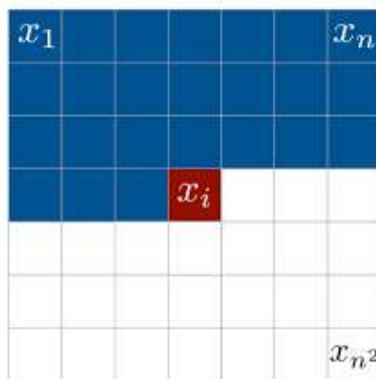


✓ HW5

Autoregressive Image Modeling

✓ Introduction

In this tutorial, we implement an autoregressive likelihood model for the task of image modeling. Autoregressive models are naturally strong generative models that constitute one of the current state-of-the-art architectures on likelihood-based image modeling, and are also the basis for large language generation models such as GPT3. Similar to the language generation autoregressive models work on images by modeling the likelihood of a pixel given all previous ones. Generally, autoregressive model over high-dimensional data \mathbf{x} factor the joint distribution as the following product of conditionals:



$$p(\mathbf{x}) = p(x_1, \dots, x_n) = \prod_{i=1}^n p(x_i | x_1, \dots, x_{i-1})$$

Learning these conditionals is often much simpler than learning the joint distribution $p(\mathbf{x})$ all together. However, disadvantages of autoregressive models include slow sampling, especially for large images, as we need height-times-width forward passes through the model. In addition, for some applications, we require a latent space as modeled in VAEs and Normalizing Flows. For instance, in autoregressive models, we cannot interpolate between two images because of the lack of a latent representation. We will explore and discuss these benefits and drawbacks alongside with our implementation.

✓ Libraries

```
%%capture
! pip install torchinfo
```

```

## Standard libraries
import numpy as np

## Imports for plotting
import matplotlib.pyplot as plt
plt.set_cmap('cividis')
%matplotlib inline
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('svg', 'pdf') # For export

## Progress bar
from tqdm.notebook import tqdm

## PyTorch
import torch
import torch.nn as nn
import torch.utils.data as data
import torch.optim as optim

# Torchvision
from torchvision.datasets import MNIST
from torchvision import transforms

# model information
import torchinfo

# Path to the folder where the datasets are/should be downloaded (e.g. MNIST)
DATASET_PATH = "../data"

# Ensure that all operations are deterministic on GPU (if used) for reproducibility
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False

# Fetching the device that will be used throughout this notebook
device = (
    torch.device("cpu") if not torch.cuda.is_available() else torch.device("cuda:0")
)
print("Using device", device)

<ipython-input-2-08e245a47d3b>:9: DeprecationWarning: `set_matplotlib_formats` is de
set_matplotlib_formats('svg', 'pdf') # For export
Using device cuda:0

```

▼ Data

```

def transform_func(img):
    threshold = int(0.33)
    return (img > threshold).float()

transform = transforms.Compose([transforms.ToTensor(), transform_func])

mnist_train = MNIST(root="../data", train=True, transform=transform, download=True)
mnist_test = MNIST(root="../data", train=False, transform=transform, download=True)

train_set, val_set = torch.utils.data.random_split(mnist_train, [50000, 10000])

train_loader = data.DataLoader(
    train_set, batch_size=128, shuffle=True, drop_last=True, pin_memory=True
)
val_loader = data.DataLoader(val_set, batch_size=128, shuffle=False, drop_last=False)
test_loader = data.DataLoader(
    mnist_test, batch_size=128, shuffle=False, drop_last=False
)

