Lecture 8 - Generative Adversarial Networks for Synthesising New Data (Part 2)

Contents

[1]:
```
from IPython.display import Image
%matplotlib inline
```

Improving the quality of synthesised images using a convolutional and Wasserstein GAN

DCGAN was proposed in 2016. Essentially we use convolutional layers for both the generator and discriminator networks.

Transposed convolution

Starting from a random vector, $\mathbf{z}$, the DCGAN first uses a fully connected layer to project $\mathbf{z}$ into a new vector with a proper size so that it can be reshaped into a spatial convolution representation, which is smaller than the output image size.

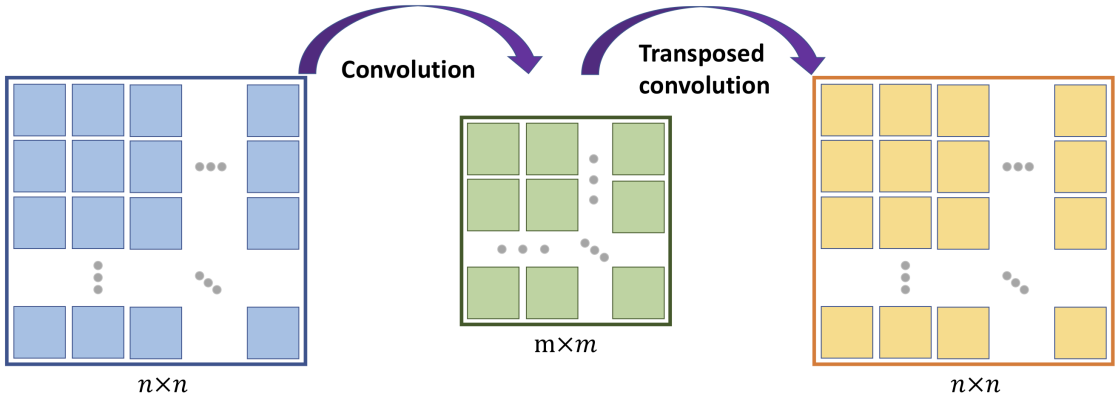
Then, a series of convolutional layers, known as *transposed convolution*, are used to *upsample* the feature maps to the desired output image size.

A *transposed convolution* operation is usually used for upsampling the feature space. It maintains the connectivity patterns betwen its input and output.

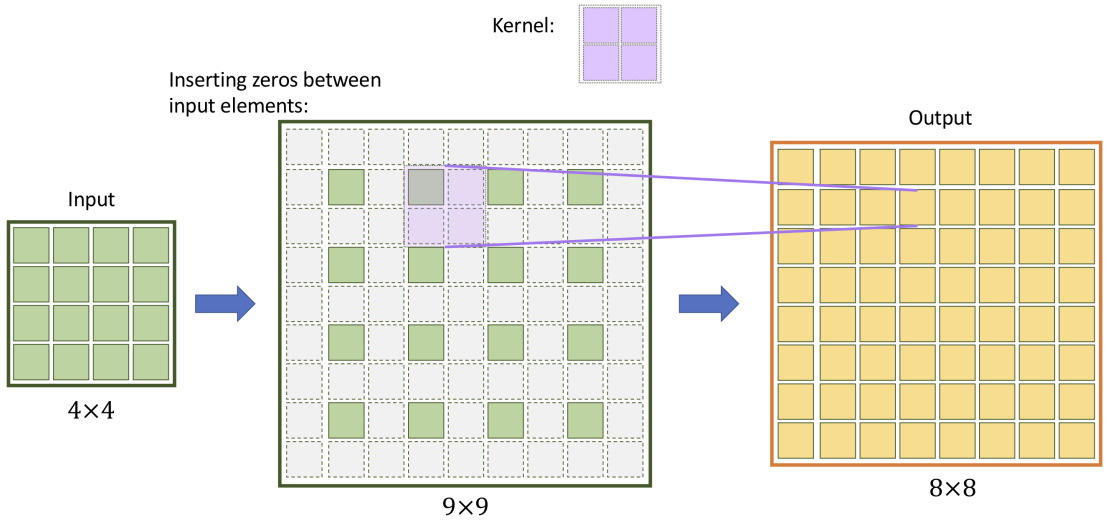
The transposed convolution is merely focused on recovering the dimensionality of the feature space and not the actual values.

[2]:
```
Image(filename='figures/17_09.png', width=700)
```

[2]:



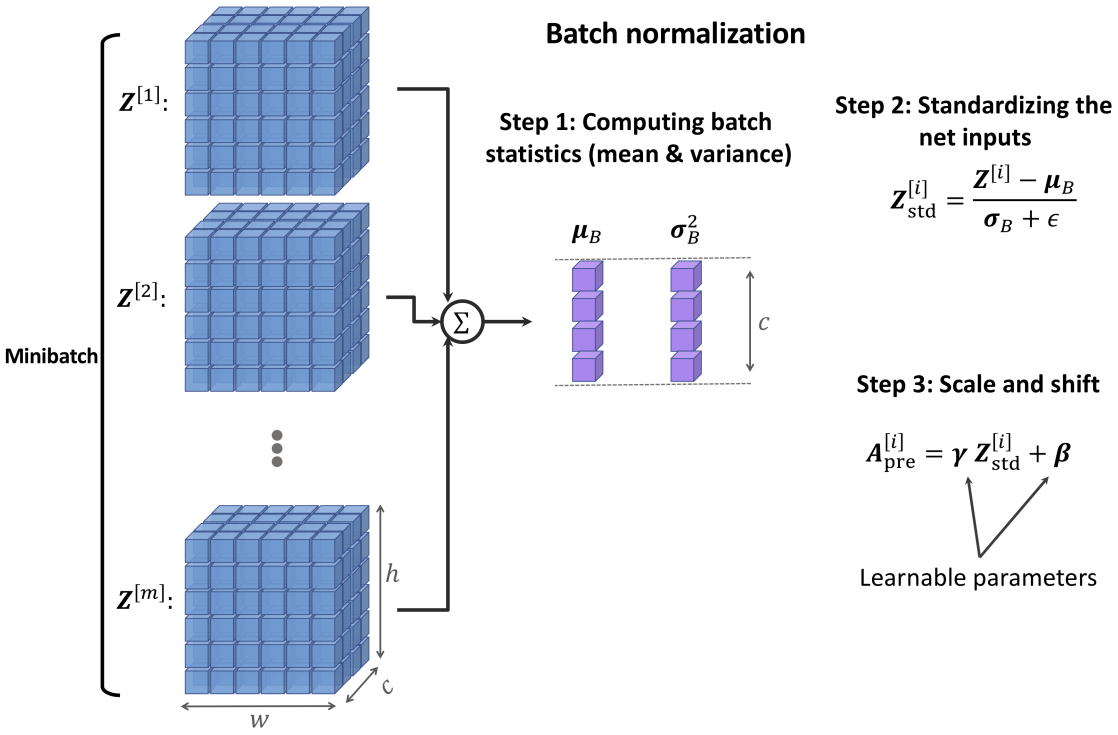
[3]:
```
Image(filename='figures/17_10.png', width=700)
```
[3]:



