EE-559 – Deep learning

10.1. Auto-regression

François Fleuret
https://fleuret.org/ee559/
Feb 9, 2020

Given $X_1, \dots, X_T$ random variables, the chain rule is that

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

Auto-regression methods use this principle, and model components of a signal in sequence, **each one conditionally to the ones already modeled.**

Deep neural networks are a proper class of models for such conditional densities (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

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

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

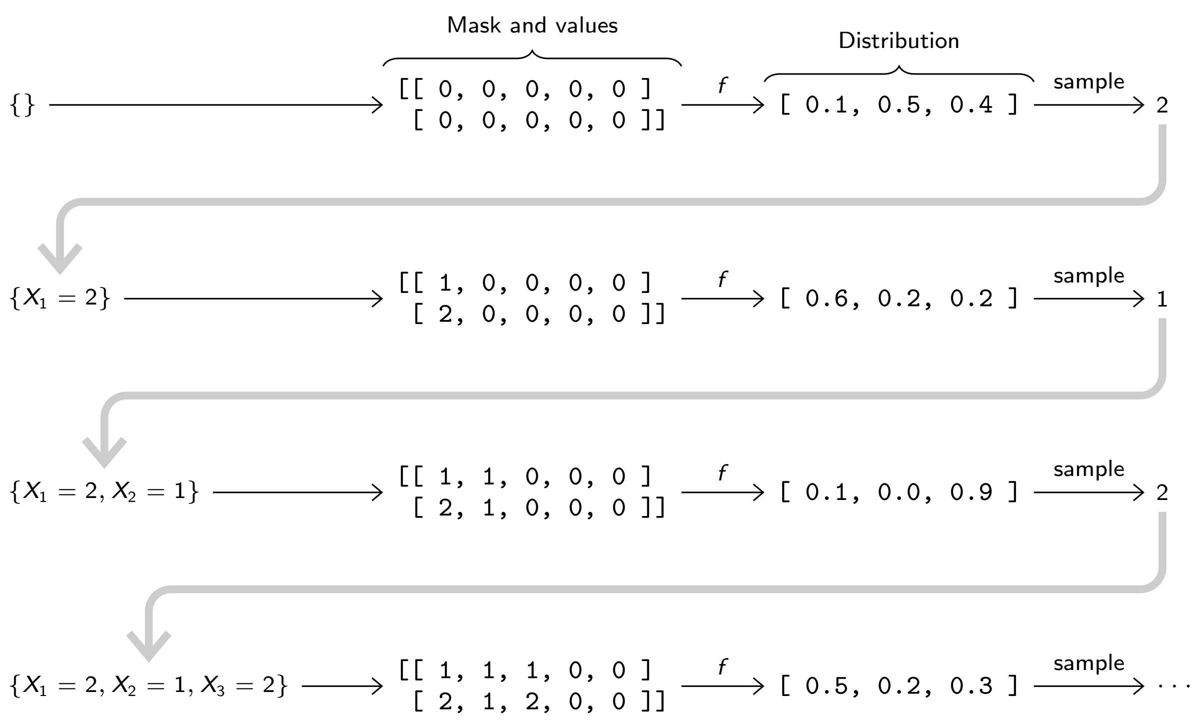
This can be generalized in principle by mapping to parameters of any distribution on $\mathbb{R}$.

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

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

The index ordering for the sampling is a design decision. It can be fixed during train and test, or be adaptive.

With $C = 3$ and $T = 5$:



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

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

where ℓ is the cross-entropy.

In practice, for each batch, we sample an index to predict in each 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 ]]
```

Input

Masks

```
[[ 1, 1, 0, 0, 0, 0 ],
 [ 1, 1, 1, 1, 1, 0 ],
 [ 1, 1, 1, 0, 0, 0 ],
 [ 1, 0, 0, 0, 0, 0 ]]
```

Values

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

Target