```

A good practice is to always visualize some data examples to get an intuition of the data:

```

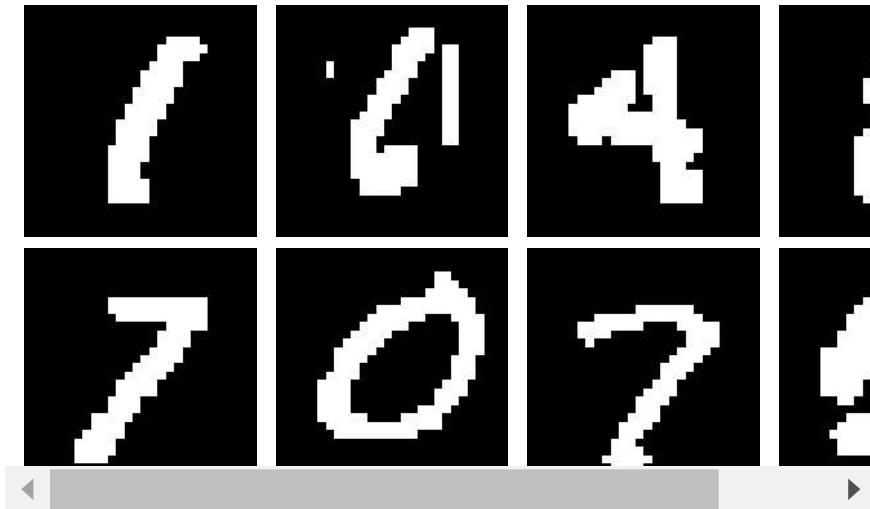
images = [train_set[i][0] for i in range(8)]

fig, axes = plt.subplots(2, 4, figsize=(8, 4)) # 2 rows, 4 columns

for i, tensor in enumerate(images):
    row = i // 4
    col = i % 4
    axes[row, col].imshow(tensor[0], cmap="gray")
    axes[row, col].axis("off")

plt.tight_layout()
plt.show()

```



✓ Masked autoregressive convolutions

The core module of PixelCNN is its masked convolutions. In contrast to language models, we don't apply an LSTM on each pixel one-by-one. This would be inefficient because images are grids instead of sequences. Thus, it is better to rely on convolutions that have shown great success in deep CNN classification models.

Nevertheless, we cannot just apply standard convolutions without any changes. Remember that during training of autoregressive models, we want to use teacher forcing which both helps the model training, and significantly reduces the time needed for training. For image modeling, teacher forcing is implemented by using a training image as input to the model, and we want to obtain as output the prediction for each pixel based on *only* its predecessors. Thus, we need to ensure that the prediction for a specific pixel can *only* be influenced by its predecessors and not by its own value or any "future" pixels. For this, we apply convolutions with a mask.

Which mask we use depends on the ordering of pixels we decide on, i.e. which is the first pixel we predict, which is the second one, etc. The most commonly used ordering is to denote the upper left pixel as the start pixel, and sort the pixels row by row, as shown in the visualization at the top of the tutorial. Thus, the second pixel is on the right of the first one (first row, second column), and once we reach the end of the row, we start in the second row, first column. If we now want to apply this to our convolutions, we need to ensure that the prediction of pixel 1 is not influenced by its own "true" input, and all pixels on its right and in any lower row. In convolutions, this means that we want to set those entries of the weight matrix to zero that take pixels on the right and below into account. As an example for a 5x5 kernel, see a mask below (figure credit - [Aaron van den Oord](#)):

1	1	1	1	1
1	1	1	1	1
1	1	0	0	0
0	0	0	0	0
0	0	0	0	0

(a)

1	1	1	1	1
1	1	1	1	1
1	1	1	0	0
0	0	0	0	0
0	0	0	0	0

(b)

Before looking into the application of masked convolutions in PixelCNN in detail, let's first implement a module that allows us to apply an arbitrary mask to a convolution:

```

class MaskedConvolution(nn.Module):
    def __init__(self, c_in, c_out, mask, **kwargs):
        """
        Implements a convolution with mask applied on its weights.
        Inputs:
            c_in - Number of input channels
            c_out - Number of output channels
            mask - Tensor of shape [kernel_size_H, kernel_size_W] with 0s where
                  the convolution should be masked, and 1s otherwise.
            kwargs - Additional arguments for the convolution
        """
        super().__init__()

        # For simplicity: calculate padding automatically
        kernel_size = (mask.shape[0], mask.shape[1])
        dilation = 1 if "dilation" not in kwargs else kwargs["dilation"]
        padding = tuple([dilation * (kernel_size[i] - 1) // 2 for i in range(2)])

        # Actual convolution
        self.conv = nn.Conv2d(c_in, c_out, kernel_size, padding=padding, **kwargs)

        # Mask as buffer => it is no parameter but still a tensor of the module
        # (must be moved with the devices)
        self.register_buffer("mask", mask[None, None])

    def forward(self, x):
        # Apply mask
        with torch.no_grad():
            self.conv.weight.data *= self.mask

        # Perform convolution
        return self.conv(x)

