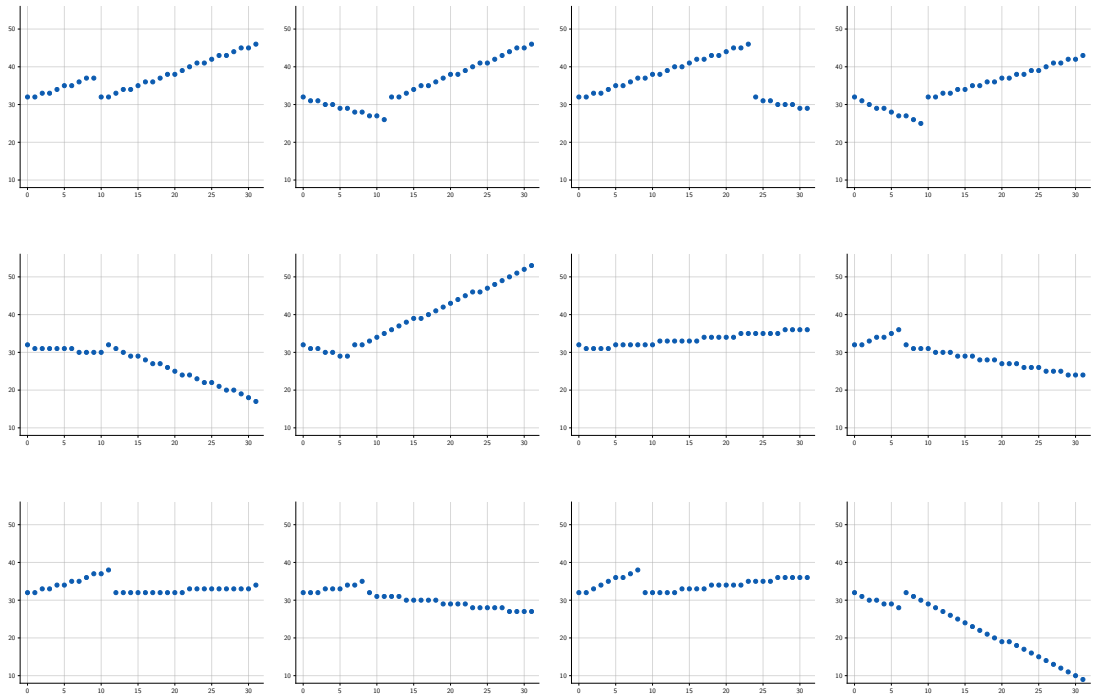


Values are re-centered and discretized into $2T$ values.

Notes

Each sequence is of length T, and take values in $\{0, \ldots 2T - 1\}$ hence $C = 2T$ different possible values at each point.
A sequence starts at the middle value, $x_1 = T$.
The sequence behaves linearly with a slope in $[-1, 1]$ until a "cut" point at which it re-starts again at value T with a new slope in $[-1, 1]$.
This split point is chosen uniformly in $[1, T]$.
The sketch shows an example where the two slopes are of opposite signs, but they may be of same sign.

Some train sequences



Notes

This toy example exhibits 2 interesting features:

- a macro structure (the cut), and
- at every point, a local affine structure.

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
```

Notes

Me use a standard convolutional network. Note the `1d` suffix for 1d modules functions, as opposed to 2d for images.
The input to the network has two channels: one for the mask, and one for the value tensor.
The output of the model is a tensor of size specified with the argument `nb_values`.

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()
```

Notes

`gather` gets in one single tensor the values at the indexes `idx` picked to predict.

