Deep learning

10.2. Causal convolutions

François Fleuret

https://fleuret.org/dlc/

UNIVERSITÉ DE GENÈVE

If we use an autoregressive model with a **masked input** as we saw in lecture 10.1. "Auto-regression"

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

the input differs from a position to another.

During training, even though the full sequence is known, common computation is lost.

Instead of predicting [the distribution of] one component, the model could predict [the distributions] at every position of the sequence, that is

$$f : \mathbb{R}^T \to \mathbb{R}^{T \times C}.$$

It can be used for synthesis with

$$\begin{aligned}
x_1 &\leftarrow \text{sample}\left(f_1(0, \dots, 0)\right) \\
x_2 &\leftarrow \text{sample}\left(f_2(x_1, 0, \dots, 0)\right) \\
x_3 &\leftarrow \text{sample}\left(f_3(x_1, x_2, 0, \dots, 0)\right) \\
&\dots \\
x_T &\leftarrow \text{sample}\left(f_T(x_1, x_2, \dots, x_{T-1}, 0)\right)
\end{aligned}$$

where the 0s simply fill in for unknown values, and the mask is not needed.

If additionally, the model is such that "future values" do not influence the prediction at a certain time, that is

$$\forall t, x_1, \dots, x_t, \alpha_1, \dots, \alpha_{T-t}, \beta_1, \dots, \beta_{T-t},$$
$$f_{t+1}(x_1, \dots, x_t, \alpha_1, \dots, \alpha_{T-t}) = f_{t+1}(x_1, \dots, x_t, \beta_1, \dots, \beta_{T-t})$$

then, we have in particular

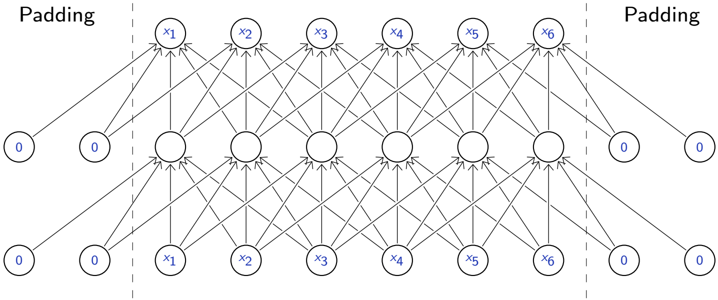
$$\begin{aligned}
f_1(0, \dots, 0) &= f_1(x_1, \dots, x_T) \\
f_2(x_1, 0, \dots, 0) &= f_2(x_1, \dots, x_T) \\
f_3(x_1, x_2, 0, \dots, 0) &= f_3(x_1, \dots, x_T) \\
&\dots \\
f_T(x_1, x_2, \dots, x_{T-1}, 0) &= f_T(x_1, \dots, x_T)
\end{aligned}$$

This provides a tremendous computational advantage during training, since all the $f_t(x_1, \dots, x_T)$ can be computed with a single forward pass:

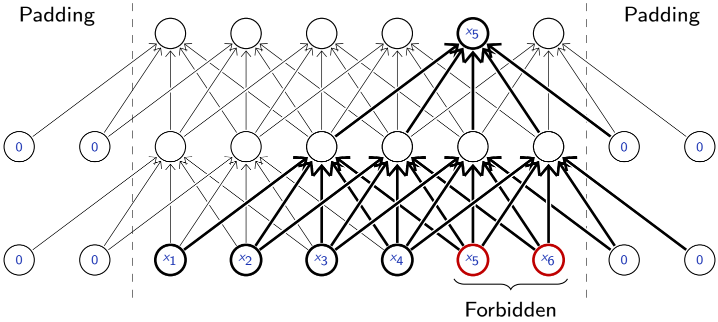
$$\ell(f, x) = \sum_t \ell(f_t(x_1, \dots, x_{t-1}, 0, \dots, 0), x_t)$$
$$= \sum_t \ell(\underbrace{f_t(x_1, \dots, x_T)}_{f \text{ is computed once}}, x_t).$$

Such models are referred to as **causal,** since the future cannot affect the past.

