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

$$\forall x_1, \ldots, x_T, \quad P(X_1 = x_1, \ldots, X_T = x_T) =$$

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

$$\forall x_1, \ldots, x_T, \quad P(X_1 = x_1, \ldots, X_T = x_T) = \prod_{t=1}^{T} P(X_t = x_t | X_1 = x_1, \ldots, X_{t-1} = x_{t-1}).$$

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 $T = 5$

<table>
<thead>
<tr>
<th>Event</th>
<th>Mask tensor</th>
<th>Value tensor</th>
</tr>
</thead>
<tbody>
<tr>
<td>${X_2 = 3}$</td>
<td>[0, 1, 0, 0, 0]</td>
<td>[0, 3, 0, 0, 0]</td>
</tr>
<tr>
<td>${X_1 = 1, X_2 = 2, X_3 = 3, X_4 = 4, X_5 = 5}$</td>
<td>[1, 1, 1, 1, 1]</td>
<td>[1, 2, 3, 4, 5]</td>
</tr>
<tr>
<td>${X_5 = 50, X_2 = 20}$</td>
<td>[0, 1, 0, 0, 1]</td>
<td>[0, 20, 0, 0, 50]</td>
</tr>
</tbody>
</table>
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 of 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.
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

\begin{align*}
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) \\
& \quad \ldots \\
x_T & \leftarrow \text{sample} \left( f(\{X_1 = x_1, X_2 = x_2, \ldots, X_{T-1} = x_{T-1}\}) \right)
\end{align*}
For instance, with $C = 3$ and $T = 5$, we could have:

\[
\begin{bmatrix}
0, 0, 0, 0, 0 \\
0, 0, 0, 0, 0
\end{bmatrix}
\]

Distribution $f_2$ sample $\{X_1 = 2\}$

\[
\begin{bmatrix}
1, 0, 0, 0, 0 \\
2, 0, 0, 0, 0
\end{bmatrix}
\]

$0.6, 0.2, 0.2$

Distribution $f_1$ sample $\{X_1 = 2, X_2 = 1\}$

\[
\begin{bmatrix}
1, 1, 0, 0, 0 \\
2, 1, 0, 0, 0
\end{bmatrix}
\]

$0.1, 0.0, 0.9$

Distribution $f_2$ sample $\{X_1 = 2, X_2 = 1, X_3 = 2\}$

\[
\begin{bmatrix}
1, 1, 1, 0, 0 \\
2, 1, 2, 0, 0
\end{bmatrix}
\]

$0.5, 0.2, 0.3$
For instance, with $C = 3$ and $T = 5$, we could have:

$$\{0, 0, 0, 0, 0\}$$

Mask and values

$$\begin{bmatrix} 0.1, 0.5, 0.4 \\ 0.6, 0.2, 0.2 \\ 0.1, 0.0, 0.9 \end{bmatrix}$$
For instance, with $C = 3$ and $T = 5$, we could have:

\[
\begin{bmatrix}
\begin{array}{cccc}
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0
\end{array}
\end{bmatrix}
\]

Mask and values

\[
\begin{bmatrix}
0.1 & 0.5 & 0.4
\end{bmatrix}
\]

Distribution

\[
\begin{bmatrix}
0.6 & 0.2 & 0.2
\end{bmatrix}
\]

\[
\begin{bmatrix}
1 & 1 & 0 & 0 & 0 \\
2 & 1 & 0 & 0 & 0
\end{bmatrix}
\]

\[
\begin{bmatrix}
0.1 & 0.0 & 0.9
\end{bmatrix}
\]

\[
\begin{bmatrix}
1 & 1 & 1 & 0 & 0 \\
2 & 1 & 2 & 0 & 0
\end{bmatrix}
\]

\[
\begin{bmatrix}
0.5 & 0.2 & 0.3
\end{bmatrix}
\]