Synthesis

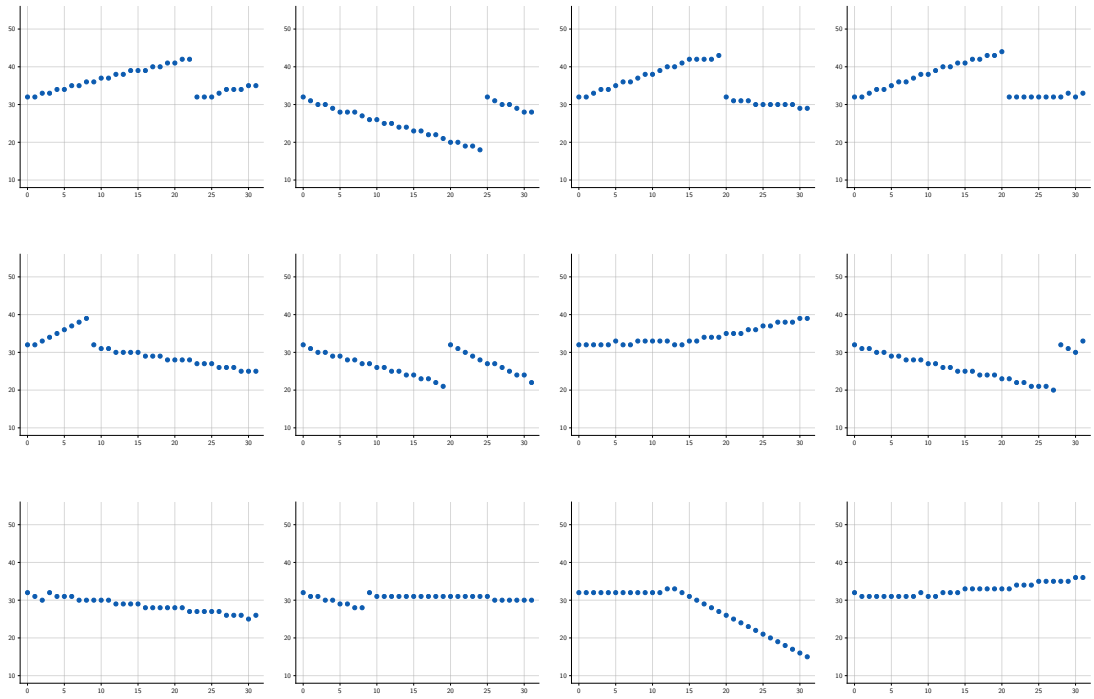
```
nb = 25
generated = torch.zeros(nb, len, device = device, dtype = torch.int64)
tics = torch.arange(len, device = device).view(1, -1).expand(nb, -1)

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()
```


Notes

To synthetize one sequence, components are generated one after the other for all the sequences in parallel.
At each time step, the mask and the value tensors are created from the time step and the values already generated stored in `generated`, the values at the current time step are drawn from the distributions and stored.

Some generated sequences



Notes

The results of the generated sequences are quite satisfying. The model learned that there is a cut, and the both parts before and after are linear. The model was able to leverage the convolutional structure of the network:

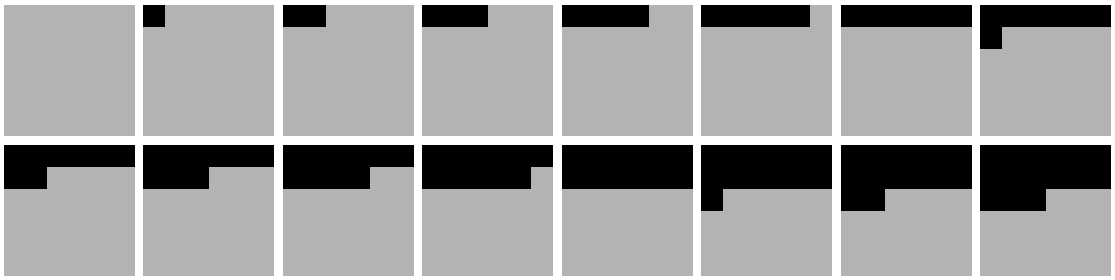
- to predict that, given what had been generated so far, the next value has to be "aligned",
- to predict that, if there were no cut until a certain point, it should break the first pattern.

Image auto-regression

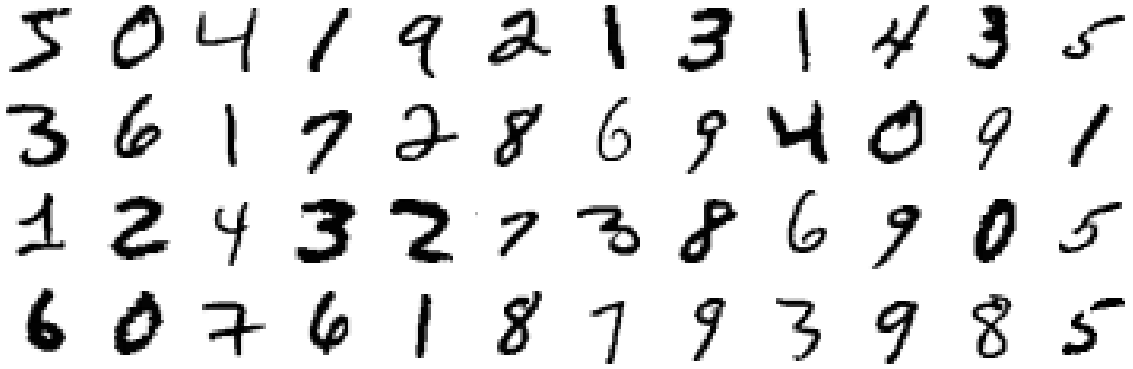
The exact same auto-regressive approach generalizes to any tensor shape, as long as a visiting order of the coefficients is provided.

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



Notes

MNIST samples are 28×28 gray-scale images. Pixels are in $[0, 255]$. For auto-regression, such a 28×28 image will be interpreted as a sequence of length 784, corresponding to the pixels visited from top to bottom, and from left to right.