```

```

class PixelCNNMaskConvolution(MaskedConvolution):
    def __init__(self, c_in, c_out, kernel_size=3, mask_center=False, **kwargs):
        # Mask out all pixels below. For efficiency, we could also reduce the kernel
        # size in height, but for simplicity, we stick with masking here.
        mask = torch.ones(kernel_size, kernel_size)
        center = kernel_size // 2

        # Zero out all elements to the right of the center in the same row
        mask[center, center+1:] = 0
        # Zero out all elements in the rows below the center row
        mask[center+1:] = 0

        # For the very first convolution, we will also mask the center row
        if mask_center:
            mask[center, center] = 0

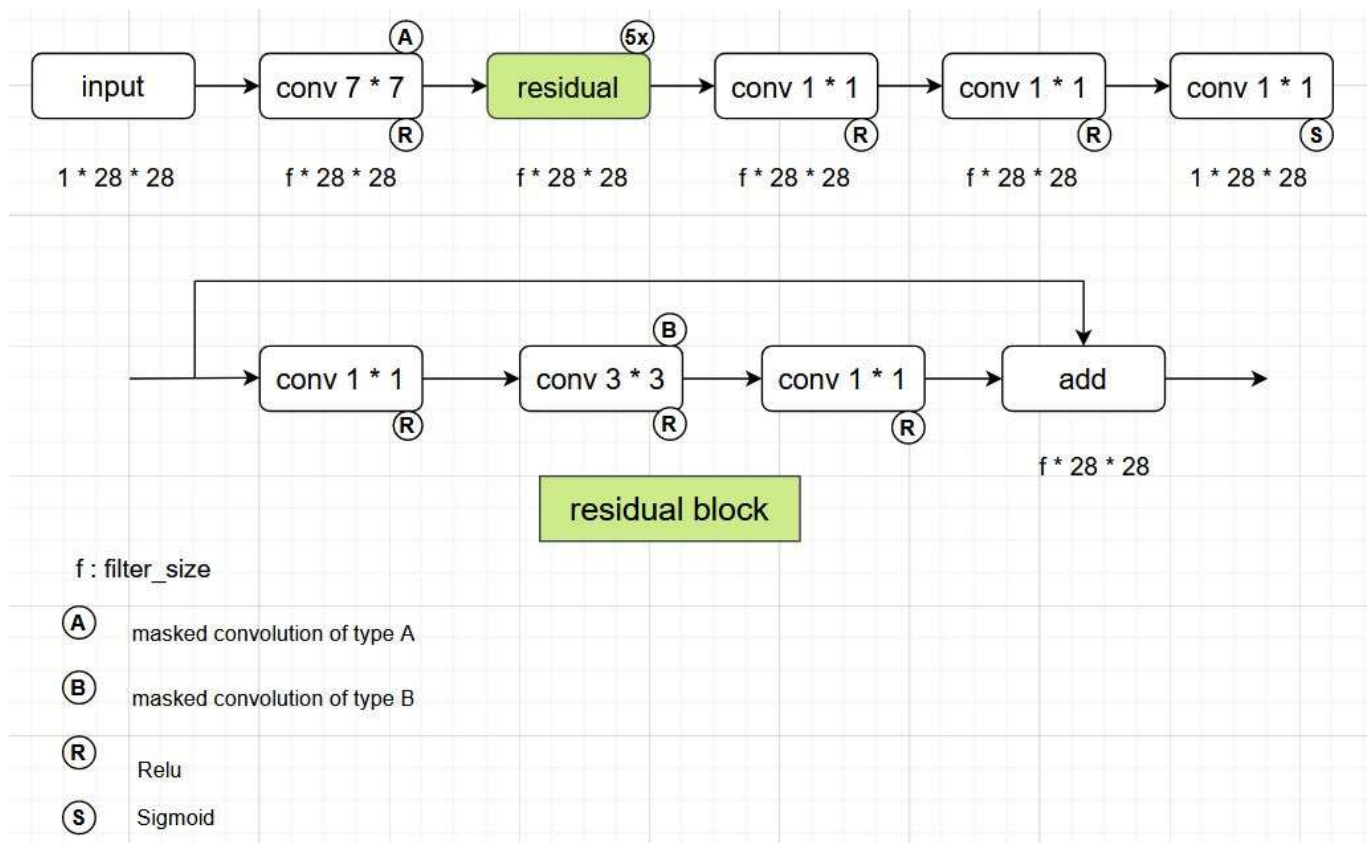
        super().__init__(c_in, c_out, mask, **kwargs)

