Chapter 2
Autoregressive Models

2.1 Introduction

Before we start discussing how we can model the distribution $p(\mathbf{x})$, we refresh our memory about the core rules of probability theory, namely, the **sum rule** and the **product rule**. Let us introduce two random variables $\mathbf{x}$ and $\mathbf{y}$. Their joint distribution is $p(\mathbf{x}, \mathbf{y})$. The **product rule** allows us to *factorize* the joint distribution in two manners, namely:

$$p(\mathbf{x}, \mathbf{y}) = p(\mathbf{x}|\mathbf{y})p(\mathbf{y}) \tag{2.1}$$

$$= p(\mathbf{y}|\mathbf{x})p(\mathbf{x}). \tag{2.2}$$

In other words, the joint distribution could be represented as a product of a marginal distribution and a conditional distribution. The **sum rule** tells us that if we want to calculate the marginal distribution over one of the variables, we must integrate out (or sum out) the other variable, that is:

$$p(\mathbf{x}) = \sum_{\mathbf{y}} p(\mathbf{x}, \mathbf{y}). \tag{2.3}$$

These two rules will play a crucial role in probability theory and statistics and, in particular, in formulating deep generative models.

Now, let us consider a high-dimensional random variable $\mathbf{x} \in \mathcal{X}^D$ where $\mathcal{X} = \{0, 1, \ldots, 255\}$ (e.g., pixel values) or $\mathcal{X} = \mathbb{R}$. Our goal is to model $p(\mathbf{x})$. Before we jump into thinking of specific parameterization, let us first apply the product rule to express the joint distribution in a different manner:

$$p(\mathbf{x}) = p(x_1) \prod_{d=2}^{D} p(x_d|\mathbf{x}_{<d}), \tag{2.4}$$

J. M. Tomczak, *Deep Generative Modeling*,
https://doi.org/10.1007/978-3-030-93158-2_2

where $\mathbf{x}_{<d} = [x_1, x_2, \ldots, x_{d-1}]^\top$. For instance, for $\mathbf{x} = [x_1, x_2, x_3]^\top$, we have $p(\mathbf{x}) = p(x_1)p(x_2|x_1)p(x_3|x_1, x_2)$.

As we can see, the product rule applied multiple times to the joint distribution provides a principled manner of factorizing the joint distribution into many conditional distributions. That's great news! However, modeling all conditional distributions $p(x_d|\mathbf{x}_{<d})$ separately is simply infeasible! If we did that, we would obtain D separate models, and the complexity of each model would grow due to varying conditioning. A natural question is whether we can do better, and the answer is yes.

2.2 Autoregressive Models Parameterized by Neural Networks

As mentioned earlier, we aim for modeling the joint distribution $p(\mathbf{x})$ using conditional distributions. A potential solution to the issue of using D separate model is utilizing a single, shared model for the conditional distribution. However, we need to make some assumptions to use such a shared model. In other words, we look for an **autoregressive model** (ARM). In the next subsection, we outline ARMs parameterized with various *neural networks*. After all, we are talking about deep generative models so using a neural network is not surprising, isn't it?

2.2.1 Finite Memory

The first attempt to limiting the complexity of a conditional model is to assume a *finite memory*. For instance, we can assume that each variable is dependent on no more than two other variables, namely:

$$p(\mathbf{x}) = p(x_1)p(x_2|x_1)\prod_{d=3}^{D} p(x_d|x_{d-1}, x_{d-2}). \tag{2.5}$$

Then, we can use a small neural network, e.g., multi-layered perceptron (MLP), to predict the distribution of x_d. If $\mathcal{X} = \{0, 1, \ldots, 255\}$, the MLP takes x_{d-1}, x_{d-2} and outputs probabilities for the categorical distribution of x_d, θ_d. The MLP could be of the following form:

$$[x_{d-1}, x_{d-2}] \rightarrow \text{Linear}(2, M) \rightarrow \text{ReLU} \rightarrow \text{Linear}(M, 256) \rightarrow \text{softmax} \rightarrow \theta_d, \tag{2.6}$$

where M denotes the number of hidden units, e.g., $M = 300$. An example of this approach is depicted in Fig. 2.1.