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

\[
\begin{bmatrix}
0, 0, 0, 0, 0 \\
0, 0, 0, 0, 0
\end{bmatrix}
\]

\[
\begin{bmatrix}
0.1, 0.5, 0.4 \\
0.6, 0.2, 0.2
\end{bmatrix}
\]

\[
\begin{bmatrix}
1, 0, 0, 0, 0 \\
2, 0, 0, 0, 0
\end{bmatrix}
\]

\[
\begin{bmatrix}
0.1, 0.0, 0.9 \\
0.1, 0.0, 0.9
\end{bmatrix}
\]

\[
\begin{bmatrix}
1, 1, 0, 0, 0 \\
2, 1, 0, 0, 0
\end{bmatrix}
\]

\[
\begin{bmatrix}
0.5, 0.2, 0.3 \\
0.5, 0.2, 0.3
\end{bmatrix}
\]

\[
\begin{bmatrix}
1, 1, 1, 0, 0 \\
2, 1, 2, 0, 0
\end{bmatrix}
\]

\[
\begin{bmatrix}
0.1, 0.5, 0.4 \\
0.1, 0.5, 0.4
\end{bmatrix}
\]

\[
\begin{bmatrix}
0.1, 0.5, 0.4 \\
0.1, 0.5, 0.4
\end{bmatrix}
\]

\[
\begin{bmatrix}
0.1, 0.5, 0.4 \\
0.1, 0.5, 0.4
\end{bmatrix}
\]

\[
\begin{bmatrix}
0.1, 0.5, 0.4 \\
0.1, 0.5, 0.4
\end{bmatrix}
\]
For instance, with $C = 3$ and $T = 5$, we could have:

\[ \begin{bmatrix} 0, 0, 0, 0, 0 \\ 0, 0, 0, 0, 0 \end{bmatrix} \]

Mask and values

\[
\begin{bmatrix}
0.1, 0.5, 0.4 \\
0.6, 0.2, 0.2
\end{bmatrix}
\]

Distribution

\[
\begin{bmatrix}
1, 1, 0, 0, 0 \\
2, 1, 0, 0, 0 \\
1, 1, 1, 0, 0 \\
2, 1, 2, 0, 0
\end{bmatrix}
\]

sample

2

\{X_1 = 2\}
For instance, with $C = 3$ and $T = 5$, we could have:

\[
\begin{align*}
\{\} &\rightarrow \begin{bmatrix} 0, 0, 0, 0, 0 \\ 0, 0, 0, 0, 0 \end{bmatrix} \quad f \rightarrow [0.1, 0.5, 0.4] \rightarrow 2 \\
\{X_1 = 2\} &\rightarrow \begin{bmatrix} 1, 0, 0, 0, 0 \\ 2, 0, 0, 0, 0 \end{bmatrix}
\end{align*}
\]
For instance, with $C = 3$ and $T = 5$, we could have:

\[
\begin{align*}
\{\} & \rightarrow \begin{bmatrix}
0, 0, 0, 0, 0 \\
0, 0, 0, 0, 0
\end{bmatrix} \quad \text{Mask and values} \\
& \rightarrow [0.1, 0.5, 0.4] \quad \text{Distribution}
\end{align*}
\]

\[
\{X_1 = 2\} \rightarrow \begin{bmatrix}
1, 0, 0, 0, 0 \\
2, 0, 0, 0, 0
\end{bmatrix} \text{sample} \rightarrow 2
\]

\[
\begin{align*}
& \rightarrow [0.6, 0.2, 0.2] \\
& f
\end{align*}
\]
For instance, with $C = 3$ and $T = 5$, we could have:

{} $\rightarrow$ $\begin{bmatrix} 0, 0, 0, 0, 0 \\ 0, 0, 0, 0, 0 \end{bmatrix}$ $\rightarrow$ $\begin{bmatrix} 0.1, 0.5, 0.4 \end{bmatrix}$ $\rightarrow$ 2

$\{X_1 = 2\}$ $\rightarrow$ $\begin{bmatrix} 1, 0, 0, 0, 0 \\ 2, 0, 0, 0, 0 \end{bmatrix}$ $\rightarrow$ $\begin{bmatrix} 0.6, 0.2, 0.2 \end{bmatrix}$ $\rightarrow$ 1
For instance, with $C = 3$ and $T = 5$, we could have:

- With $\{\}$, the mask and values are $[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]$ and the distribution is $[0.1, 0.5, 0.4]$. Sampling gives $2$.

- With $\{X_1 = 2\}$, the mask and values are $[[1, 0, 0, 0, 0], [2, 0, 0, 0, 0]]$ and the distribution is $[0.6, 0.2, 0.2]$. Sampling gives $1$.