```

Note that we have an input argument called `mask_center`. Remember that the input to the model is the actual input image. Hence, the very first convolution we apply cannot use the center pixel as input, but must be masked. All consecutive convolutions, however, should use the center pixel as we otherwise lose the features of the previous layer. Hence, the input argument `mask_center` is `True` for the very first convolutions, and `False` for all others.

✓ Implementing PixelCNN

you should implement PixelCNN according to below diagram



```
class ResidualBlock(nn.Module):
    def __init__(self, c_in, c_hidden):
        super().__init__()

        # First 1x1 convolution reduces channel dimension
        self.conv_1 = nn.Conv2d(c_in, c_hidden // 2, kernel_size=1)

        # Middle convolutions: masked 3x3 convolutions of type B
        self.conv_pixelcnn = PixelCNNMaskConvolution(c_hidden // 2, c_hidden // 2, kernel_size=3, mask_center=False)

        # Last 1x1 convolution expands channel dimension
        self.conv_2 = nn.Conv2d(c_hidden // 2, c_hidden, kernel_size=1)

        self.relu = nn.ReLU()

    def forward(self, x):
        identity = x
        out = self.conv_1(x)
        out = self.relu(out)
        out = self.conv_pixelcnn(out)
        out = self.relu(out)
        out = self.conv_2(out)
        out = out + identity # Skip connection
        out = self.relu(out)
        return out
```

```
class PixelCNN(nn.Module):
    def __init__(self, c_in, c_hidden, num_res_block=5):
        super().__init__()

        # Initial masked convolution (type A)
        self.conv_pixelcnn = PixelCNNMaskConvolution(c_in, c_hidden, kernel_size=7, mask_center=True)

        # Sequence of residual blocks
        self.residual_blocks = nn.ModuleList([ResidualBlock(c_hidden, c_hidden) for _ in range(num_res_block)])

        # Additional 1x1 convolutions
        self.conv_1 = nn.Conv2d(c_hidden, c_hidden, kernel_size=1)
        self.conv_2 = nn.Conv2d(c_hidden, c_hidden, kernel_size=1)

        # Output convolution
        self.conv_out = nn.Conv2d(c_hidden, 1, kernel_size=1)

        # Activation functions
        self.relu = nn.ReLU()
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        out = self.conv_pixelcnn(x)
        out = self.relu(out)

        # Pass through the sequence of residual blocks
        for block in self.residual_blocks:
            out = block(out)

        # Additional convolutions and activations
        out = self.conv_1(out)
        out = self.relu(out)
        out = self.conv_2(out)
        out = self.relu(out)

        # Final output convolution
        out = self.conv_out(out)
        out = self.sigmoid(out)

        return out

    @torch.no_grad()
    def sample(self, img_shape, img=None):
        # Create empty image or use the provided one
        if img is None:
            img = torch.zeros(img_shape).to(device)
        else:
            img = img.clone().to(device)

        batch, channels, rows, cols = img.shape

        # Sampling pixel by pixel
        for row in tqdm(range(rows)):
            for col in range(cols):
                for channel in range(channels):
                    # Pass the image through the network
                    logits = self.forward(img)
                    probs = torch.sigmoid(logits)

                    # Sample each pixel from a Bernoulli distribution
                    img[:, channel, row, col] = torch.bernoulli(probs[:, channel, row, col])

        return img
```