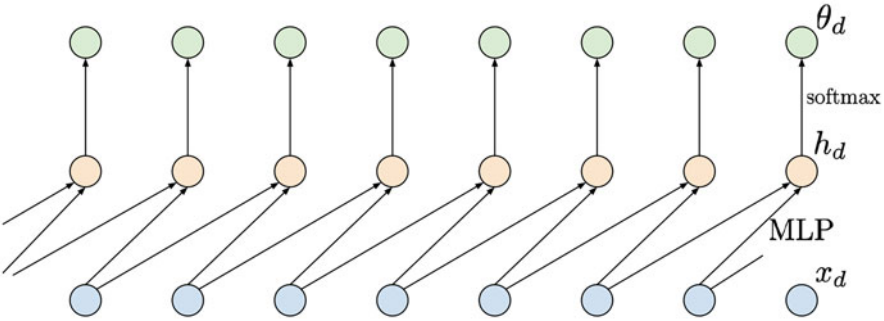

Fig. 2.1 An example of applying a shared MLP depending on two last inputs. Inputs are denoted by blue nodes (bottom), intermediate representations are denoted by orange nodes (middle), and output probabilities are denoted by green nodes (top). Notice that a probability θ_d is **not** dependent on x_d

It is important to notice that now we use a single, shared MLP to predict probabilities for x_d. Such a model is not only non-linear but also its parameterization is convenient due to a relatively small number of weights to be trained. However, the obvious drawback of this approach is a **limited memory** (i.e., only two last variables in our example). Moreover, it is unclear a priori how many variables we should use in conditioning. In many problems, e.g., image processing, learning *long-range statistics* is crucial to understand complex patterns in data; therefore, having long-range memory is essential.

2.2.2 Long-Range Memory Through RNNs

A possible solution to the problem of a short-range memory modeled by an MLP relies on applying a recurrent neural network (RNN) [1, 2]. In other words, we can model the conditional distributions as follows [3]:

$$p(x_d|\mathbf{x}_{<d}) = p\left(x_d|\text{RNN}(x_{d-1}, h_{d-1})\right), \tag{2.7}$$

where $h_d = \text{RNN}(x_{d-1}, h_{d-1})$, and h_d is a hidden context, which acts as a *memory* that allows learning long-range dependencies. An example of using an RNN is presented in Fig. 2.2.

This approach gives a single parameterization, thus, it is efficient and also solves the problem of a finite memory. So far so good! Unfortunately, RNNs suffer from other issues, namely:

- They are sequential, hence, slow.

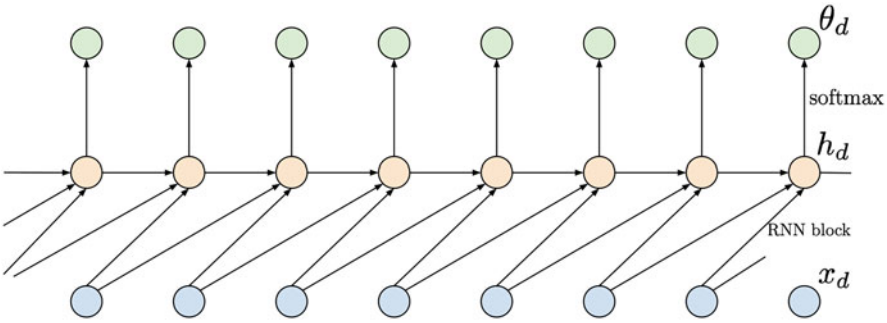

Fig. 2.2 An example of applying an RNN depending on two last inputs. Inputs are denoted by blue nodes (bottom), intermediate representations are denoted by orange nodes (middle), and output probabilities are denoted by green nodes (top). Notice that compared to the approach with a shared MLP, there is an additional dependency between intermediate nodes h_d

- If they are badly conditioned (i.e., the eigenvalues of a weight matrix are larger or smaller than 1, then they suffer from exploding or vanishing gradients, respectively, that hinders learning long-range dependencies.

There exist methods to help training RNNs like gradient clipping or, more generally, gradient regularization [4] or orthogonal weights [5]. However, here we are not interested in looking into rather specific solutions to new problems. We seek for a different parameterization that could solve our original problem, namely, modeling long-range dependencies in an ARM.

2.2.3 Long-Range Memory Through Convolutional Nets

In [6, 7] it was noticed that convolutional neural networks (CNNs) could be used instead of RNNs to model long-range dependencies. To be more precise, one-dimensional convolutional layers (Conv1D) could be stacked together to process sequential data. The advantages of such an approach are the following:

- Kernels are shared (i.e., an efficient parameterization).
- The processing is done in parallel that greatly speeds up computations.
- By stacking more layers, the effective kernel size grows with the network depth.