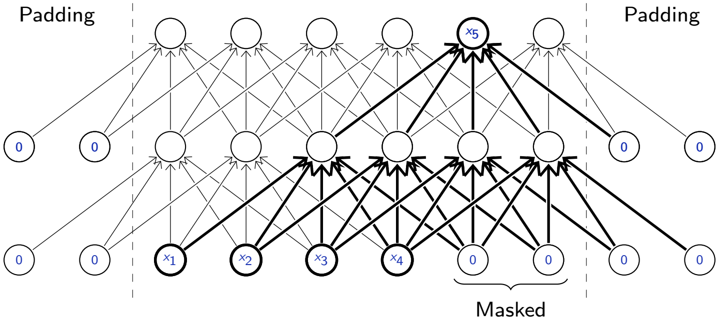
We can illustrate this with convolutional models. Standard convolutions let information flow "to the past," and masked input was a way to condition only on already generated values.



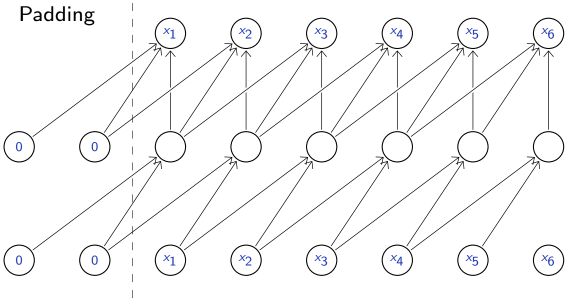
We can illustrate this with convolutional models. Standard convolutions let information flow "to the past," and masked input was a way to condition only on already generated values.



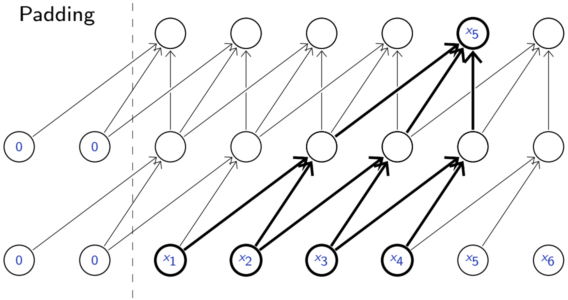
We can illustrate this with convolutional models. Standard convolutions let information flow "to the past," and masked input was a way to condition only on already generated values.



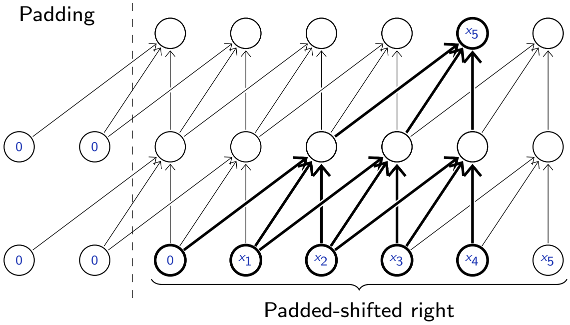
Such a model can be made **causal** with convolutions that let information flow only to the future, combined with a first convolution that hides the present.



Such a model can be made **causal** with convolutions that let information flow only to the future, combined with a first convolution that hides the present.



Another option for the first layer is to shift the input by one entry to hide the present.



PyTorch's convolutional layers do no accept asymmetric padding, but we can do it with `F.pad`, which even accepts negative padding to remove entries.

For a n-dim tensor, the padding specification is

$$(start_n, end_n, start_{n-1}, end_{n-1}, \ldots, start_{n-k}, end_{n-k})$$

```
>>> x = torch.randint(10, (2, 1, 5))
>>> x
tensor([[[1, 6, 3, 9, 1]],
        [[4, 8, 2, 2, 9]]])
>>> F.pad(x, (-1, 1))
tensor([[[6, 3, 9, 1, 0]],
        [[8, 2, 2, 9, 0]]])
>>> F.pad(x, (0, 0, 2, 0))
tensor([[[0, 0, 0, 0, 0],
         [0, 0, 0, 0, 0],
         [1, 6, 3, 9, 1]],
        [[0, 0, 0, 0, 0],
         [0, 0, 0, 0, 0],
         [4, 8, 2, 2, 9]]])
```

PyTorch's convolutional layers do no accept asymmetric padding, but we can do it with `F.pad`, which even accepts negative padding to remove entries.