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)
```


Notes

In practice, we need a way to go from images to sequences, and from sequences to images.
`seq2tensor` takes as input a tensor of dimension $N \times 784$, that represents N sequences of length 784, and outputs a tensor of size $N \times 1 \times 28 \times 28$ corresponding to a standard batch of MNIST image.
`tensor2seq` does the opposite and transforms a batch of N MNIST images $N \times 1 \times 28 \times 28$, into a tensor of N sequences $N \times 784$.

Model

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

        self.features = nn.Sequential(
            nn.Conv2d(2, 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
```

Notes

The model used here is a standard convnet with the association of convolutions, pooling, and ReLU, followed by two fully connected layers.
The main difference with a LeNet is that the input has two channels:

- one for the mask, which indicates the pixels already known, and
- the other for the actual pixel values.

Here, the number of classes is the number of gray levels (from 0 to 255) that the component of the sequence can take.

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()
```

Notes

The training goes as follows:

- The input images are first converted to sequences to easily generate the masks: the first values will be 1, while the last will be 0.
- The value tensor is also created by zeroing the locations of the pixels which are 0 in the masks.
- Both the masks and the value tensors are then converted to an image shape with `seq2tensor` to properly feed the convnet.

Synthesis

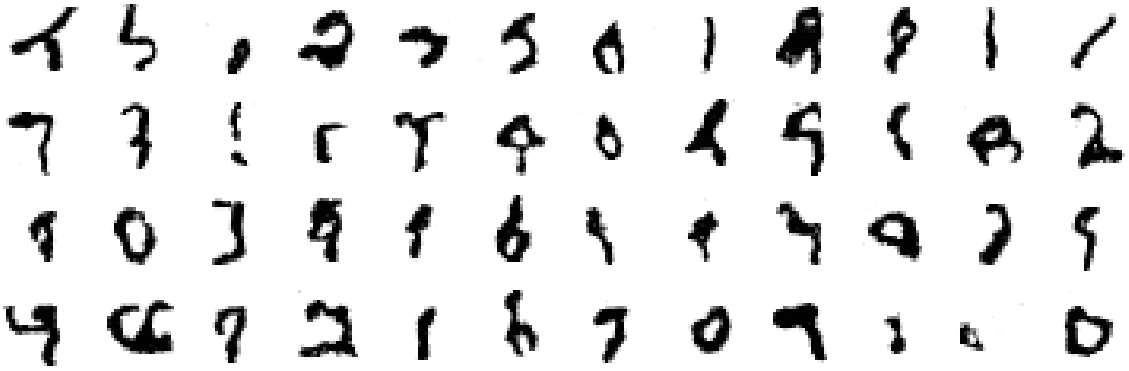
```
nb = 48
generated = torch.zeros((nb,) + train_input.shape[1:],
                        device = device, dtype = torch.int64)
sequences = tensor2seq(generated)
tics = torch.arange(sequences.size(1), device = device).view(1, -1).expand(nb, -1)