These three traits seem to place Conv1D-based neural networks as a perfect solution to our problem. However, can we indeed use them straight away?

A Conv1D can be applied to calculate embeddings like in [7], but it cannot be used for autoregressive models. Why? Because we need convolutions to be **causal** [8]. *Causal* in this context means that a Conv1D layer is dependent on the last k inputs but the current one (*option A*) or with the current one (*option B*). In other words, we must "cut" the kernel in half and forbid it to look into the next variables

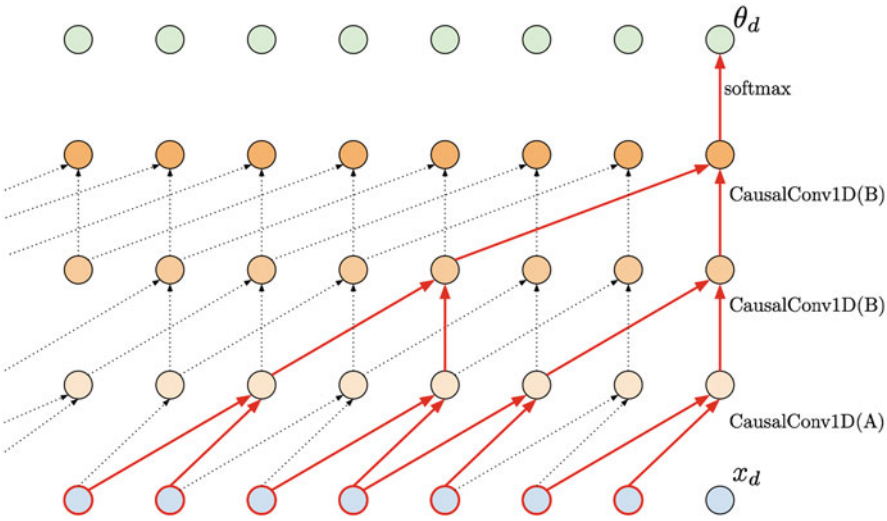

Fig. 2.3 An example of applying causal convolutions. The kernel size is 2, but by applying dilation in higher layers, a much larger input could be processed (red edges), thus, a larger memory is utilized. Notice that the first layers must be option A to ensure proper processing

(look into the future). Importantly, the option A is required in the first layer because the final output (i.e., the probabilities θ_d) cannot be dependent on x_d. Additionally, if we are concerned about the effective kernel size, we can use *dilation* larger than 1.

In Fig. 2.3 we present an example of a neural network consisting of 3 *causal Conv1D* layers. The first CausalConv1D is of type A, i.e., it does not take into account only the last k inputs without the current one. Then, in the next two layers, we use CausalConv1D (option B) with dilations 2 and 3. Typically, the dilation values are 1, 2, 4, and 8 (v.d. Oord et al., 2016a); however, taking 2 and 4 would not nicely fit in a figure. We highlight in red all connections that go from the output layer to the input layer. As we can notice, stacking CausalConv1D layers with the dilation larger than 1 allows us to learn long-range dependencies (in this example, by looking at 7 last inputs).

An example of an implementation of CausalConv1D layer is presented below. If you are still confused about the option A and the option B, please analyze the code snippet step-by-step.

```
class CausalConv1d(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size,
    dilation, A=False, **kwargs):
        super(CausalConv1d, self).__init__()

        # The general idea is the following: We take the built-in
     PyTorch Conv1D. Then, we must pick a proper padding, because
     we must ensure the convolutional is causal. Eventually, we
```

```
    must remove some final elements of the output, because we
    simply don't need them! Since CausalConv1D is still a
    convolution, we must define the kernel size, dilation, and
    whether it is option A (A=True) or option B (A=False).
    Remember that by playing with dilation we can enlarge the
    size of the memory.

        # attributes:
        self.kernel_size = kernel_size
        self.dilation = dilation
        self.A = A # whether option A (A=True) or B (A=False)
        self.padding = (kernel_size - 1) * dilation + A * 1

        # we will do padding by ourselves in the forward pass!
        self.conv1d = torch.nn.Conv1d(in_channels, out_channels,
                                      kernel_size, stride=1,
                                      padding=0,
                                      dilation=dilation,**kwargs)

    def forward(self, x):
        # We do padding only from the left! This is more
    efficient implementation.
        x = torch.nn.functional.pad(x, (self.padding, 0))
        conv1d_out = self.conv1d(x)
        if self.A:
            # Remember, we cannot be dependent on the current
    component; therefore, the last element is removed.
            return conv1d_out[:, :, :-1]
        else:
            return conv1d_out
```

Listing 2.1 Causal convolution 1D

The CausalConv1D layers are better-suited to modeling sequential data than RNNs. They obtain not only better results (e.g., classification accuracy) but also allow learning long-range dependencies more efficiently than RNNs [8]. Moreover, they do not suffer from exploding/vanishing gradient issues. As a result, they seem to be a perfect parameterization for autoregressive models! Their supremacy has been proven in many cases, including audio processing by WaveNet, a neural network consisting of CausalConv1D layers [9], or image processing by PixelCNN, a model with CausalConv2D components [10].