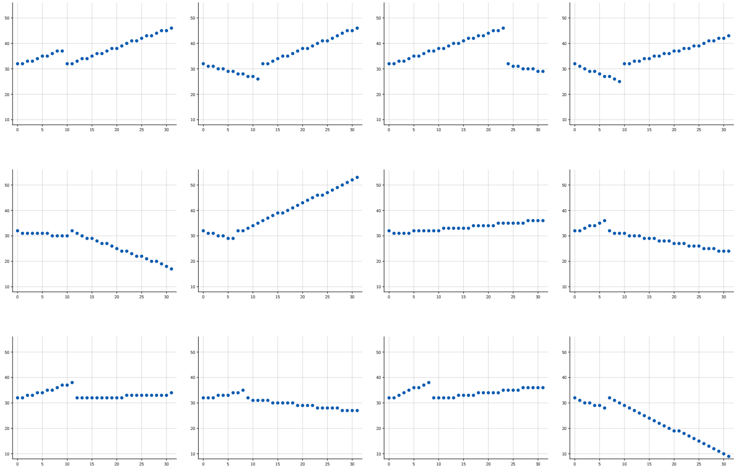
For a n-dim tensor, the padding specification is

$$(start_n, end_n, start_{n-1}, end_{n-1}, \dots, start_{n-k}, end_{n-k})$$

```
>>> x = torch.randint(10, (2, 1, 5))
>>> x
tensor([[[1, 6, 3, 9, 1]],
        [[4, 8, 2, 2, 9]]])
>>> F.pad(x, (-1, 1))
tensor([[[6, 3, 9, 1, 0]],
        [[8, 2, 2, 9, 0]]])
>>> F.pad(x, (0, 0, 2, 0))
tensor([[[0, 0, 0, 0, 0],
         [0, 0, 0, 0, 0],
         [1, 6, 3, 9, 1]],
        [[0, 0, 0, 0, 0],
         [0, 0, 0, 0, 0],
         [4, 8, 2, 2, 9]]])
```

Similar processing can be achieved with the modules `nn.ConstantPad1d`, `nn.ConstantPad2d`, or `nn.ConstantPad3d`.

Here some train sequences as in lecture 10.1. “Auto-regression”.



Model

```
class NetToy1d(nn.Module):
    def __init__(self, nb_classes, ks = 2, nc = 32):
        super().__init__()
        self.pad = (ks - 1, 0)
        self.conv0 = nn.Conv1d(1, nc, kernel_size = 1)
        self.conv1 = nn.Conv1d(nc, nc, kernel_size = ks)
        self.conv2 = nn.Conv1d(nc, nc, kernel_size = ks)
        self.conv3 = nn.Conv1d(nc, nc, kernel_size = ks)
        self.conv4 = nn.Conv1d(nc, nc, kernel_size = ks)
        self.conv5 = nn.Conv1d(nc, nb_classes, kernel_size = 1)

    def forward(self, x):
        x = F.relu(self.conv0(F.pad(x, (1, -1))))
        x = F.relu(self.conv1(F.pad(x, self.pad)))
        x = F.relu(self.conv2(F.pad(x, self.pad)))
        x = F.relu(self.conv3(F.pad(x, self.pad)))
        x = F.relu(self.conv4(F.pad(x, self.pad)))
        x = self.conv5(x)
        return x.permute(0, 2, 1).contiguous()
```