Summary ResidualBlock

```
c_in_channels = 128
c_hidden_channels = 128
res_block = ResidualBlock(c_in_channels, c_hidden_channels)

batch_size = 128

input_shape = (batch_size, c_in_channels, 28, 28)
torchinfo.summary(model=res_block, input_size=input_shape)
```

Layer (type:depth-idx)	Output Shape	Param #
ResidualBlock	[128, 128, 28, 28]	--
├─Conv2d: 1-1	[128, 64, 28, 28]	8,256
├─ReLU: 1-2	[128, 64, 28, 28]	--

```

PixelCNNMaskConvolution: 1-3      [128, 64, 28, 28]    --
└─Conv2d: 2-1                    [128, 64, 28, 28]    36,928
ReLU: 1-4                        [128, 64, 28, 28]    --
└─Conv2d: 1-5                    [128, 128, 28, 28]   8,320
ReLU: 1-6                        [128, 128, 28, 28]   --
=====
Total params: 53,504
Trainable params: 53,504
Non-trainable params: 0
Total mult-adds (G): 5.37
=====
Input size (MB): 51.38
Forward/backward pass size (MB): 205.52
Params size (MB): 0.21
Estimated Total Size (MB): 257.12
=====

```

Summary of PixelCNN

```

c_in_channels = 1
c_hidden_channels = 128

pixelcnn = PixelCNN(c_in_channels, c_hidden_channels)

batch_size = 128

input_shape = (batch_size, c_in_channels, 28, 28)

torchinfo.summary(model=pixelcnn, input_size=input_shape)

```

```

=====
PixelCNN                        [128, 1, 28, 28]    --
└─PixelCNNMaskConvolution: 1-1  [128, 128, 28, 28]   --
└─└─Conv2d: 2-1                 [128, 128, 28, 28]   6,400
ReLU: 1-2                      [128, 128, 28, 28]   --
└─ModuleList: 1-3              --
└─└─ResidualBlock: 2-2          [128, 128, 28, 28]   --
└─└─└─Conv2d: 3-1               [128, 64, 28, 28]    8,256
└─└─└─ReLU: 3-2                 [128, 64, 28, 28]    --
└─└─└─PixelCNNMaskConvolution: 3-3 [128, 64, 28, 28]   36,928
└─└─└─ReLU: 3-4                 [128, 64, 28, 28]    --
└─└─└─Conv2d: 3-5               [128, 128, 28, 28]   8,320
└─└─└─ReLU: 3-6                 [128, 128, 28, 28]   --
└─└─ResidualBlock: 2-3          [128, 128, 28, 28]   --
└─└─└─Conv2d: 3-7               [128, 64, 28, 28]    8,256
└─└─└─ReLU: 3-8                 [128, 64, 28, 28]    --
└─└─└─PixelCNNMaskConvolution: 3-9 [128, 64, 28, 28]   36,928
└─└─└─ReLU: 3-10                [128, 64, 28, 28]    --
└─└─└─Conv2d: 3-11              [128, 128, 28, 28]   8,320
└─└─└─ReLU: 3-12                [128, 128, 28, 28]   --
└─└─ResidualBlock: 2-4          [128, 128, 28, 28]   --
└─└─└─Conv2d: 3-13              [128, 64, 28, 28]    8,256
└─└─└─ReLU: 3-14                [128, 64, 28, 28]    --
└─└─└─PixelCNNMaskConvolution: 3-15 [128, 64, 28, 28]   36,928
└─└─└─ReLU: 3-16                [128, 64, 28, 28]    --
└─└─└─Conv2d: 3-17              [128, 128, 28, 28]   8,320
└─└─└─ReLU: 3-18                [128, 128, 28, 28]   --
└─└─ResidualBlock: 2-5          [128, 128, 28, 28]   --
└─└─└─Conv2d: 3-19              [128, 64, 28, 28]    8,256
└─└─└─ReLU: 3-20                [128, 64, 28, 28]    --
└─└─└─PixelCNNMaskConvolution: 3-21 [128, 64, 28, 28]   36,928
└─└─└─ReLU: 3-22                [128, 64, 28, 28]    --
└─└─└─Conv2d: 3-23              [128, 128, 28, 28]   8,320
└─└─└─ReLU: 3-24                [128, 128, 28, 28]   --
└─└─ResidualBlock: 2-6          [128, 128, 28, 28]   --
└─└─└─Conv2d: 3-25              [128, 64, 28, 28]    8,256
└─└─└─ReLU: 3-26                [128, 64, 28, 28]    --
└─└─└─PixelCNNMaskConvolution: 3-27 [128, 64, 28, 28]   36,928
└─└─└─ReLU: 3-28                [128, 64, 28, 28]    --
└─└─└─Conv2d: 3-29              [128, 128, 28, 28]   8,320
└─└─└─ReLU: 3-30                [128, 128, 28, 28]   --
└─Conv2d: 1-4                  [128, 128, 28, 28]   16,512
└─ReLU: 1-5                    [128, 128, 28, 28]   --
└─Conv2d: 1-6                  [128, 128, 28, 28]   16,512
└─ReLU: 1-7                    [128, 128, 28, 28]   --
└─Conv2d: 1-8                  [128, 1, 28, 28]     129
└─Sigmoid: 1-9                [128, 1, 28, 28]    --
=====
Total params: 307,073
Trainable params: 307,073
Non-trainable params: 0
Total mult-adds (G): 30.82
=====
Input size (MB): 0.40
Forward/backward pass size (MB): 1336.69
=====