Then, is there any drawback of applying autoregressive models parameterized by causal convolutions? Unfortunately, yes, there is and it is connected with sampling. If we want to evaluate probabilities for given inputs, we need to calculate the forward pass where all calculations are done in parallel. However, if we want to sample new objects, we must iterate through all positions (think of a big for-loop, from the first variable to the last one) and iteratively predict probabilities and sample new values. Since we use convolutions to parameterize the model, we must do D full forward passes to get the final sample. That is a big waste, but, unfortunately, that is the price we must pay for all "goodies" following from the convolutional-based

parameterization of the ARM. Fortunately, there is on-going research on speeding up computations, e.g., see [11].

2.3 Deep Generative Autoregressive Model in Action!

Alright, let us talk more about details and how to implement an ARM. Here, and in the whole book, we focus on images, e.g., $\mathbf{x} \in \{0, 1, \ldots, 15\}^{64}$. Since images are represented by integers, we will use the categorical distribution to represent them (in next chapters, we will comment on the choice of distribution for images and present some alternatives). We model $p(\mathbf{x})$ using an ARM parameterized by CausalConv1D layers. As a result, each conditional is the following:

$$p(x_d|\mathbf{x}_{<d}) = \text{Categorical}\left(x_d|\theta_d\left(\mathbf{x}_{<d}\right)\right) \tag{2.8}$$

$$= \prod_{l=1}^{L} \left(\theta_{d,l}\right)^{[x_d=l]}, \tag{2.9}$$

where $[a = b]$ is the Iverson bracket (i.e., $[a = b] = 1$ if $a = b$, and $[a = b] = 0$ if $a \neq b$), and $\theta_d\left(\mathbf{x}_{<d}\right) \in [0, 1]^{16}$ is the output of the CausalConv1D-based neural network with the softmax in the last layer, so $\sum_{l=1}^{L} \theta_{d,l} = 1$. To be very clear, the last layer must have 16 output channels (because there are 16 possible values per pixel), and the softmax is taken over these 16 values. We stack CausalConv1D layers with non-linear activation functions in between (e.g., LeakyRELU). Of course, we must remember about taking the option A CausalConv1D as the first layer! Otherwise we break the assumption about taking into account x_d in predicting θ_d.

What about the objective function? ARMs are the likelihood-based models, so for given N i.i.d. datapoints $\mathcal{D} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$, we aim at maximizing the logarithm of the likelihood function, that is (we will use the product and sum rules again):

$$\ln p(\mathcal{D}) = \ln \prod_n p(\mathbf{x}_n) \tag{2.10}$$

$$= \sum_n \ln p(\mathbf{x}_n) \tag{2.11}$$

$$= \sum_n \ln \prod_d p(x_{n,d}|\mathbf{x}_{n,<d}) \tag{2.12}$$

$$= \sum_n \left(\sum_d \ln p(x_{n,d}|\mathbf{x}_{n,<d}) \right) \tag{2.13}$$

$$= \sum_n \left(\sum_d \ln \text{Categorical}\left(x_d|\theta_d\left(\mathbf{x}_{<d}\right)\right) \right) \tag{2.14}$$

$$= \sum_n \left(\sum_d \left(\sum_{l=1}^{L} [x_d = l] \ln \theta_d \left(\mathbf{x}_{<d} \right) \right) \right). \tag{2.15}$$

For simplicity, we assumed that $\mathbf{x}_{<1} = \emptyset$, i.e., no conditioning. As we can notice, the objective function takes a very nice form! First, the logarithm over the i.i.d. data $\mathcal{D}$ results in a sum over datapoints of the logarithm of individual distributions $p(\mathbf{x}_n)$. Second, applying the product rule, together with the logarithm, results in another sum, this time over dimensions. Eventually, by parameterizing the conditionals by CausalConv1D, we can calculate all θ_d in one forward pass and then check the pixel value (see the last line of $\ln p(\mathcal{D})$). Ideally, we want $\theta_{d,l}$ to be as close to 1 as possible if $x_d = l$.