- With $\{X_1 = 2, X_2 = 1\}$, the diagram is not fully visible to determine the result.
For instance, with $C = 3$ and $T = 5$, we could have:

\[
\begin{align*}
\{\} & \rightarrow \begin{bmatrix} 0, 0, 0, 0, 0 \\ 0, 0, 0, 0, 0 \end{bmatrix} & f & \rightarrow \begin{bmatrix} 0.1, 0.5, 0.4 \end{bmatrix} & \rightarrow \text{sample} & \rightarrow 2 \\
\{X_1 = 2\} & \rightarrow \begin{bmatrix} 1, 0, 0, 0, 0 \\ 2, 0, 0, 0, 0 \end{bmatrix} & f & \rightarrow \begin{bmatrix} 0.6, 0.2, 0.2 \end{bmatrix} & \rightarrow \text{sample} & \rightarrow 1 \\
\{X_1 = 2, X_2 = 1\} & \rightarrow \begin{bmatrix} 1, 1, 0, 0, 0 \\ 2, 1, 0, 0, 0 \end{bmatrix} & f & \rightarrow \begin{bmatrix} 0.1, 0.0, 0.9 \end{bmatrix} & \rightarrow \text{sample} & \rightarrow 2
\end{align*}
\]
For instance, with $C = 3$ and $T = 5$, we could have:

- $\{\}$
  - Mask and values: $[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]$
  - Distribution: $[0.1, 0.5, 0.4]$
  - Sample: $2$

- $\{X_1 = 2\}$
  - Mask and values: $[[1, 0, 0, 0, 0], [2, 0, 0, 0, 0]]$
  - Distribution: $[0.6, 0.2, 0.2]$
  - Sample: $1$

- $\{X_1 = 2, X_2 = 1\}$
  - Mask and values: $[[1, 1, 0, 0, 0], [2, 1, 0, 0, 0]]$
  - Distribution: $[0.1, 0.0, 0.9]$
  - Sample: $2$

- $\{X_1 = 2, X_2 = 1, X_3 = 2\}$
For instance, with $C = 3$ and $T = 5$, we could have:

- $\{\}$
  - Mask and values: $[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]$
  - Distribution: $[0.1, 0.5, 0.4]$ (sample 2)

- $\{X_1 = 2\}$
  - Mask and values: $[[1, 0, 0, 0, 0], [2, 0, 0, 0, 0]]$
  - Distribution: $[0.6, 0.2, 0.2]$ (sample 1)

- $\{X_1 = 2, X_2 = 1\}$
  - Mask and values: $[[1, 1, 0, 0, 0], [2, 1, 0, 0, 0]]$
  - Distribution: $[0.1, 0.0, 0.9]$ (sample 2)

- $\{X_1 = 2, X_2 = 1, X_3 = 2\}$
  - Mask and values: $[[1, 1, 1, 0, 0], [2, 1, 2, 0, 0]]$
  - Distribution: $[0.5, 0.2, 0.3]$ (sample ...
The package `torch.distributions` provides the necessary tools to sample from a variety of distributions.

```python
>>> 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

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

```python
>>> 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

```python
>>> 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

\[
\mathcal{L}(f) = - \sum_n \sum_t \log \hat{p}(X_t = x_{n,t} \mid X_1 = x_{n,1}, \ldots, X_{t-1} = x_{n,t-1})
\]

\[
= \sum_n \sum_t \ell \left( f((1, \ldots, 1, 0, \ldots, 0), (x_{n,1}, \ldots, x_{n,t-1}, 0, \ldots, 0)), x_{n,t} \right)
\]

where \( \ell \) 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 ]]

François Fleuret
Deep learning / 10.1. Auto-regression 8 / 25
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

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

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

Input (masks)

\[
\begin{bmatrix}
1, 1, 0, 0, 0, 0 \\
1, 1, 1, 1, 1, 0 \\
1, 1, 1, 0, 0, 0 \\
1, 0, 0, 0, 0, 0
\end{bmatrix}
\]

Input (values)

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

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

Input (masks)

\[
\begin{bmatrix}
  1, 1, 0, 0, 0, 0 \\
  1, 1, 1, 1, 1, 0 \\
  1, 1, 1, 0, 0, 0 \\
  1, 0, 0, 0, 0, 0 \\
\end{bmatrix}
\]

Targets

\[
\begin{bmatrix}
  8, 5, 7, 0 \\
\end{bmatrix}
\]

Input (values)

\[
\begin{bmatrix}
  8, 5, 7, 0 \\
\end{bmatrix}
\]
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.
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

```python
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.

```python
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

```python
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(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
Training loop

```python
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.

Original | Input | Several synthesis
Some remarks:

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