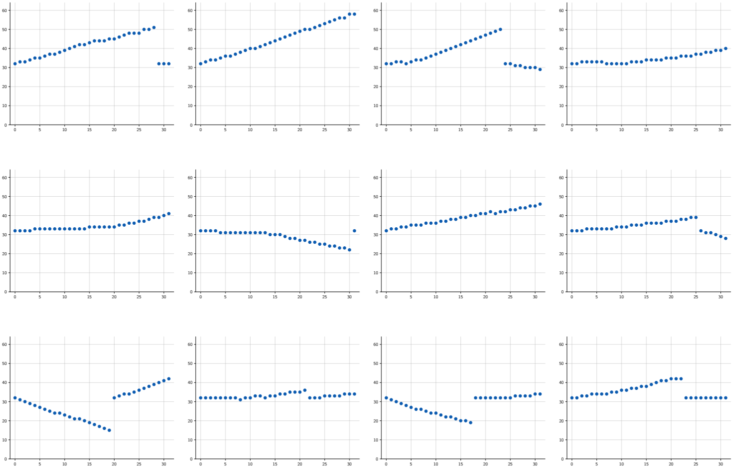
Training loop

```
for sequences in train_input.split(args.batch_size):
    input = (sequences - mean)/std

    output = model(input)

    loss = cross_entropy(
        output.view(-1, output.size(-1)),
        sequences.view(-1)
    )

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

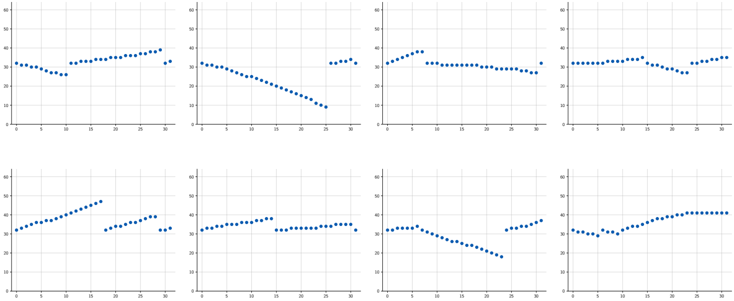
Synthesis

```
generated = train_input.new_zeros((48,) + train_input.size()[1:])

flat = generated.view(generated.size(0), -1)

for t in range(flat.size(1)):
    input = (generated.float() - mean) / std
    output = model(input)
    logits = output.view(flat.size() + (-1,))[:, t]
    dist = torch.distributions.categorical.Categorical(logits = logits)
    flat[:, t] = dist.sample()
```

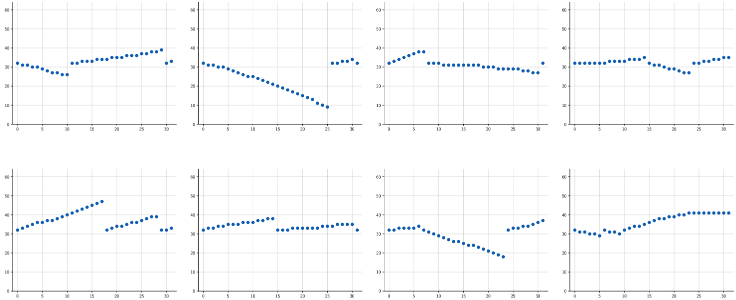
Some generated sequences



The global structure may not be properly generated.



The global structure may not be properly generated.

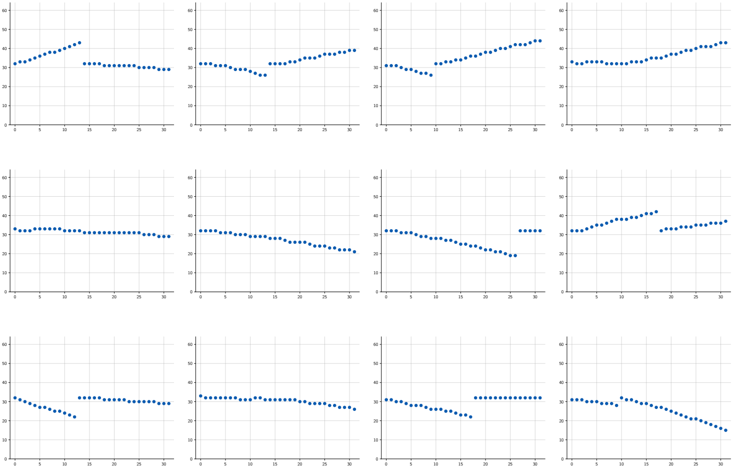


This can be fixed with **dilated convolutions** to have a larger context.

Model