Batch normalization

One of the main ideas behind **BatchNorm** is normalising the layer inputs and preventing changes in their distribution during training, which enables faster and better convergence.

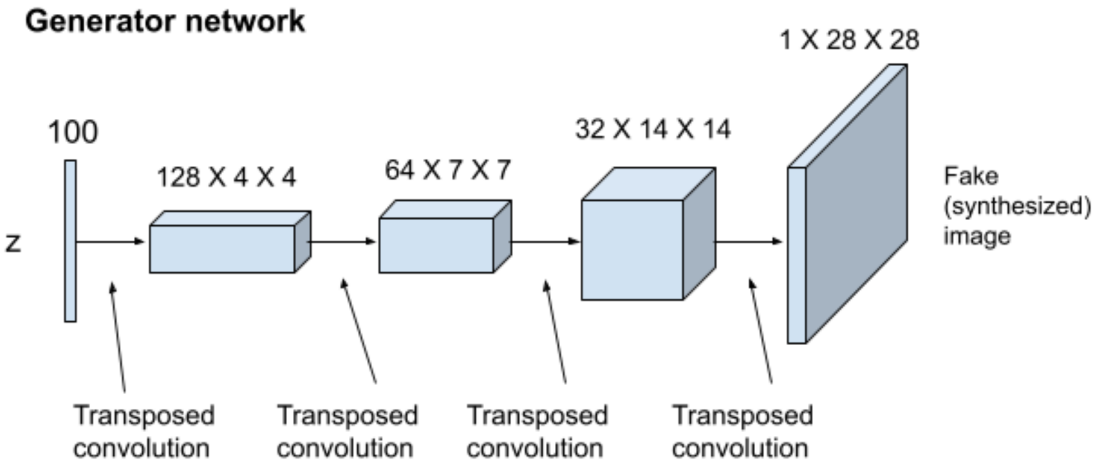
[4]:
```
Image(filename='figures/17_11.png', width=700)
```
[4]:



Implementing the generator and discriminator

[5]:
```
Image(filename='figures/17_12.png', width=700)
```

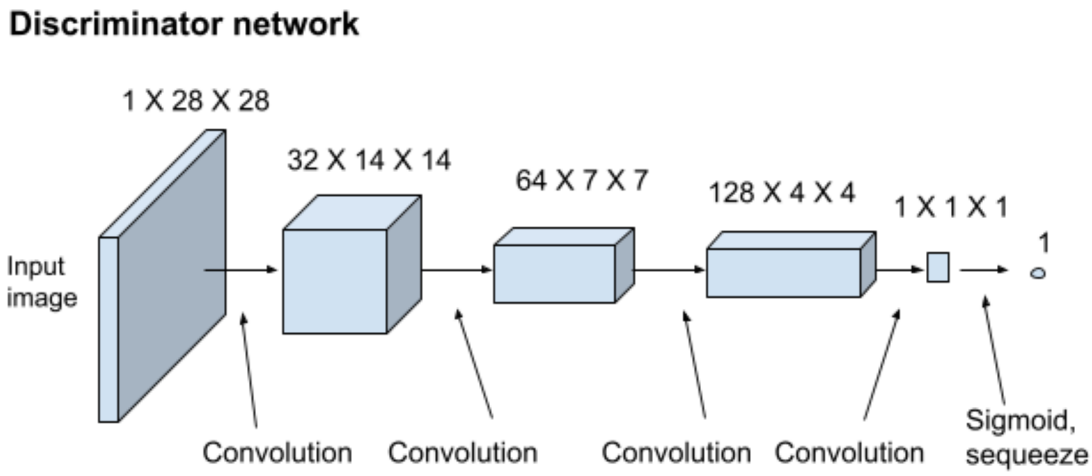
[5]:



Each transposed convolutional layer is followed by BatchNorm and leaky ReLU activation functions, except the last one, which uses tanh activation (without BatchNorm).

[6]:
```
Image(filename='figures/17_13.png', width=700)
```

[6]:



Each convolutional layer is also followed by BatchNorm and leaky ReLU activation.

The last convolutional layer uses kernels of size 7×7 and a single filter to reduce the spatial dimensionality of the output to $1 \times 1 \times 1$.

Finally, the convolutional output is followed by a sigmoid function and squeezed to one dimension.

Notice that the number of feature maps follows different trends between the generator and the discriminator.

In the generator, we start with a large number of feature maps and decrease them as we progress toward the last layer.

In the discriminator, we start with a small number of channels and increase it toward the last layer.

This is an important point for designing CNNs with the number of feature maps and the spatial size of the feature maps in reverse order.

When the spatial size of the feature maps increases, the number of feature maps decreases and vice versa.