```

▼ Training PixelCNN

```
c_in_channels = 1
c_hidden_channels = 128

model = PixelCNN(c_in_channels, c_hidden_channels)
model.to(device)
criterion = nn.BCELoss() # Use reconstruction loss (e.g., BCE)
optimizer = optim.Adam(model.parameters(), lr=0.0005)

# Training loop
num_epochs = 25
for epoch in tqdm(range(num_epochs), desc="epoch"):
    model.train()
    running_loss = 0.0

    for data, _ in tqdm(train_loader, desc="batch", leave=False):
        data = data.to(device)
        optimizer.zero_grad()
        outputs = model(data)
        loss = criterion(outputs, data)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

    print(
        f"Epoch [{epoch + 1}/{num_epochs}], Train Loss: {running_loss / len(train_loader):.4f}"
    )
    running_loss = 0.0

# Validation loop (optional)
model.eval()
val_loss = 0.0
with torch.no_grad():
    for data, _ in val_loader:
        data = data.to(device)
        outputs = model(data)
        loss = criterion(outputs, data)
        val_loss += loss.item()

val_loss /= len(test_loader)
print(f"Epoch [{epoch + 1}/{num_epochs}], Val Loss: {val_loss:.4f}")
```



```

epoch: 100%                                25/25 [17:49<00:00, 42.55s/it]

Epoch [1/25], Train Loss: 0.1341
Epoch [1/25], Val Loss: 0.0947
Epoch [2/25], Train Loss: 0.0928
Epoch [2/25], Val Loss: 0.0912
Epoch [3/25], Train Loss: 0.0909
Epoch [3/25], Val Loss: 0.0900
Epoch [4/25], Train Loss: 0.0895
Epoch [4/25], Val Loss: 0.0889
Epoch [5/25], Train Loss: 0.0884
Epoch [5/25], Val Loss: 0.0879
Epoch [6/25], Train Loss: 0.0877
Epoch [6/25], Val Loss: 0.0870
Epoch [7/25], Train Loss: 0.0872
Epoch [7/25], Val Loss: 0.0868
Epoch [8/25], Train Loss: 0.0867
Epoch [8/25], Val Loss: 0.0866
Epoch [9/25], Train Loss: 0.0863
Epoch [9/25], Val Loss: 0.0863
Epoch [10/25], Train Loss: 0.0860
Epoch [10/25], Val Loss: 0.0862
Epoch [11/25], Train Loss: 0.0858
Epoch [11/25], Val Loss: 0.0859
Epoch [12/25], Train Loss: 0.0855
Epoch [12/25], Val Loss: 0.0853
Epoch [13/25], Train Loss: 0.0852
Epoch [13/25], Val Loss: 0.0853
Epoch [14/25], Train Loss: 0.0851
Epoch [14/25], Val Loss: 0.0853
Epoch [15/25], Train Loss: 0.0848
Epoch [15/25], Val Loss: 0.0846
Epoch [16/25], Train Loss: 0.0847
Epoch [16/25], Val Loss: 0.0845
Epoch [17/25], Train Loss: 0.0845
Epoch [17/25], Val Loss: 0.0844
Epoch [18/25], Train Loss: 0.0844
Epoch [18/25], Val Loss: 0.0843
Epoch [19/25], Train Loss: 0.0842
Epoch [19/25], Val Loss: 0.0840
Epoch [20/25], Train Loss: 0.0841
Epoch [20/25], Val Loss: 0.0841
Epoch [21/25], Train Loss: 0.0839
Epoch [21/25], Val Loss: 0.0839
Epoch [22/25], Train Loss: 0.0838
Epoch [22/25], Val Loss: 0.0838
Epoch [23/25], Train Loss: 0.0837
Epoch [23/25], Val Loss: 0.0838
Epoch [24/25], Train Loss: 0.0836
Epoch [24/25], Val Loss: 0.0838
Epoch [25/25], Train Loss: 0.0835
Epoch [25/25], Val Loss: 0.0834

