Deep learning

10.1. Auto-regression

François Fleuret
https://fleuret.org/ee559/
Nov 2, 2020
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

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

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

Autoregressive methods are a very powerful tool to model densities in high dimensions. They come from the world of classical statistics and rely on a standard result of probability theory, the chain rule, which allows to compute the joint probability given conditional probabilities.

An autoregressive model visits the input signal in a given ordering and each component of the signal is modeled conditionally on the ones already modeled.

A deep autoregressive model is a deep model which models the distribution of one component given the conditioning over a bunch of them.
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 practice, implementing an event whose values are conditioned on others can be done using 2 tensors:

- a Boolean tensor (a mask tensor) which specifies which values are conditioned, and
- another tensor which contains the actually values.

The values of the value tensor are zeros on the unconditioned component (i.e. at the coordinates at which the mask are zeros).

For instance (following math notations where indices start at 1) with a signal of $T = 5$ components, event $\{X_5 = 50, X_2 = 20\}$ only depends on $X_5$ and $X_2$ and the mask tensor is [0, 1, 0, 0, 1], and the value tensor is [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

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

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 on $\mathbb{R}$.

In the following, the components of the signal will take a finite number of values $C$. The model outputs a tensor in $\mathbb{R}^C$, whose components can be interpreted as the probability of each value of the signal.

For instance with $C = 3$, each component of the signal can take out of 3 values. If the output is $[0.1, 0.5, 0.4]$, class 1 has a probability of 0.1 of being taken, class 2 of 0.5, and class 3 of 0.4.

An analogy with one hot encoding vector?

Given what has already been modeled, the function outputs the distribution over the next component. In practice, there is an underlying ordering of the components that is followed during training and testing which lead the masks tensors to be of the form $[1, -, 1, 0, -, 0]$

In the continuous case, the output of $f$ can the the parameters of the distribution $(\nu, \sigma) \in \mathbb{R}$ in the case of a Gaussian distribution.
Given such a model and a sampling procedure sample, the generative process for a full sequence is

\[
\begin{align*}
    x_1 & \leftarrow \text{sample}(f(\{})) \\
    x_2 & \leftarrow \text{sample}(f(\{X_1 = x_1\})) \\
    x_3 & \leftarrow \text{sample}(f(\{X_1 = x_1, X_2 = x_2\})) \\
    \ldots \\
    x_T & \leftarrow \text{sample}(f(\{X_1 = x_1, X_2 = x_2, \ldots, X_{T-1} = x_{T-1}\}))
\end{align*}
\]

A sampling procedure takes as input the probabilities (or logits) output by the model (a tensor in \(\mathbb{R}^C\)) and outputs the value of the component.

To generate a full signal:

- The generation of a full signal start with no conditioning at all: both the mask and value tensors are full of zeros. \(f(\{})\) is the output of the model with no conditioning at all. After sampling from the output \(f(\{})\), we obtain \(x_1\), the first value of the signal.
- Given the first component \(x_1\), we can now call \(f\) with this conditioning: the Boolean tensor is \([1, 0, \ldots, 0]\) and the value tensor is \([x_1, 0, \ldots, 0]\). \(f(\{X_1 = x_1\})\) return the distribution to sample the second component.
- etc.
With $C = 3$ and $T = 5$:

We consider the case of a signal with $T = 5$ components which can take $C = 3$ different values.

The generation process is the following:

- We start with an empty event which is encoded with mask $[0, 0, 0, 0, 0]$ and values $[0, 0, 0, 0, 0]$. The event is passed to model $f$ which outputs the distribution over the 3 classes $[0.1, 0.5, 0.4]$. The sampling over this distribution returns the sample value for the first component of the signal, here 2.

- Then, to generate the second component of the signal, we condition over the one just generated: the mask is $[1, 0, 0, 0, 0]$ and the value tensor is $[2, 0, 0, 0, 0]$. $f$ now outputs a new distribution over the possible values for the second component, here $[0.6, 0.2, 0.2]$. The sampling now returns the value for the second component, here 1.

- This process is repeated until reaching the end of the signal. To generate the final component, the mask will be $[1, 1, 0, 0, 0]$, and the value tensor for instance $[2, 1, 2, 0, 0]$.