```
class NetToy1dWithDilation(nn.Module):
    def __init__(self, nb_classes, ks = 2, nc = 32):
        super().__init__()
        self.conv0 = nn.Conv1d(1, nc, kernel_size = 1)
        self.pad1 = ((ks-1) * 2, 0)
        self.conv1 = nn.Conv1d(nc, nc, kernel_size = ks, dilation = 2)
        self.pad2 = ((ks-1) * 4, 0)
        self.conv2 = nn.Conv1d(nc, nc, kernel_size = ks, dilation = 4)
        self.pad3 = ((ks-1) * 8, 0)
        self.conv3 = nn.Conv1d(nc, nc, kernel_size = ks, dilation = 8)
        self.pad4 = ((ks-1) * 16, 0)
        self.conv4 = nn.Conv1d(nc, nc, kernel_size = ks, dilation = 16)
        self.conv5 = nn.Conv1d(nc, nb_classes, kernel_size = 1)

    def forward(self, x):
        x = F.relu(self.conv0(F.pad(x, (1, -1))))
        x = F.relu(self.conv1(F.pad(x, self.pad1)))
        x = F.relu(self.conv2(F.pad(x, self.pad2)))
        x = F.relu(self.conv3(F.pad(x, self.pad3)))
        x = F.relu(self.conv4(F.pad(x, self.pad4)))
        x = self.conv5(x)
        return x.permute(0, 2, 1).contiguous()
```

Some generated sequences



The WaveNet model proposed by Oord et al. (2016a) for voice synthesis relies in large part on such an architecture.

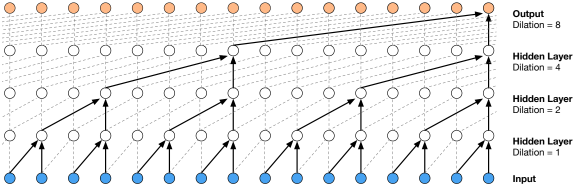


Figure 3: Visualization of a stack of *dilated* causal convolutional layers.

(Oord et al., 2016a)

Causal convolutions for images

The same mechanism can be implemented for images, using causal convolution:

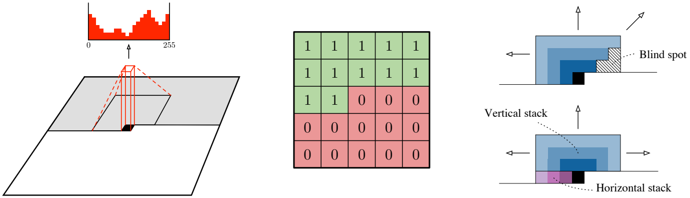
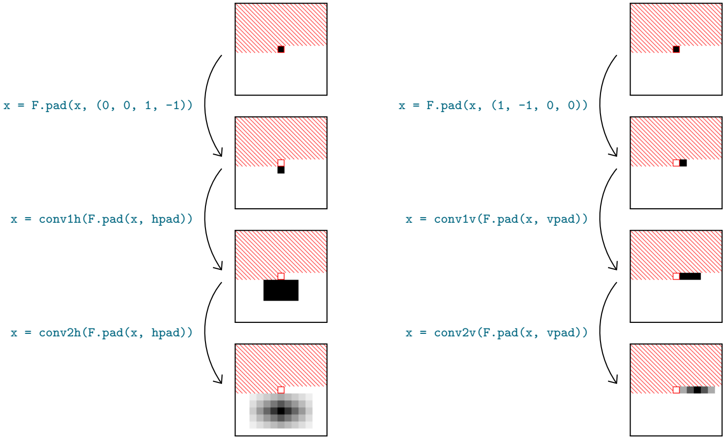


Figure 1: **Left**: A visualization of the PixelCNN that maps a neighborhood of pixels to prediction for the next pixel. To generate pixel x_i the model can only condition on the previously generated pixels $x_1, \dots x_{i-1}$. **Middle**: an example matrix that is used to mask the 5x5 filters to make sure the model cannot read pixels below (or strictly to the right) of the current pixel to make its predictions. **Right**: Top: PixelCNNs have a *blind spot* in the receptive field that can not be used to make predictions. Bottom: Two convolutional stacks (blue and purple) allow to capture the whole receptive field.

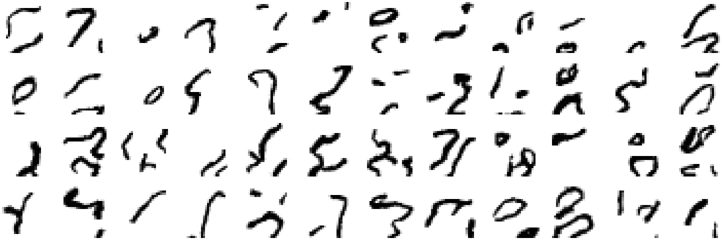
(Oord et al., 2016b)