[7]:
```
import torch


print(torch.__version__)
print("GPU Available:", torch.cuda.is_available())

if torch.cuda.is_available():
    device = torch.device("cuda:0")
else:
    device = "cpu"
```

```
2.2.0+cu118
GPU Available: True
```

[8]:
```
import torch.nn as nn
import numpy as np
import os

import matplotlib.pyplot as plt
%matplotlib inline
os.environ["KMP_DUPLICATE_LIB_OK"]="TRUE"
```

Train the DCGAN model

[9]:
```
import torchvision
from torchvision import transforms


image_path = './'
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=(0.5), std=(0.5))
])
mnist_dataset = torchvision.datasets.MNIST(root=image_path,
                                           train=True,
                                           transform=transform,
                                           download=False)

batch_size = 64

torch.manual_seed(1)
np.random.seed(1)

## Set up the dataset
from torch.utils.data import DataLoader
mnist_dl = DataLoader(mnist_dataset, batch_size=batch_size,
                      shuffle=True, drop_last=True)
```

[10]:
```
# torch.nn.ConvTranspose2d(in_channels, out_channels, kernel_size, stride=1,␣
 ↪padding=0,
# output_padding=0, groups=1, bias=True, dilation=1, padding_mode='zeros',␣
 ↪device=None, dtype=None)
```

[11]:
```
def make_generator_network(input_size, n_filters):
    model = nn.Sequential(
        nn.ConvTranspose2d(input_size, n_filters*4, 4, 1, 0,
                           bias=False),
        nn.BatchNorm2d(n_filters*4),
```

```
        nn.LeakyReLU(0.2),

        nn.ConvTranspose2d(n_filters*4, n_filters*2, 3, 2, 1, bias=False),
        nn.BatchNorm2d(n_filters*2),
        nn.LeakyReLU(0.2),

        nn.ConvTranspose2d(n_filters*2, n_filters, 4, 2, 1, bias=False),
        nn.BatchNorm2d(n_filters),
        nn.LeakyReLU(0.2),

        nn.ConvTranspose2d(n_filters, 1, 4, 2, 1, bias=False),
        nn.Tanh())
    return model

class Discriminator(nn.Module):
    def __init__(self, n_filters):
        super().__init__()
        self.network = nn.Sequential(
            nn.Conv2d(1, n_filters, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2),

            nn.Conv2d(n_filters, n_filters*2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(n_filters * 2),
            nn.LeakyReLU(0.2),

            nn.Conv2d(n_filters*2, n_filters*4, 3, 2, 1, bias=False),
            nn.BatchNorm2d(n_filters*4),
            nn.LeakyReLU(0.2),

            nn.Conv2d(n_filters*4, 1, 4, 1, 0, bias=False),
            nn.Sigmoid())

    def forward(self, input):
        output = self.network(input)
        return output.view(-1, 1).squeeze(0)
```

[12]:
```
z_size = 100
image_size = (28, 28)
n_filters = 32
gen_model = make_generator_network(z_size, n_filters).to(device)
print(gen_model)
disc_model = Discriminator(n_filters).to(device)
print(disc_model)
```

```
Sequential(
  (0): ConvTranspose2d(100, 128, kernel_size=(4, 4), stride=(1, 1), bias=False)
  (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
```

```
  (2): LeakyReLU(negative_slope=0.2)
  (3): ConvTranspose2d(128, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), bias=False)
  (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (5): LeakyReLU(negative_slope=0.2)
  (6): ConvTranspose2d(64, 32, kernel_size=(4, 4), stride=(2, 2), padding=(1,
1), bias=False)
  (7): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (8): LeakyReLU(negative_slope=0.2)
  (9): ConvTranspose2d(32, 1, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1),
bias=False)
  (10): Tanh()
)
Discriminator(
  (network): Sequential(
    (0): Conv2d(1, 32, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1),
bias=False)
    (1): LeakyReLU(negative_slope=0.2)
    (2): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1),
bias=False)
    (3): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (4): LeakyReLU(negative_slope=0.2)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),
bias=False)
    (6): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (7): LeakyReLU(negative_slope=0.2)
    (8): Conv2d(128, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (9): Sigmoid()
  )
)
```

[13]:
```python
## Loss function and optimizers:
loss_fn = nn.BCELoss()
g_optimizer = torch.optim.Adam(gen_model.parameters(), 0.0003)
d_optimizer = torch.optim.Adam(disc_model.parameters(), 0.0002)
```

The `create_noise()` function for generating random input must change to output a tensor of four dimensions instead of a vector.

[14]:
```python
def create_noise(batch_size, z_size, mode_z):
    if mode_z == 'uniform':
        input_z = torch.rand(batch_size, z_size, 1, 1)*2 - 1
    elif mode_z == 'normal':
        input_z = torch.randn(batch_size, z_size, 1, 1)
```

```
    return input_z
```

[15]:
```
## Train the discriminator
def d_train(x):
    disc_model.zero_grad()

    # Train discriminator with a real batch
    batch_size = x.size(0)
    x = x.to(device)
    d_labels_real = torch.ones(batch_size, 1, device=device)

    d_proba_real = disc_model(x)
    d_loss_real = loss_fn(d_proba_real, d_labels_real)

    # Train discriminator on a fake batch
    input_z = create_noise(batch_size, z_size, mode_z).to(device)
    g_output = gen_model(input_z)

    d_proba_fake = disc_model(g_output)
    d_labels_fake = torch.zeros(batch_size, 1, device=device)
    d_loss_fake = loss_fn(d_proba_fake, d_labels_fake)

    # gradient backprop & optimize ONLY D's parameters
    d_loss = d_loss_real + d_loss_fake
    d_loss.backward()
    d_optimizer.step()

    return d_loss.data.item(), d_proba_real.detach(), d_proba_fake.detach()
```

[16]:
```
## Train the generator
def g_train(x):
    gen_model.zero_grad()

    batch_size = x.size(0)
    input_z = create_noise(batch_size, z_size, mode_z).to(device)
    g_labels_real = torch.ones((batch_size, 1), device=device)

    g_output = gen_model(input_z)
    d_proba_fake = disc_model(g_output)
    g_loss = loss_fn(d_proba_fake, g_labels_real)

    # gradient backprop & optimize ONLY G's parameters
    g_loss.backward()
    g_optimizer.step()

    return g_loss.data.item()
```

[17]:
```
mode_z = 'uniform'
fixed_z = create_noise(batch_size, z_size, mode_z).to(device)

def create_samples(g_model, input_z):
    g_output = g_model(input_z)
    images = torch.reshape(g_output, (batch_size, *image_size))
    return (images+1)/2.0

epoch_samples = []

num_epochs = 100
torch.manual_seed(1)

for epoch in range(1, num_epochs+1):
    gen_model.train()
    d_losses, g_losses = [], []
    for i, (x, _) in enumerate(mnist_dl):
        d_loss, d_proba_real, d_proba_fake = d_train(x)
        d_losses.append(d_loss)
        g_losses.append(g_train(x))

    print(f'Epoch {epoch:03d} | Avg Losses >>'
          f' G/D {torch.FloatTensor(g_losses).mean():.4f}'
          f'/{torch.FloatTensor(d_losses).mean():.4f}')
    gen_model.eval()
    epoch_samples.append(
        create_samples(gen_model, fixed_z).detach().cpu().numpy())
```