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()
```


Notes

The synthesis procedure is very similar to the one in the toy example.

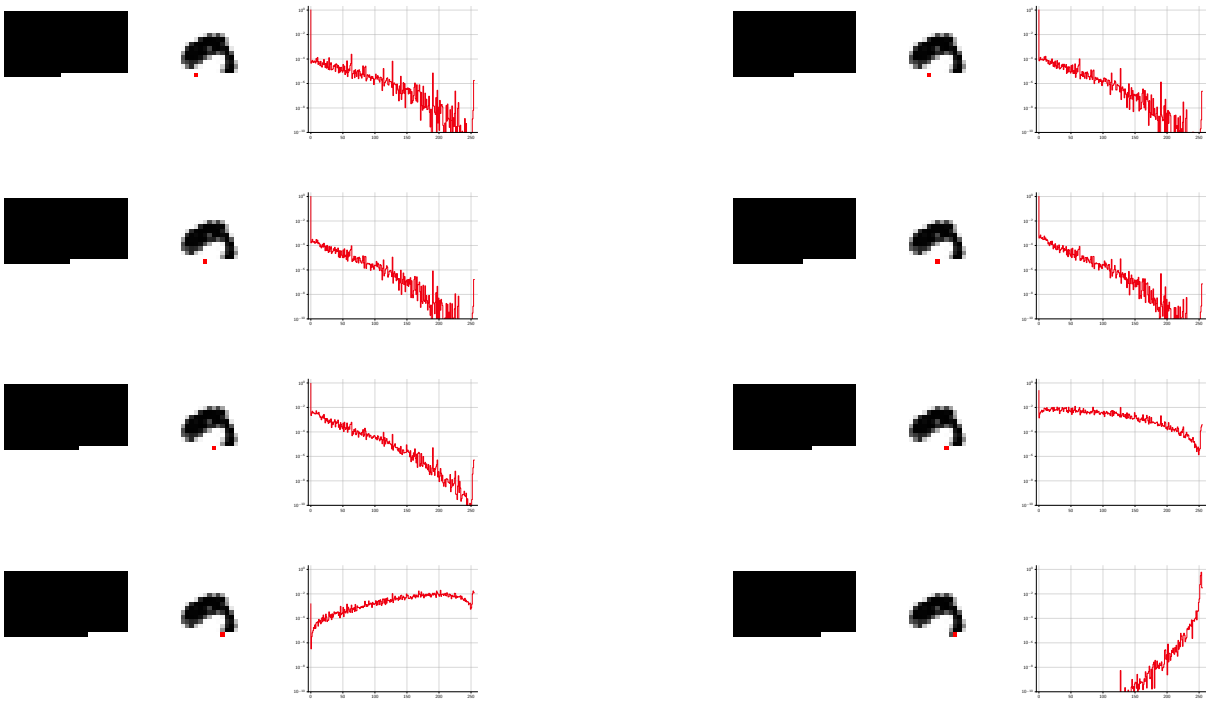
Some generated images



Notes

Although not close to state-of-the-art results, these generated images are satisfying given the simplicity of the model.
The model obviously captured main aspects of the density to be modeled. The overall shape is properly captured: a dark writing in the middle while nothing is drawn around it. Some generated images truly look like real digits.

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.



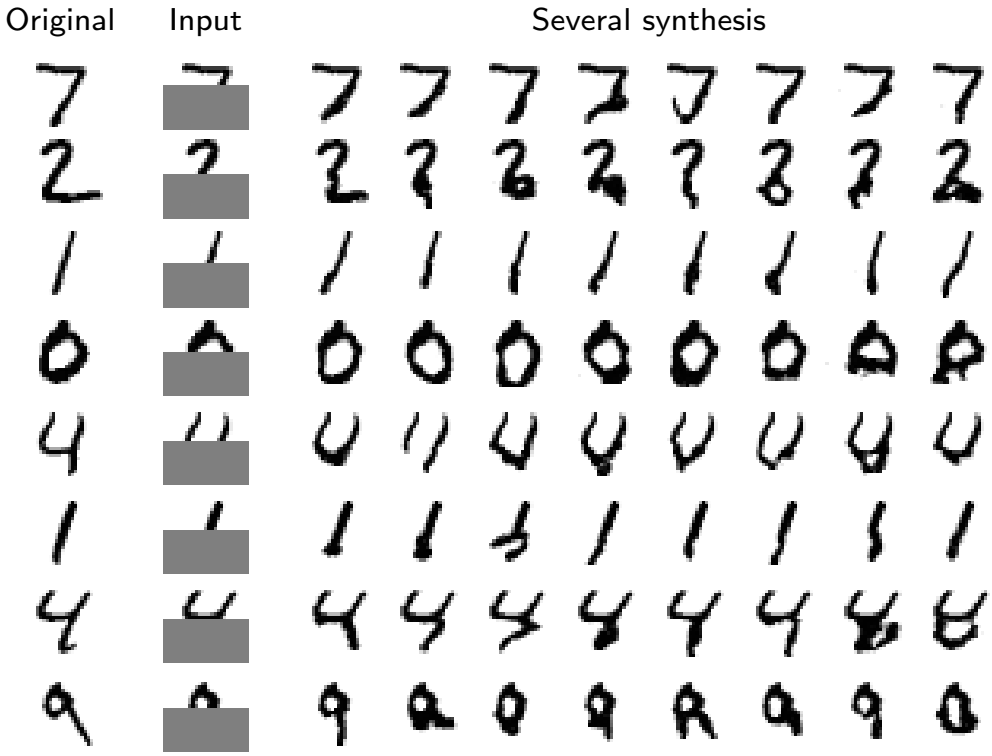
Notes

Each group of three images show:

- Left: the mask. In black the pixels already generated.
- Middle: the pixel sampled so far. The red dot is the location of the pixel which we want to sample next.
- Right: the distribution predicted by the model at he position marked by the red dot, given the mask and the values generated so far. The x axis range from 0 to 255 which correspond to the gray level.

These results confirm the proper behavior of the model: When a pixel should "naturally" be white, because it has no black neighbor on its left or above, and is closer to the frame, the distribution is heavily biased toward white values. When it should extend an existing black trace above, the distribution shifts toward dark values.

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



Notes

In-painting is the process to filling pixel values which have been removed. The in-painting algorithm has access to the rest of the pixels. Here, the pixels were removed in the gray patch.
The main difference with the previous situation is that we do not generate the image from scratch, but we start from a part of a true digit, in this case half of the image, the first fourteen rows.
The generated result are not perfect but often have a consistent structure.

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.