```

✓ Sampling

One way of qualitatively analysing generative models is by looking at the actual samples. Let's therefore use our sampling function to generate a few digits:

```

samples = model.sample(img_shape=(8, 1, 28, 28)).cpu()

fig, axes = plt.subplots(2, 4, figsize=(8, 4)) # 2 rows, 4 columns

for i, tensor in enumerate(samples):
    row = i // 4
    col = i % 4
    axes[row, col].imshow(tensor[0], cmap="gray")
    axes[row, col].axis("off")

plt.tight_layout()
plt.show()

```

100%

28/28 [00:01<00:00, 17.75it/s]



✓ PixelCNN suffers from blind spot

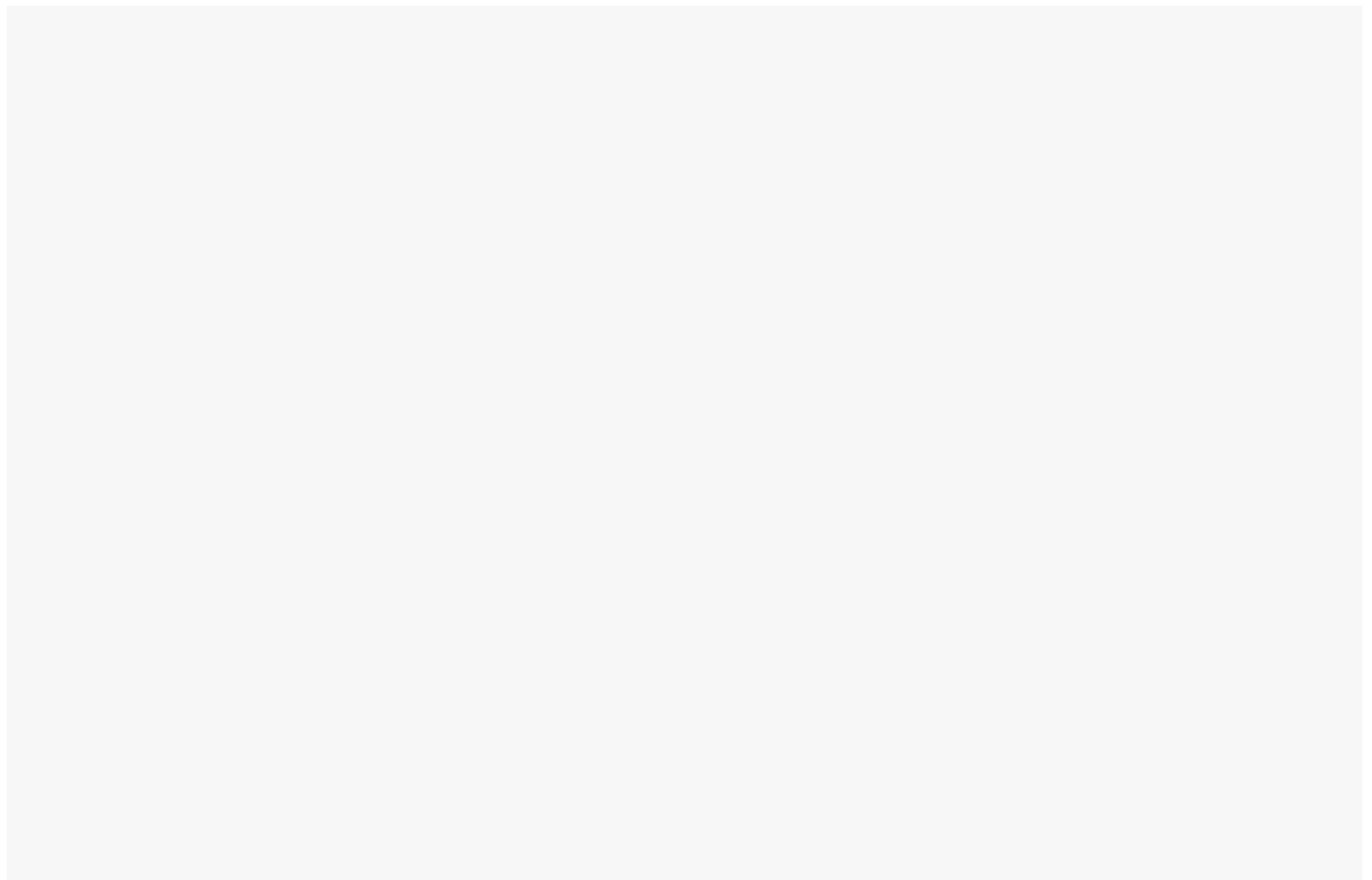
PixelCNN faces the challenge of a blind spot, an issue we're keen on investigating further. While there exist additional papers that have successfully tackled this problem, our current focus lies solely on exploring and understanding this particular challenge without delving into the analysis of those solutions.