2.3.1 Code

Uff... Alright, let's take a look at some code. The full code is available under the following: https://github.com/jmtomczak/intro_dgm. Here, we focus only on the code for the model. We provide details in the comments.

```
class ARM(nn.Module):
    def __init__(self, net, D=2, num_vals=256):
        super(ARM, self).__init__()

        # Remember, always credit the author, even if it's you ;)
        print('ARM by JT.')

        # This is a definition of a network. See the next cell.
        self.net = net
        # This is how many values a pixel can take.
        self.num_vals = num_vals
        # This is the problem dimentionality (the number of
    pixels)
        self.D = D

    # This function calculates the arm output.
    def f(self, x):
        # First, we apply causal convolutions.
        h = self.net(x.unsqueeze(1))
        # In channels, we have the number of values. Therefore,
    we change the order of dims.
        h = h.permute(0, 2, 1)
        # We apply softmax to calculate probabilities.
        p = torch.softmax(h, 2)
        return p

    # The forward pass calculates the log-probability of an image
    .
    def forward(self, x, reduction='avg'):
        if reduction == 'avg':
```

```
            return -(self.log_prob(x).mean())
        elif reduction == 'sum':
            return -(self.log_prob(x).sum())
        else:
            raise ValueError('reduction could be either `avg` or
    `sum`.')

    # This function calculates the log-probability (log-
    categorical).
    # See the full code in the separate file for details.
    def log_prob(self, x):
        mu_d = self.f(x)
        log_p = log_categorical(x, mu_d, num_classes=self.
    num_vals, reduction='sum', dim=-1).sum(-1)

        return log_p

    # This function implements sampling procedure.
    def sample(self, batch_size):
        # As you can notice, we first initialize a tensor with
    zeros.
        x_new = torch.zeros((batch_size, self.D))

        # Then, iteratively, we sample a value for a pixel.
        for d in range(self.D):
            p = self.f(x_new)
            x_new_d = torch.multinomial(p[:, d, :], num_samples
    =1)
            x_new[:, d] = x_new_d[:,0]

        return x_new
```

Listing 2.2 Autoregressive model parameterized by causal convolutions 1D

```
# An example of a network. NOTICE: The first layer is A=True,
    while all the others are A=False.
# At this point we should know already why :)
M = 256

net = nn.Sequential(
    CausalConv1d(in_channels=1, out_channels=MM, dilation=1,
    kernel_size=kernel, A=True, bias=True),
    nn.LeakyReLU(),
    CausalConv1d(in_channels=MM, out_channels=MM, dilation=1,
    kernel_size=kernel, A=False, bias=True),
    nn.LeakyReLU(),
    CausalConv1d(in_channels=MM, out_channels=MM, dilation=1,
    kernel_size=kernel, A=False, bias=True),
    nn.LeakyReLU(),
    CausalConv1d(in_channels=MM, out_channels=num_vals, dilation
    =1, kernel_size=kernel, A=False, bias=True))
```

Listing 2.3 An example of a network

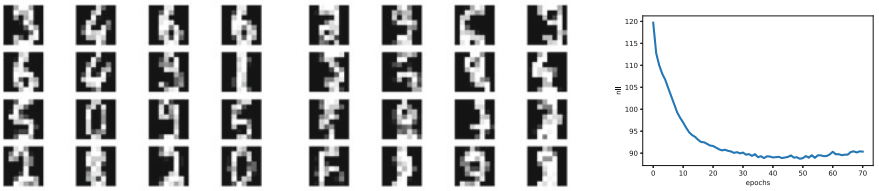

Fig. 2.4 An example of outcomes after the training: (**a**) Randomly selected real images. (**b**) Unconditional generations from the ARM. (**c**) The validation curve during training