```
[ 8,
  5,
  7,
  0 ]
```

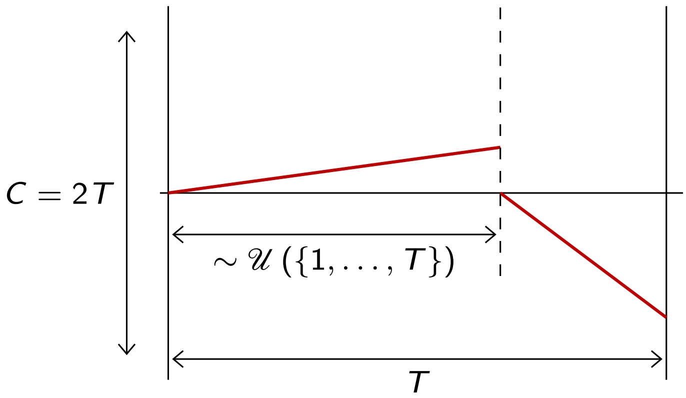

Even when there is a clear metric structure on the value space, best results are obtained with cross-entropy over a discretization of it.

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 single sample is $\ell_n = -\log \hat{p}(y_n)$ hence $e^{\ell_n} = \frac{1}{\hat{p}(y_n)}$.

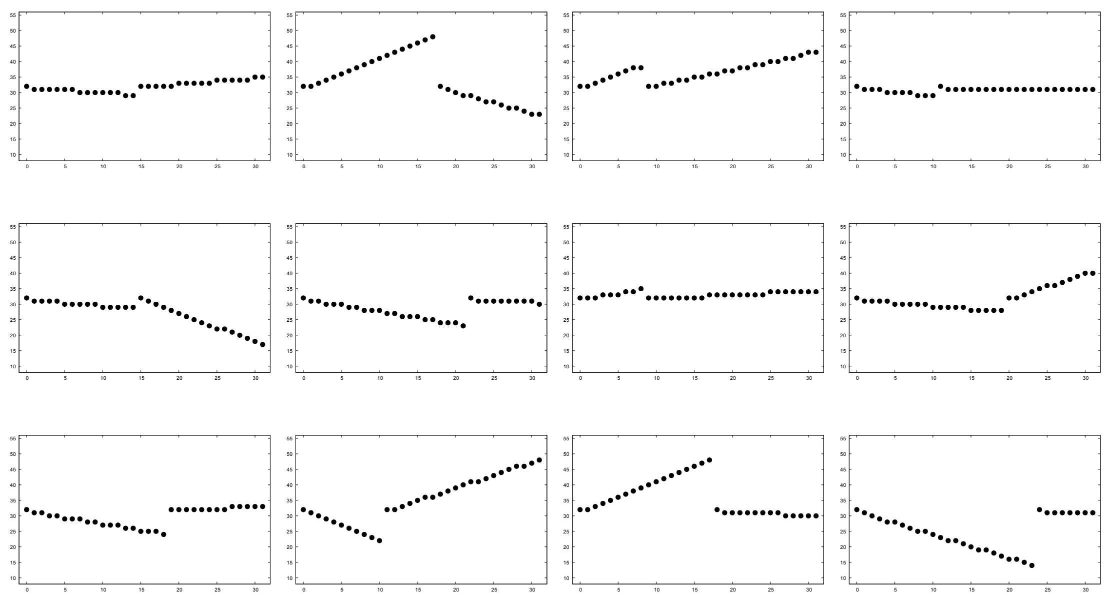
If the predicted posterior was uniform on N values, this loss value would correspond to $N = e^{\ell_n}$. This is the **perplexity** and is often monitored as a more intuitive quantity.

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 $\sim \mathscr{U}([-1, 1])$.



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

Some train sequences



Model

```
class Net(nn.Module):
    def __init__(self, nb_values):
        super(Net, self).__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 the input, set the mask and values as two channels
    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

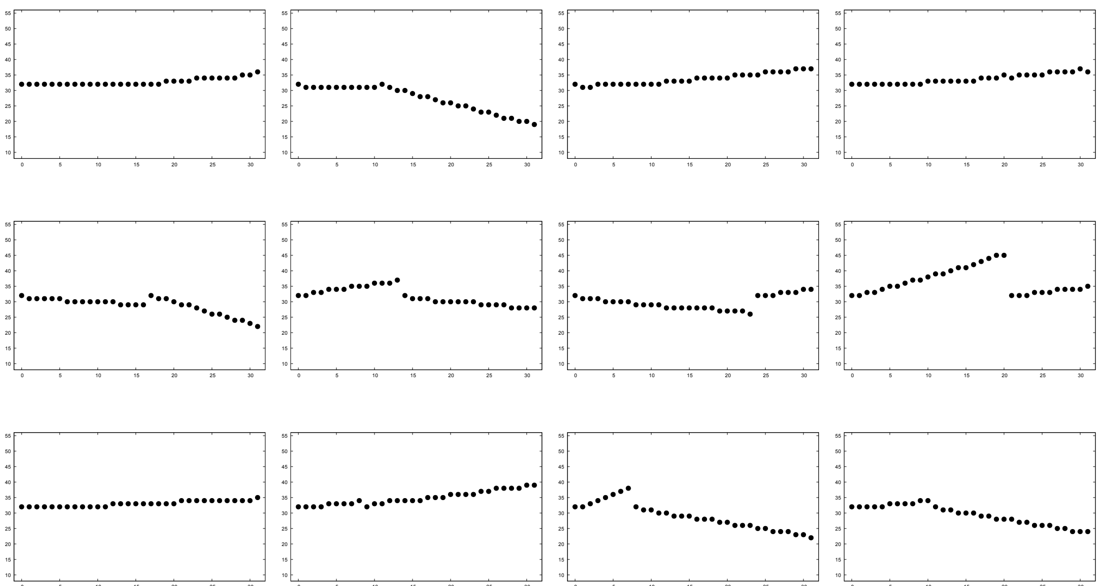
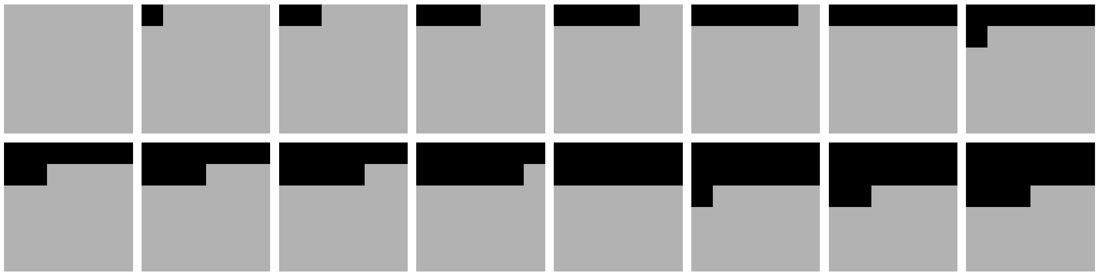