The training process learns a model of the posterior distribution over the next component to predict.
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]])
```

In PyTorch, the sampling can be done in batch. In this case,

- the input is of size $(M, C)$. Each of the $M$ rows represent a discrete distribution over $C$ classes,
- the output is of size $(N, M)$: $N$ is the number of samples that we want to draw, $M$ the number of distributions. The values taken are in $[0, C - 1]$.
With a finite distribution and the output values interpreted as logits, training consists of maximizing the likelihood of the training samples, hence minimizing

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

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

where \( \ell \) is the cross-entropy.

The training consists in doing maximum likelihood by making the data as likely as possible under the model.

The loss is computed on all the components (sum on \( t \)) of all the training samples (sum on \( n \)): we want to maximize the (log) likelihood of a component \( (X_t = x_{n,t}) \) given what has already been predicted \( (X_1 = x_{n,1}, \ldots, X_{t-1} = x_{n,t-1}) \). In practice, we minimize the opposite of the likelihood.

Maximizing the probability of each component of each samples is formally equivalent to minimizing the cross-entropy between the predicted logits and the target value, over all the individual components.
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.

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

**Training sequences**

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

**Input**

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

**Masks**

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

**Values**

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

**Target**

In practice, the training can be done by minibatch which are generated on the fly:

- Given one sequence \([3, 1, 8, 1, 0, 3]\) which can be interpreted as a time series, here of length six, from left to right,
- we pick a random time step (3 for instance, with value 8)
- At time step 3, we know the values of the first two time steps, so the corresponding mask is \([1, 1, 0, 0, 0, 0]\) and the corresponding value tensor is \([3, 1, 0, 0, 0, 0]\).
- 8 is the value to predict at current time step, given the the past two time steps.
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 $\sim \mathcal{U}([-1, 1])$.

Each sequence contains $T$ components. Each component can take one of $C = 2T$ values.

A sequence starts at value $T$ ($x_1 = T$). The sequence behaves linearly with a slope in $[-1, 1]$ until a point at which we start again at value $T$ with a new slope in $[-1, 1]$. The breaking points is chosen uniformly in $[1, T]$.

The sketch shows an example where the two slopes are of opposite sign, but they may be of same sign (both increasing or both decreasing, as shown in the next illustration).
This toy example exhibits 2 interesting features:

- a macro structure (the cut), and
- in each part, a local relation (the slope).
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

The network used is a standard network. Note the 1d suffix for 1D function, as opposed to the 2D for images.

The input of the network has two dimensions: one for the mask, and one for the value tensor.

The output of the model is a tensor of size $C = 2T$ (variable nb_values).
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 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()
```

Note that the total numbers of sample may not be a multiple of the batch size. The last minibatch can therefore be smaller than the others.

Function `gather` groups in one single tensor the actual values of each row at the index selected to be the one to predict: the index is different at each row.

Remember that the sequences are normalized to start at $X_i = T$.

The dimension of `input` is `(nb, 2, 32)`, the length of the sequences being 32.
To synthesize one sequence, components are generated one after the other, each time passing the already generated components to the model.

- At each time step, the mask and the value tensor are created.
- At start, the mask (tics) is full of zeros.
- The mask and the value tensor are passed to the model which outputs the distribution for the next value to predict.
- The value at the current time step is drawn from this distribution.
The results of the generated sequences are quite satisfying. The model learned that before and after the cut, both parts 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.

```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)
```

In the case of a 2D image, one has to provide the order in which the components (the pixels) will be visited.

Images can be visited in “raster scan order” (or “row-wise”) which correspond to the underlying storage in memory, top-to-bottom and left-to-right.
Some of the MNIST train images

MNIST images are 28 × 28 gray-scale images. Pixels are in [0, 255], 0 being represented by white here, and 255 by black.

In this case, 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

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

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

In practice, we need a way to go from images to sequences (mainly for training), and from sequences to images (mainly for visualization after synthesis).

`seq2tensor` takes as input a tensor of dimension $N \times 784$, which is $N$ sequences of length 784, and outputs a tensor of size $N \times 1 \times 28 \times 28$ corresponding to the standard batch of MNIST image.

`tensor2seq` does the opposite and transforms the minibatch of $N$ MNIST images (size $N \times 1 \times 28 \times 28$) into a tensor of $N$ sequences (size $N \times 784$).
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()

The training goes as follows:

- As in the toy example with piece-wise linear function, the number of training samples may not be a multiple of the minibatch size.
- 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 values 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()

The synthesis procedure is very similar to the one in the to example case.
Although not close to the state-of-the-art models, 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, like 1, 6, 9.
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.

Each group of three images show:

- (left) The mask. Black areas are the pixels already generated and white ones are the ones not generated yet;
- (middle) The current image being drawn. The red dot is the location of the pixel which we want to predict the value at.
- (right) The distribution predicted by the model given the mask and the values generated so far. The x axis range from 0 to 255 which correspond to the gray level.

Below I describe the shape of the distribution but don’t refer explicitly to one in particular because the images on the video do not correspond to new images generated by `--compute`

When the distribution is somewhat decreasing, a white pixel is more likely to be generated, because low gray level values have a higher probability. When the distribution is flat, the network is uncertain on the gray level. This usually correspond to the interface between the white background of the image and the “digit” itself. When the distribution is increasing, the predicted pixel will be darker, and this corresponds to a location within a stroke.
The same generative process can be used for in-painting, by starting the process with available pixel values.

In-painting is the process to filling pixel values which have been removed. The in-painting algorithm has access to the rest of the image pixels. Here, the removed pixels are shown with the gray patch.

The main difference with the previous situation is that we don’t 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 do consistent completion.
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 visiting order of the components is arbitrary. The ordering can be natural in the case of time sequences, or if there are additional constrains, like for real-time applications.

Cross-entropy with categorical distribution on the discretized values works better than using a continuous modeling of the distribution. This is probably because:

- it gives more capacity to the model,
- the optimization is simpler, and
- categorical distributions and cross-entropy better deals with multi-modal distribution without explicit constrains.

The perplexity offers a way to interpret the number of values over which the model is hesitant. In the case of a uniform distribution over \( N \) values (which we say say is “hesitant” on the \( N \) values), each of them have a probability of \( \frac{1}{N} \) and the perplexity would be \( e^{-\log \frac{1}{N}} = N \).
References