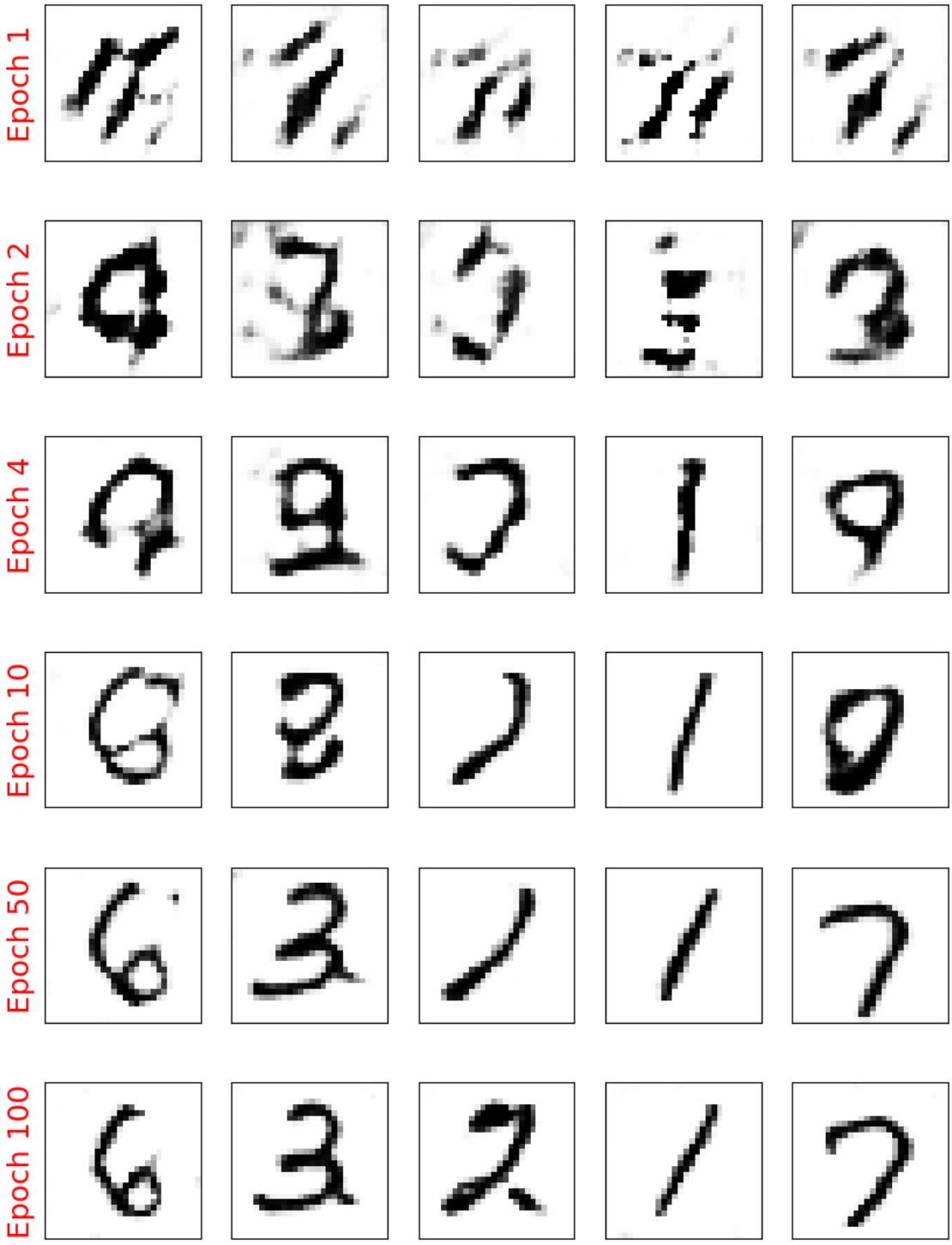
```
Epoch 001 | Avg Losses >> G/D 4.9723/0.0950
Epoch 002 | Avg Losses >> G/D 4.9875/0.1307
Epoch 003 | Avg Losses >> G/D 3.9528/0.2480
Epoch 004 | Avg Losses >> G/D 3.3329/0.2903
Epoch 005 | Avg Losses >> G/D 3.1356/0.2985
Epoch 006 | Avg Losses >> G/D 3.0575/0.3065
Epoch 007 | Avg Losses >> G/D 2.9838/0.3326
Epoch 008 | Avg Losses >> G/D 2.9943/0.2993
Epoch 009 | Avg Losses >> G/D 3.0121/0.2905
Epoch 010 | Avg Losses >> G/D 3.1279/0.2789
Epoch 011 | Avg Losses >> G/D 3.1872/0.2932
Epoch 012 | Avg Losses >> G/D 3.0964/0.2852
Epoch 013 | Avg Losses >> G/D 3.2038/0.2787
Epoch 014 | Avg Losses >> G/D 3.2222/0.2639
Epoch 015 | Avg Losses >> G/D 3.1557/0.2669
Epoch 016 | Avg Losses >> G/D 3.2351/0.2459
Epoch 017 | Avg Losses >> G/D 3.3024/0.2632
Epoch 018 | Avg Losses >> G/D 3.3195/0.2479
Epoch 019 | Avg Losses >> G/D 3.3728/0.2555
Epoch 020 | Avg Losses >> G/D 3.3828/0.2556
```

```
Epoch 021 | Avg Losses >> G/D 3.3904/0.2521
Epoch 022 | Avg Losses >> G/D 3.4402/0.2587
Epoch 023 | Avg Losses >> G/D 3.3915/0.2408
Epoch 024 | Avg Losses >> G/D 3.4119/0.2511
Epoch 025 | Avg Losses >> G/D 3.5215/0.2215
Epoch 026 | Avg Losses >> G/D 3.4802/0.2417
Epoch 027 | Avg Losses >> G/D 3.6205/0.2220
Epoch 028 | Avg Losses >> G/D 3.5457/0.2495
Epoch 029 | Avg Losses >> G/D 3.4958/0.2406
Epoch 030 | Avg Losses >> G/D 3.6623/0.2248
Epoch 031 | Avg Losses >> G/D 3.6214/0.2272
Epoch 032 | Avg Losses >> G/D 3.6703/0.2096
Epoch 033 | Avg Losses >> G/D 3.6258/0.2348
Epoch 034 | Avg Losses >> G/D 3.7709/0.2007
Epoch 035 | Avg Losses >> G/D 3.7544/0.1976
Epoch 036 | Avg Losses >> G/D 3.7711/0.1998
Epoch 037 | Avg Losses >> G/D 3.8014/0.2078
Epoch 038 | Avg Losses >> G/D 3.8633/0.1899
Epoch 039 | Avg Losses >> G/D 3.9147/0.1946
Epoch 040 | Avg Losses >> G/D 3.9252/0.1954
Epoch 041 | Avg Losses >> G/D 3.9986/0.1729
Epoch 042 | Avg Losses >> G/D 3.9962/0.1800
Epoch 043 | Avg Losses >> G/D 4.0905/0.1566
Epoch 044 | Avg Losses >> G/D 4.0505/0.1816
Epoch 045 | Avg Losses >> G/D 4.1602/0.1533
Epoch 046 | Avg Losses >> G/D 4.1360/0.1630
Epoch 047 | Avg Losses >> G/D 4.1782/0.1647
Epoch 048 | Avg Losses >> G/D 4.2308/0.1517
Epoch 049 | Avg Losses >> G/D 4.3364/0.1778
Epoch 050 | Avg Losses >> G/D 4.3593/0.1608