Image auto-regression

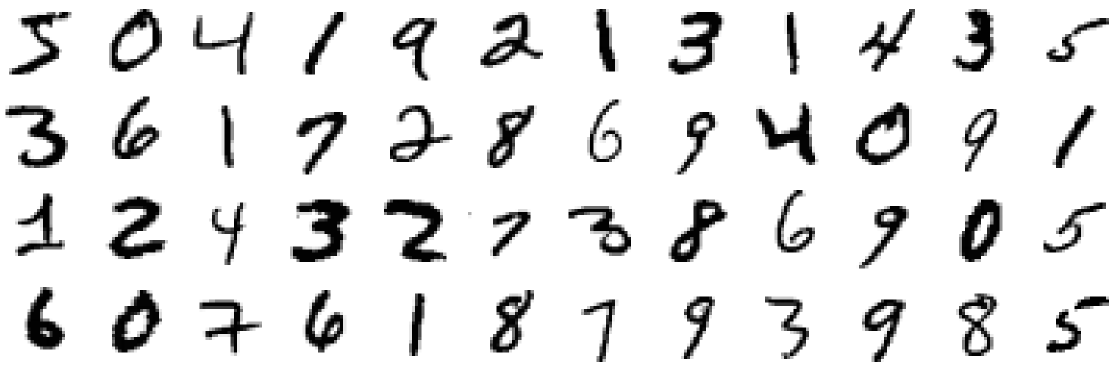
The exact same auto-regressive approach generalizes to any tensor shape, as long as a visiting order of the coefficients is provided.

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)
```

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()
```