```
ks = 5
hpad = (ks//2, ks//2, ks//2, 0)
conv1h = nn.Conv2d(1, 1, kernel_size = (ks//2+1, ks))
conv2h = nn.Conv2d(1, 1, kernel_size = (ks//2+1, ks))
vpad = (ks//2,     0,     0, 0)
conv1v = nn.Conv2d(1, 1, kernel_size = (1, ks//2+1))
conv2v = nn.Conv2d(1, 1, kernel_size = (1, ks//2+1))
```



```
class PixelCNN(nn.Module):
    def __init__(self, nb_classes, in_channels = 1, ks = 5):
        super().__init__()

        self.hpad = (ks//2, ks//2, ks//2, 0)
        self.vpad = (ks//2,     0,     0, 0)

        self.conv1h = nn.Conv2d(in_channels, 32, kernel_size = (ks//2+1, ks))
        self.conv2h = nn.Conv2d(32, 64, kernel_size = (ks//2+1, ks))
        self.conv1v = nn.Conv2d(in_channels, 32, kernel_size = (1, ks//2+1))
        self.conv2v = nn.Conv2d(32, 64, kernel_size = (1, ks//2+1))
        self.final1 = nn.Conv2d(128, 128, kernel_size = 1)
        self.final2 = nn.Conv2d(128, nb_classes, kernel_size = 1)

    def forward(self, x):
        xh = F.pad(x, (0, 0, 1, -1))
        xv = F.pad(x, (1, -1, 0, 0))
        xh = F.relu(self.conv1h(F.pad(xh, self.hpad)))
        xv = F.relu(self.conv1v(F.pad(xv, self.vpad)))
        xh = F.relu(self.conv2h(F.pad(xh, self.hpad)))
        xv = F.relu(self.conv2v(F.pad(xv, self.vpad)))
        x = F.relu(self.final1(torch.cat((xh, xv), 1)))
        x = self.final2(x)

        return x.permute(0, 2, 3, 1).contiguous()
```

Some generated images



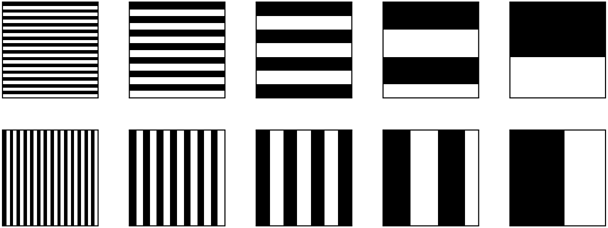
Such a fully convolutional model has no way to make the prediction position-dependent, which results here in local consistency, but fragmentation.

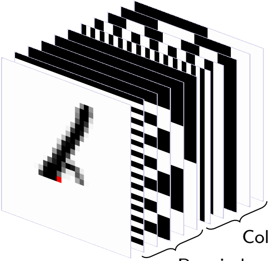
A classical fix is to supplement the input with a **positional encoding,** that is a multi-channel input that provides full information about the location.

Such a fully convolutional model has no way to make the prediction position-dependent, which results here in local consistency, but fragmentation.

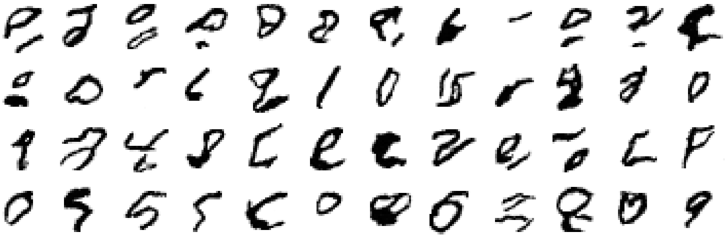
A classical fix is to supplement the input with a **positional encoding,** that is a multi-channel input that provides full information about the location.

Here with a resolution of 28×28 we can encode the positions with 5 Boolean channels per coordinate.




Input tensor with
positional encoding
Column index encoding
Row index encoding

Some generated images



The end

References

A. Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu. **WaveNet: A generative model for raw audio**. CoRR, abs/1609.03499, 2016a.

A. Oord, N. Kalchbrenner, O. Vinyals, L. Espeholt, A. Graves, and K. Kavukcuoglu. **Conditional image generation with PixelCNN decoders**. CoRR, abs/1606.05328, 2016b.