Epoch 051 | Avg Losses >> G/D 4.2618/0.1665
Epoch 052 | Avg Losses >> G/D 4.3498/0.1546
Epoch 053 | Avg Losses >> G/D 4.3082/0.1568
Epoch 054 | Avg Losses >> G/D 4.4111/0.1435
Epoch 055 | Avg Losses >> G/D 4.3707/0.1530
Epoch 056 | Avg Losses >> G/D 4.5076/0.1200
Epoch 057 | Avg Losses >> G/D 4.4705/0.1612
Epoch 058 | Avg Losses >> G/D 4.3794/0.1468
Epoch 059 | Avg Losses >> G/D 4.5151/0.1292
Epoch 060 | Avg Losses >> G/D 4.4448/0.1677
Epoch 061 | Avg Losses >> G/D 4.6481/0.1129
Epoch 062 | Avg Losses >> G/D 4.4879/0.1676
Epoch 063 | Avg Losses >> G/D 4.6180/0.1190
Epoch 064 | Avg Losses >> G/D 4.6757/0.1301
Epoch 065 | Avg Losses >> G/D 4.6559/0.1282
Epoch 066 | Avg Losses >> G/D 4.6814/0.1428
Epoch 067 | Avg Losses >> G/D 4.7575/0.0999
Epoch 068 | Avg Losses >> G/D 4.6140/0.1526
```

```
Epoch 069 | Avg Losses >> G/D 4.7775/0.1074
Epoch 070 | Avg Losses >> G/D 4.9386/0.1071
Epoch 071 | Avg Losses >> G/D 4.6921/0.1331
Epoch 072 | Avg Losses >> G/D 4.7206/0.1356
Epoch 073 | Avg Losses >> G/D 4.7148/0.1114
Epoch 074 | Avg Losses >> G/D 4.8521/0.1118
Epoch 075 | Avg Losses >> G/D 4.7393/0.1363
Epoch 076 | Avg Losses >> G/D 4.8592/0.1055
Epoch 077 | Avg Losses >> G/D 4.7506/0.1360
Epoch 078 | Avg Losses >> G/D 4.8573/0.1177
Epoch 079 | Avg Losses >> G/D 5.0776/0.0872
Epoch 080 | Avg Losses >> G/D 4.8844/0.1308
Epoch 081 | Avg Losses >> G/D 4.9048/0.1162
Epoch 082 | Avg Losses >> G/D 4.9917/0.1083
Epoch 083 | Avg Losses >> G/D 4.8710/0.1264
Epoch 084 | Avg Losses >> G/D 4.9354/0.1099
Epoch 085 | Avg Losses >> G/D 5.0177/0.1049
Epoch 086 | Avg Losses >> G/D 4.9688/0.1182
Epoch 087 | Avg Losses >> G/D 5.0076/0.1226
Epoch 088 | Avg Losses >> G/D 5.0254/0.0964
Epoch 089 | Avg Losses >> G/D 4.9067/0.1575
Epoch 090 | Avg Losses >> G/D 5.0764/0.0883
Epoch 091 | Avg Losses >> G/D 5.0349/0.1381
Epoch 092 | Avg Losses >> G/D 4.9630/0.1088
Epoch 093 | Avg Losses >> G/D 5.1014/0.1031
Epoch 094 | Avg Losses >> G/D 5.0356/0.1249
Epoch 095 | Avg Losses >> G/D 5.1472/0.0855
Epoch 096 | Avg Losses >> G/D 5.1476/0.1106
Epoch 097 | Avg Losses >> G/D 5.1307/0.1096
Epoch 098 | Avg Losses >> G/D 5.1388/0.1001
Epoch 099 | Avg Losses >> G/D 5.0935/0.0958
Epoch 100 | Avg Losses >> G/D 5.1870/0.1087
```