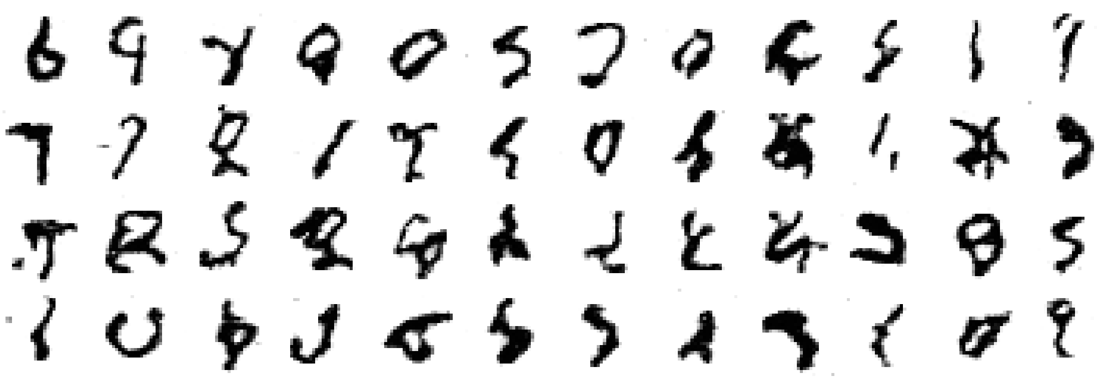

Synthesis

```
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()
```

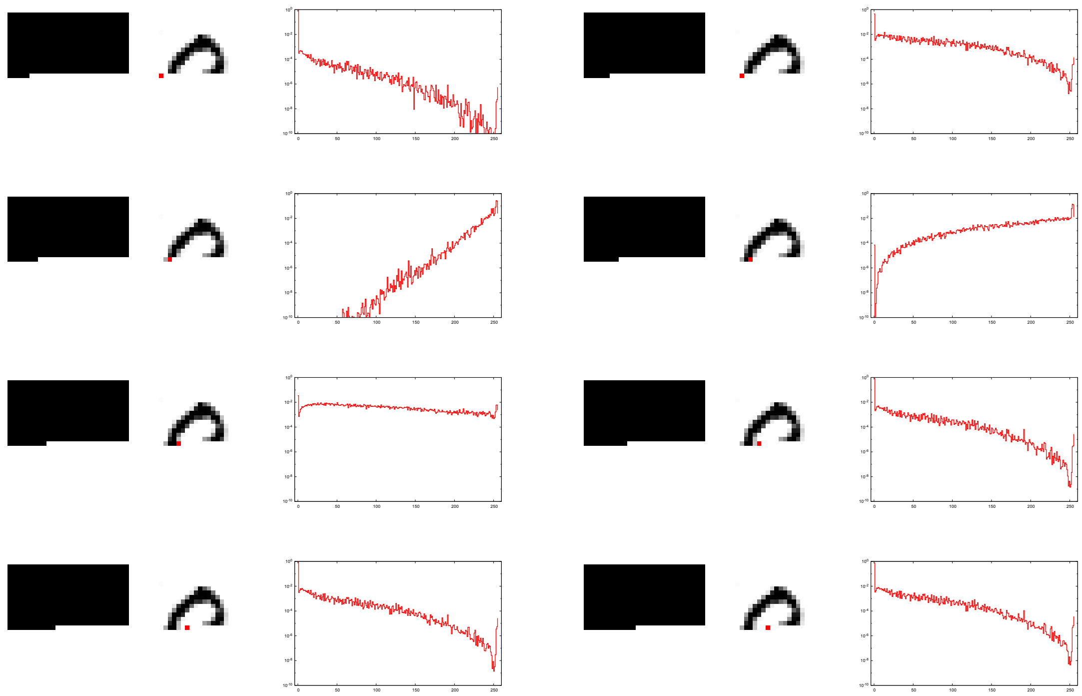
Model

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

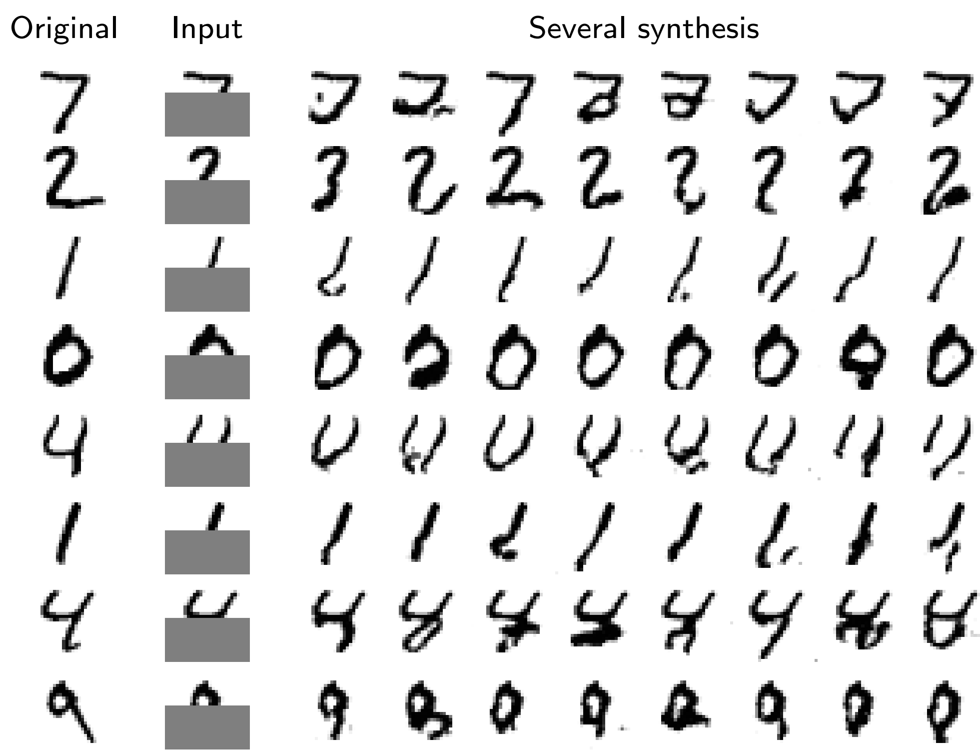

Some generated images



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.



References

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