what is receptive field of a convolution layer ?

The receptive field of a convolutional layer refers to the area of the input data that affects the activation of a particular neuron in that layer. In simpler terms, it represents the region in the input space that contributes to the calculation of a feature at a given layer in a convolutional neural network (CNN).

You are now required to analyze the following code snippets and answer to the provided question

✓ Visualizing the receptive field



```

inp_img = torch.zeros(1, 1, 11, 11)
inp_img.requires_grad_()

def show_center_recep_field(img, out):
    """
    Calculates the gradients of the input with respect to the output center pixel,
    and visualizes the overall receptive field.
    Inputs:
        img - Input image for which we want to calculate the receptive field on.
        out - Output features/loss which is used for backpropagation, and should be
              the output of the network/computation graph.
    """
    # Determine gradients
    loss = out[0, :, img.shape[2] // 2, img.shape[3] // 2].sum()
    loss.backward(
        retain_graph=True
    ) # Retain graph as we want to stack multiple layers and show the receptive field of all of them
    img_grads = img.grad.abs()
    img.grad.fill_(0) # Reset grads

    # Plot receptive field
    img = img_grads.squeeze().cpu().numpy()
    fig, ax = plt.subplots(1, 2)
    pos = ax[0].imshow(img)
    ax[1].imshow(img > 0)
    # Mark the center pixel in red if it doesn't have any gradients (should be the case for standard autoregressive models)

# Initialize convolutions with equal weight to all input pixels

inp_img = torch.zeros(1, 1, 11, 11)
inp_img.requires_grad_()

pixel_cnn_conv = PixelCNNMaskConvolution(
    c_in=1, c_out=1, kernel_size=3, mask_center=True
)
pixel_cnn_conv.conv.weight.data.fill_(1)
pixel_cnn_conv.conv.bias.data.fill_(0)
pi_cnn_img = pixel_cnn_conv(inp_img)

pixel_cnn_conv = PixelCNNMaskConvolution(
    c_in=1, c_out=1, kernel_size=3, mask_center=False
)
pixel_cnn_conv.conv.weight.data.fill_(1)
pixel_cnn_conv.conv.bias.data.fill_(0)

# We reuse our convolutions for the 4 layers here. Note that in a standard network,

```