[18]:
```python
selected_epochs = [1, 2, 4, 10, 50, 100]
fig = plt.figure(figsize=(10, 14))
for i,e in enumerate(selected_epochs):
    for j in range(5):
        ax = fig.add_subplot(6, 5, i*5+j+1)
        ax.set_xticks([])
        ax.set_yticks([])
        if j == 0:
            ax.text(
                -0.06, 0.5, f'Epoch {e}',
                rotation=90, size=18, color='red',
                horizontalalignment='right',
                verticalalignment='center',
                transform=ax.transAxes)
```

```
        image = epoch_samples[e-1][j]
        ax.imshow(image, cmap='gray_r')

# plt.savefig('figures/ch17-dcgan-samples.pdf')
plt.show()
```



Comparing to vanilla GAN, the new examples from DCGAN look to be of a much better quality.

The simplest approach to compare quality of the output is visual assessment.

A much more objective way of training the generator would be to minimise the dissimilarity between the distribution observed in the real data and the distribution observed in synthesised examples.

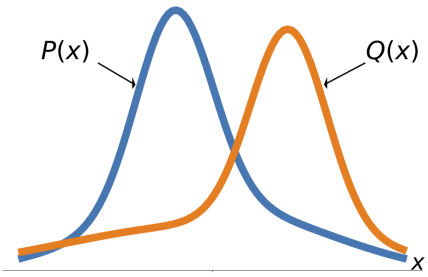
This leads to **Wasserstein GAN** where we used a modified loss function based on the so-called Wasserstein-1 (or earth mover's) distance between the distributions of real and fake images for improving the training performance.

Dissimilarity measures between two distributions

[19]:
```
Image(filename='figures/17_14.png', width=700)
```

[19]:

Measures	Formulation
Total variation (TV)	$TV(P,Q) = \sup_x \lvert P(x) - Q(x) \rvert$
Kullback-Leibler (KL) divergence	$KL(P\lvert\lvert Q) = \int P(x) \log \frac{P(x)}{Q(x)} dx$
Jensen-Shannon (JS) divergence	$JS(P,Q) = \frac{1}{2}\left(KL\left(P\lvert\lvert \frac{P+Q}{2}\right) + KL\left(Q\lvert\lvert \frac{P+Q}{2}\right)\right)$
Earth mover's (EM) distance	$EM(P,Q) = \inf_{\gamma \in \Pi(P,Q)} E_{(u,v)\in\gamma}(\lVert u - v \rVert)$



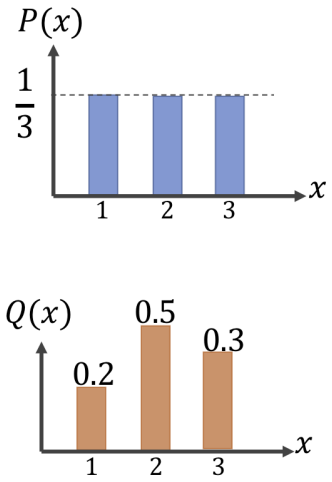
The EM distance can be interpreted as the minimal amount of work needed to transform one distribution into the other.

Computing EM distance is an optimisation problem by itself, which is to find the optimal transfer plan, $\gamma(u, v)$.

It can be mathematically shown that the loss function in the original GAN indeed minimizes the JS divergence between the distribution of real and fake examples. However, JS divergence has issues in training a GAN model, and replacing it with EM distance shows a much better performance.

[20]:
```
Image(filename='figures/17_15.png', width=800)
```

[20]:



Total variation:

$$TV(P,Q) = \sup_x \left\{ \left|\frac{1}{3} - 0.2\right|, \left|\frac{1}{3} - 0.5\right|, \left|\frac{1}{3} - 0.3\right| \right\} = 0.167$$

KL divergence:

$$KL(P||Q) = 0.33 \log\left(\frac{0.33}{0.2}\right) + 0.33 \log\left(\frac{0.33}{0.5}\right) + 0.33 \log\left(\frac{0.33}{0.3}\right) = 0.101$$

$$KL(Q||P) = 0.2 \log\left(\frac{0.2}{0.33}\right) + 0.5 \log\left(\frac{0.5}{0.33}\right) + 0.33 \log\left(\frac{0.3}{0.33}\right) = 0.099$$

JS divergence:

$$P_m \rightarrow \left[\frac{0.33+0.2}{2}, \frac{0.33+0.5}{2}, \frac{0.33+0.3}{2}\right] = [0.26, 0.42, 0.32]$$

$$\left.\begin{array}{l} KL(P||P_m) = 0.0246 \\ KL(Q||P_m) = 0.0246 \end{array}\right\} \rightarrow JS(P||Q) = 0.0248$$

EM distance:

$$EM(P,Q) = (0.33 - 0.2) + (0.33 - 0.3) = 0.16$$

What is the advantage of using EM distance?,

Assume we have two distributions, P and Q, which are two parallel lines. One line is fixed at $x = \theta$ and the other line can move across the x-axis but is initially located at $x = 0$, where $\theta > 0$.

KL, TV, and JS dissimilarity measures are $KL(P||Q) = +\infty, TV(P,Q) = 1, JS(P,Q) = \frac{1}{2}\log 2$. On the other hand, $EM(P,Q) = |\theta|$ whose gradient with respect to θ exists and can push Q toward P.

The computation of the EM distance can be simplified using a theorem called **Kantorovich-Rubinstein duality**, as follows

$$W(P_r, P_g) = \sup_{\|f\|_L \leq 1} \left(\mathbb{E}_{u \in P_r}[f(u)] - \mathbb{E}_{v \in P_g}[f(v)]\right).$$

P_r is the distribution of the real examples and P_g the distributions of fake (generated) examples.

Using EM distance in practice for GANs

Recall that deep NNs are universal function approximators, then we can simply train an NN model to approximate the Wasserstein distance function.

The *vanilla GAN* uses a discriminator in the form of a classifier - generates a probability value.

For *WGAN*, the discriminator can be changed to behave as a *critic*, which returns a scalar score. We can interpret this score as how realistic the input images are (like an art critic giving scores to artworks in a gallery).

The critic (that is, the discriminator network) returns its outputs for the batch of real image examples and the batch of synthesised examples. The loss functions are defined as

- The real image component of the discriminator's loss

$$L^D_{real} = -\frac{1}{N}\sum_i D(\mathbf{x}_i).$$

- The fake image component of the discriminator's loss

$$L^D_{fake} = \frac{1}{N}\sum_i D(G(\mathbf{z}_i)).$$

- The loss for the generator

$$L^G = -\frac{1}{N}\sum_i D(G(\mathbf{z}_i)).$$

To ensure that the 1-Lipschitz property of the critic function is preserved during training, weights are clamped to a small region, for example, $[\check{}0.01, 0.01]$.

Gradient penalty

Weight clipping can also lead to **capacity underuse**, which means that the critic network is limited to learning only some simple functions, as opposed to more complex functions.

Therefore, rather than clipping the weights, **gradient penalty (GP)** is proposed. The result is the **WGAN with gradient penalty (WGAN-GP)**.