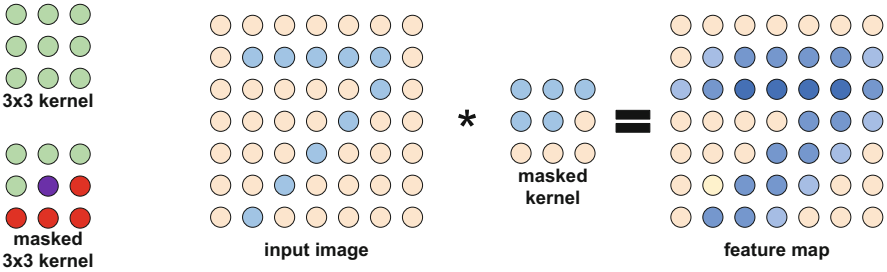

Fig. 2.5 An example of a masked 3×3 kernel (i.e., a causal 2D kernel): (*left*) A difference between a standard kernel (all weights are used; denoted by green) and a masked kernel (some weights are masked, i.e., not used; in red). For the masked kernel, we denoted the node (pixel) in the middle in violet, because it is either masked (option A) or not (option B). (*middle*) An example of an image (light orange nodes: zeros, light blue nodes: ones) and a masked kernel (option A). (*right*) The result of applying the masked kernel to the image (with padding equal to 1)

Perfect! Now we are ready to run the full code. After training our ARM, we should obtain results similar to those in Fig. 2.4.

2.4 Is It All? No!

First of all, we discussed one-dimensional causal convolutions that are typically insufficient for modeling images due to their spatial dependencies in 2D (or 3D if we consider more than 1 channel; for simplicity, we focus on a 2D case). In [10], a CausalConv2D was proposed. The idea is similar to that discussed so far, but now we need to ensure that the kernel will not look into future pixels in both the x-axis and y-axis. In Fig. 2.5, we present the difference between a standard kernel where all kernel weights are used and a masked kernel with some weights zeroed-out (or masked). Notice that in CausalConv2D we must also use option A for the first layer (i.e., we skip the pixel in the middle) and we can pick option B for the remaining layers. In Fig. 2.6, we present the same example as in Fig. 2.5 but using numeric values.

Fig. 2.6 The same example as in Fig. 2.5 but with numeric values

$$\begin{bmatrix} 0&0&0&0&0&0&0\\ 0&1&1&1&1&1&0\\ 0&0&0&0&0&1&0\\ 0&0&0&0&1&0&0\\ 0&0&0&1&0&0&0\\ 0&0&1&0&0&0&0\\ 0&1&0&0&0&0&0 \end{bmatrix} * \begin{bmatrix} 1&1&1\\ 1&1&0\\ 0&0&0 \end{bmatrix} = \begin{bmatrix} 0&0&0&0&0&0&0\\ 0&1&2&2&2&1&0\\ 1&2&3&3&3&3&2\\ 0&0&0&0&2&2&1\\ 0&0&0&2&2&1&0\\ 0&0&2&2&1&0&0\\ 0&2&2&1&0&0&0 \end{bmatrix}$$

In [12], the authors propose a further improvement on the causal convolutions. The main idea relies on creating a block that consists of vertical and horizontal convolutional layers. Moreover, they use gated non-linearity function, namely:

$$\mathbf{h} = \tanh(\mathbf{W}\mathbf{x}) \odot \sigma(\mathbf{V}\mathbf{x}). \tag{2.16}$$

See Figure 2 in [12] for details.

Further improvements on ARMs applied to images are presented in [13]. Therein, the authors propose to replace the categorical distribution used for modeling pixel values with the discretized logistic distribution. Moreover, they suggest to use a mixture of discretized logistic distributions to further increase flexibility of their ARMs.

The introduction of the causal convolution opened multiple opportunities for deep generative modeling and allowed obtaining state-of-the-art generations and density estimations. It is impossible to review all papers here, we just name a few interesting directions/applications that are worth remembering:

- An alternative ordering of pixels was proposed in [14]. Instead of using the ordering from left to right, a "zig–zag" pattern was proposed that allows pixels to depend on pixels previously sampled to the left and above.
- ARMs could be used as stand-alone models or they can be used in a combination with other approaches. For instance, they can be used for modeling a prior in the (Variational) Auto-Encoders [15].
- ARMs could be also used to model videos [16]. Factorization of sequential data like video is very natural, and ARMs fit this scenario perfectly.
- A possible drawback of ARMs is a lack of latent representation because all conditionals are modeled explicitly from data. To overcome this issue, [17] proposed to use a PixelCNN-based decoder in a Variational Auto-Encoder.
- An interesting and important research direction is about proposing new architectures/components of ARMs or speeding them up. As mentioned earlier, sampling from ARMs could be slow, but there are ideas to improve on that by predictive sampling [11, 18].
- Alternatively, we can replace the likelihood function with other similarity metrics, e.g., the Wasserstein distance between distributions as in quantile regression. In the context of ARMs, quantile regression was applied in [19], requiring only minor architectural changes, that resulted in improved quality scores.