The procedure of gradient clipping is summarised as below:

1. For each pair of real and fake examples $(\mathbf{x}^{[i]}, \widetilde{\mathbf{x}}^{[i]})$ in a given batch, choose a random number $\alpha^{[i]} \in U(0, 1)$.
2. Calculate an interpolation $\widehat{x}^{[i]} = \alpha^{[i]}\mathbf{x}^{[i]} + (1 - \alpha^{[i]})\widetilde{\mathbf{x}}^{[i]}$.
3. Compute $D(\widehat{x}^{[i]})$.
4. Calculate $\nabla_{\widehat{x}^{[i]}} D(\widehat{x}^{[i]})$.
5. Compute GP

$$L^D_{GP} = \frac{1}{N}\left(\|\nabla_{\widehat{x}^{[i]}} D(\widehat{x}^{[i]})\|_2 - 1\right)^2.$$

Total loss for discriminator $L^D_{total} = L^D_{real} + L^D_{fake} + \lambda L^D_{GP}$. Here, λ is a tunable hyperparameter.

Implementing WGAN-GP to train the DCGAN model

It is recommended to use layer normalisation in WGAN instead of batch normalisation. Layer normalisation normalises the inputs across features instead of across the batch dimension in batch normalisation. We used layer normalisation for Transformers.

[21]:
```
def make_generator_network_wgan(input_size, n_filters):
    model = nn.Sequential(
        nn.ConvTranspose2d(input_size, n_filters*4, 4, 1, 0,
                           bias=False),
        nn.InstanceNorm2d(n_filters*4),
```

```
        nn.LeakyReLU(0.2),

        nn.ConvTranspose2d(n_filters*4, n_filters*2, 3, 2, 1, bias=False),
        nn.InstanceNorm2d(n_filters*2),
        nn.LeakyReLU(0.2),

        nn.ConvTranspose2d(n_filters*2, n_filters, 4, 2, 1, bias=False),
        nn.InstanceNorm2d(n_filters),
        nn.LeakyReLU(0.2),

        nn.ConvTranspose2d(n_filters, 1, 4, 2, 1, bias=False),
        nn.Tanh())
    return model

class DiscriminatorWGAN(nn.Module):
    def __init__(self, n_filters):
        super().__init__()
        self.network = nn.Sequential(
            nn.Conv2d(1, n_filters, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2),

            nn.Conv2d(n_filters, n_filters*2, 4, 2, 1, bias=False),
            nn.InstanceNorm2d(n_filters * 2),
            nn.LeakyReLU(0.2),

            nn.Conv2d(n_filters*2, n_filters*4, 3, 2, 1, bias=False),
            nn.InstanceNorm2d(n_filters*4),
            nn.LeakyReLU(0.2),

            nn.Conv2d(n_filters*4, 1, 4, 1, 0, bias=False),
            nn.Sigmoid())

    def forward(self, input):
        output = self.network(input)
        return output.view(-1, 1).squeeze(0)
```

[22]:
```
gen_model = make_generator_network_wgan(z_size, n_filters).to(device)
disc_model = DiscriminatorWGAN(n_filters).to(device)

g_optimizer = torch.optim.Adam(gen_model.parameters(), 0.0002)
d_optimizer = torch.optim.Adam(disc_model.parameters(), 0.0002)
```

[23]:
```
from torch.autograd import grad as torch_grad


def gradient_penalty(real_data, generated_data):
    batch_size = real_data.size(0)
```

```
    # Calculate interpolation
    alpha = torch.rand(real_data.shape[0], 1, 1, 1, requires_grad=True,␣
↪device=device)
    interpolated = alpha * real_data + (1 - alpha) * generated_data

    # Calculate probability of interpolated examples
    proba_interpolated = disc_model(interpolated)

    # Calculate gradients of probabilities with respect to examples
    gradients = torch_grad(outputs=proba_interpolated, inputs=interpolated,
                           grad_outputs=torch.ones(proba_interpolated.size(),␣
↪device=device),
                           create_graph=True, retain_graph=True)[0]

    gradients = gradients.view(batch_size, -1)
    gradients_norm = gradients.norm(2, dim=1)
    return lambda_gp * ((gradients_norm - 1)**2).mean()
```

[24]:
```
## Train the discriminator
def d_train_wgan(x):
    disc_model.zero_grad()

    batch_size = x.size(0)
    x = x.to(device)

    # Calculate probabilities on real and generated data
    d_real = disc_model(x)
    input_z = create_noise(batch_size, z_size, mode_z).to(device)
    g_output = gen_model(input_z)
    d_generated = disc_model(g_output)
    d_loss = d_generated.mean() - d_real.mean() + gradient_penalty(x.data,␣
↪g_output.data)
    d_loss.backward()
    d_optimizer.step()

    return d_loss.data.item()
```

[25]:
```
## Train the generator
def g_train_wgan(x):
    gen_model.zero_grad()

    batch_size = x.size(0)
    input_z = create_noise(batch_size, z_size, mode_z).to(device)
    g_output = gen_model(input_z)

    d_generated = disc_model(g_output)
```

```
    g_loss = -d_generated.mean()

    # gradient backprop & optimize ONLY G's parameters
    g_loss.backward()
    g_optimizer.step()

    return g_loss.data.item()
```

[ ]:
```
epoch_samples_wgan = []
lambda_gp = 10.0
num_epochs = 100
torch.manual_seed(1)
critic_iterations = 5

for epoch in range(1, num_epochs+1):
    gen_model.train()
    d_losses, g_losses = [], []
    for i, (x, _) in enumerate(mnist_dl):
        for _ in range(critic_iterations):
            d_loss = d_train_wgan(x)
        d_losses.append(d_loss)
        g_losses.append(g_train_wgan(x))

    print(f'Epoch {epoch:03d} | D Loss >>'
          f' {torch.FloatTensor(d_losses).mean():.4f}')
    gen_model.eval()
    epoch_samples_wgan.append(
        create_samples(gen_model, fixed_z).detach().cpu().numpy())
```

```
Epoch 001 | D Loss >> -0.2706
Epoch 002 | D Loss >> -0.5488
Epoch 003 | D Loss >> -0.6275
Epoch 004 | D Loss >> -0.6664
Epoch 005 | D Loss >> -0.6733
Epoch 006 | D Loss >> -0.6592
Epoch 007 | D Loss >> -0.6066
Epoch 008 | D Loss >> -0.5697
Epoch 009 | D Loss >> -0.5445
Epoch 010 | D Loss >> -0.5273
Epoch 011 | D Loss >> -0.5174
Epoch 012 | D Loss >> -0.5063
Epoch 013 | D Loss >> -0.4927
Epoch 014 | D Loss >> -0.4827
Epoch 015 | D Loss >> -0.4739
Epoch 016 | D Loss >> -0.4659
Epoch 017 | D Loss >> -0.4645
Epoch 018 | D Loss >> -0.4580
Epoch 019 | D Loss >> -0.4550
```

```
Epoch 020 | D Loss >> -0.4606
Epoch 021 | D Loss >> -0.4575
Epoch 022 | D Loss >> -0.4516
Epoch 023 | D Loss >> -0.4585
Epoch 024 | D Loss >> -0.4531
Epoch 025 | D Loss >> -0.4539
Epoch 026 | D Loss >> -0.4592
Epoch 027 | D Loss >> -0.4559
Epoch 028 | D Loss >> -0.4630
Epoch 029 | D Loss >> -0.4625
Epoch 030 | D Loss >> -0.4640
Epoch 031 | D Loss >> -0.4670
Epoch 032 | D Loss >> -0.4606
Epoch 033 | D Loss >> -0.4606
Epoch 034 | D Loss >> -0.4635
Epoch 035 | D Loss >> -0.4650
Epoch 036 | D Loss >> -0.4634
Epoch 037 | D Loss >> -0.4683
Epoch 038 | D Loss >> -0.4672
Epoch 039 | D Loss >> -0.4677
Epoch 040 | D Loss >> -0.4615
Epoch 041 | D Loss >> -0.4697
Epoch 042 | D Loss >> -0.4679
Epoch 043 | D Loss >> -0.4698
Epoch 044 | D Loss >> -0.4623
Epoch 045 | D Loss >> -0.4663
Epoch 046 | D Loss >> -0.4703
Epoch 047 | D Loss >> -0.4707
Epoch 048 | D Loss >> -0.4719
Epoch 049 | D Loss >> -0.4690
Epoch 050 | D Loss >> -0.4719
Epoch 051 | D Loss >> -0.4728
Epoch 052 | D Loss >> -0.4719
Epoch 053 | D Loss >> -0.4666
Epoch 054 | D Loss >> -0.4708
Epoch 055 | D Loss >> -0.4720
Epoch 056 | D Loss >> -0.4689
Epoch 057 | D Loss >> -0.4753
Epoch 058 | D Loss >> -0.4754
Epoch 059 | D Loss >> -0.4726
Epoch 060 | D Loss >> -0.4792
Epoch 061 | D Loss >> -0.4767
Epoch 062 | D Loss >> -0.4764
Epoch 063 | D Loss >> -0.4804
Epoch 064 | D Loss >> -0.4753
Epoch 065 | D Loss >> -0.4711
Epoch 066 | D Loss >> -0.4773
Epoch 067 | D Loss >> -0.4741
```

```
Epoch 068 | D Loss >> -0.4761
Epoch 069 | D Loss >> -0.4802
Epoch 070 | D Loss >> -0.4797
Epoch 071 | D Loss >> -0.4778
Epoch 072 | D Loss >> -0.4747
Epoch 073 | D Loss >> -0.4769
Epoch 074 | D Loss >> -0.4791
Epoch 075 | D Loss >> -0.4804
Epoch 076 | D Loss >> -0.4789
Epoch 077 | D Loss >> -0.4824
```

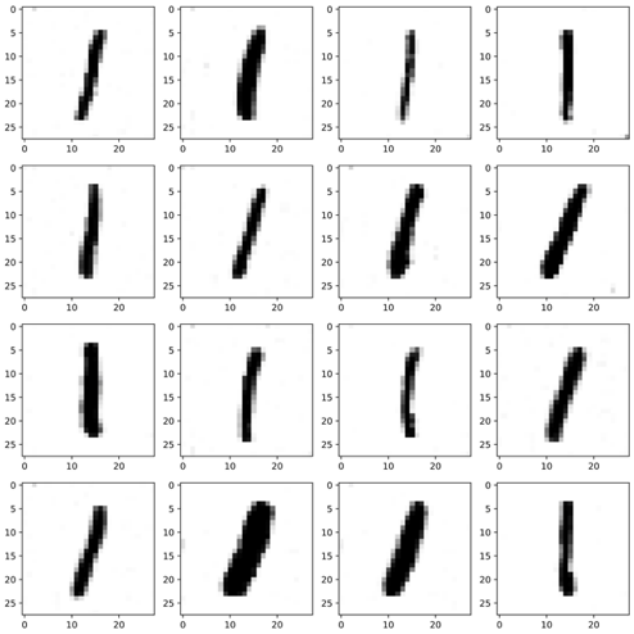
[ ]:
```
selected_epochs = [1, 2, 4, 10, 50, 100]
# selected_epochs = [1, 10, 20, 30, 50, 70]
fig = plt.figure(figsize=(10, 14))
for i,e in enumerate(selected_epochs):
    for j in range(5):
        ax = fig.add_subplot(6, 5, i*5+j+1)
        ax.set_xticks([])
        ax.set_yticks([])
        if j == 0:
            ax.text(
                -0.06, 0.5, f'Epoch {e}',
                rotation=90, size=18, color='red',
                horizontalalignment='right',
                verticalalignment='center',
                transform=ax.transAxes)

        image = epoch_samples_wgan[e-1][j]
        ax.imshow(image, cmap='gray_r')

# plt.savefig('figures/ch17-wgan-gp-samples.pdf')
plt.show()
```

Mode collapse

Due to the adversarial nature of GAN models, it is notoriously hard to train them. One common cause of failure in training GANs is when the generator gets stuck in a small subspace and learns to generate similar samples. This is called **mode collapse**.

[2]:
```
Image(filename='figures/17_16.png', width=600)
```

[2]:



To improve training of GANs, there are several tricks.

1. **Mini-batch discrimination**: Feeding batches consisting of only real or fake examples separately to the discriminator.
2. **Feature matching**: Slightly modifiy the objective function of the generator by adding an extra term that minimises the difference between the original and synthesised images based on intermediate representations (feature maps) of the discriminator.
3. **Experience replay**: GAN model can also get stuck in several modes and just hop between them. To avoid this behaviour, store some old examples and feed them to the discriminator to prevent the generator from revisiting previous modes.
4. Train multiple GANs with different random seeds so that the combination of all of them covers a larger part of the data distribution than any single one of them.