Data Science in Electron Microscopy

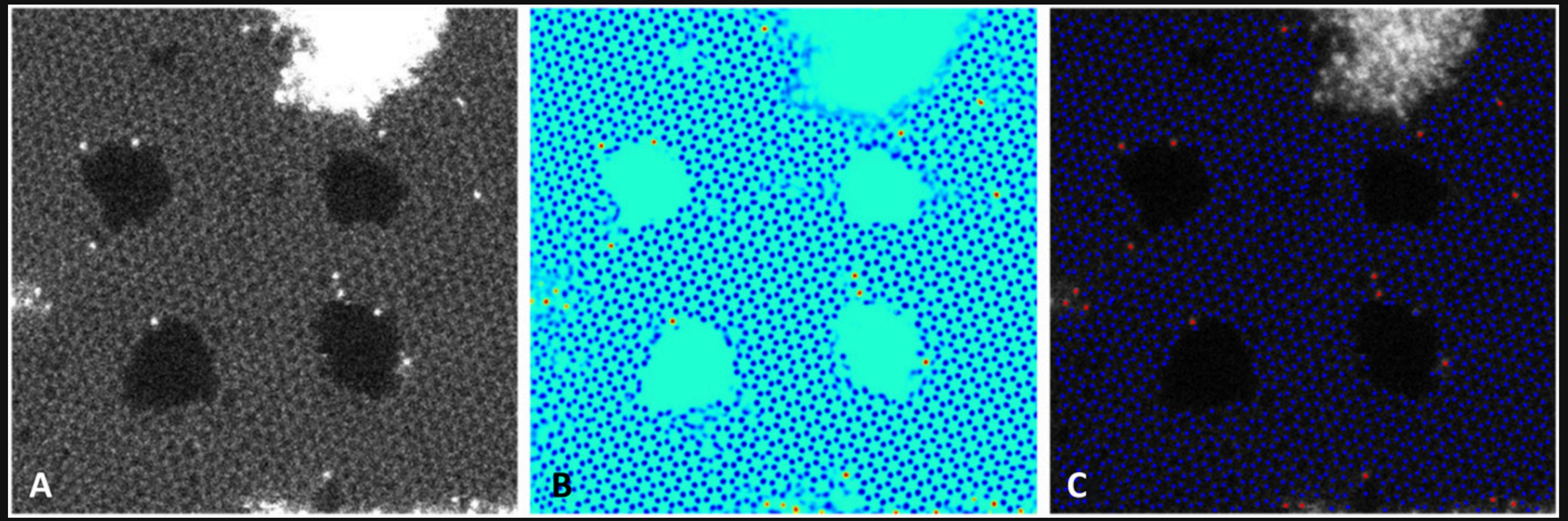
Philipp Pelz

2024

https://github.com/ECLIPSE-Lab/WS24_DataScienceForEM

VAE example: Exploring Order Parameters and Dynamic Processes in Disordered Systems via Variational Autoencoders

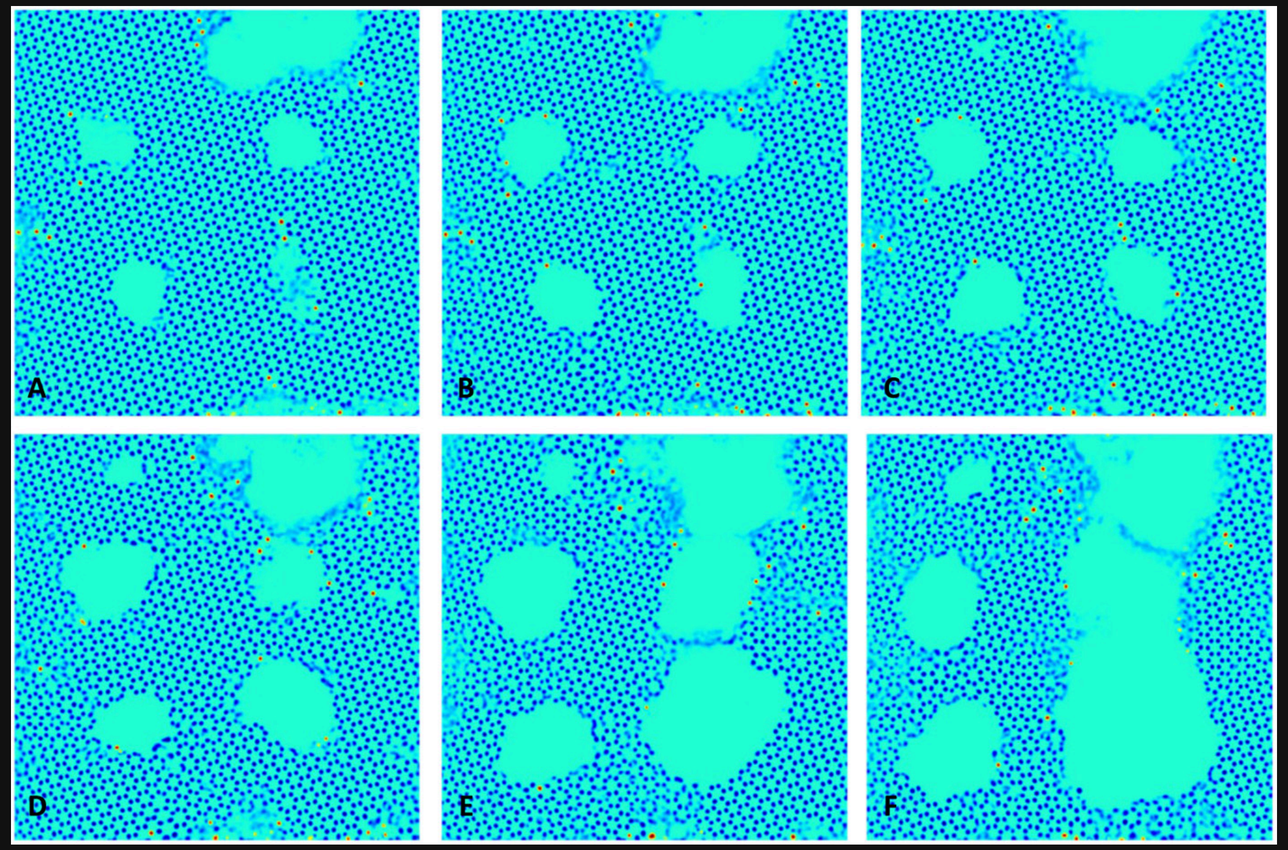
Authors: Sergei V. Kalinin, Ondrej Dyck, Stephen Jesse, Maxim Ziatdinov
Published in: Science Advances (2021)
DOI: 10.1126/sciadv.abd5084



Single image from dynamic STEM dataset corresponding to 10th frame

Introduction

- **Objective**: Analyze dynamic processes and order parameters in disordered systems.
- **Approach**: Use rotationally invariant variational autoencoders (rVAEs).
- **Application**: Studied e-beam induced processes in silicon-doped graphene.



Evolution of graphene under e-beam irradiation.

Rotationally Invariant VAEs

- **Purpose**: Handle rotational invariance in noncrystalline solids.
- **Method**:
  - Incorporate rotational and translational invariance.
  - Apply rVAEs to semantically segmented, atomically resolved data.
- **Benefit**: Captures maximum original information with reduced representation.

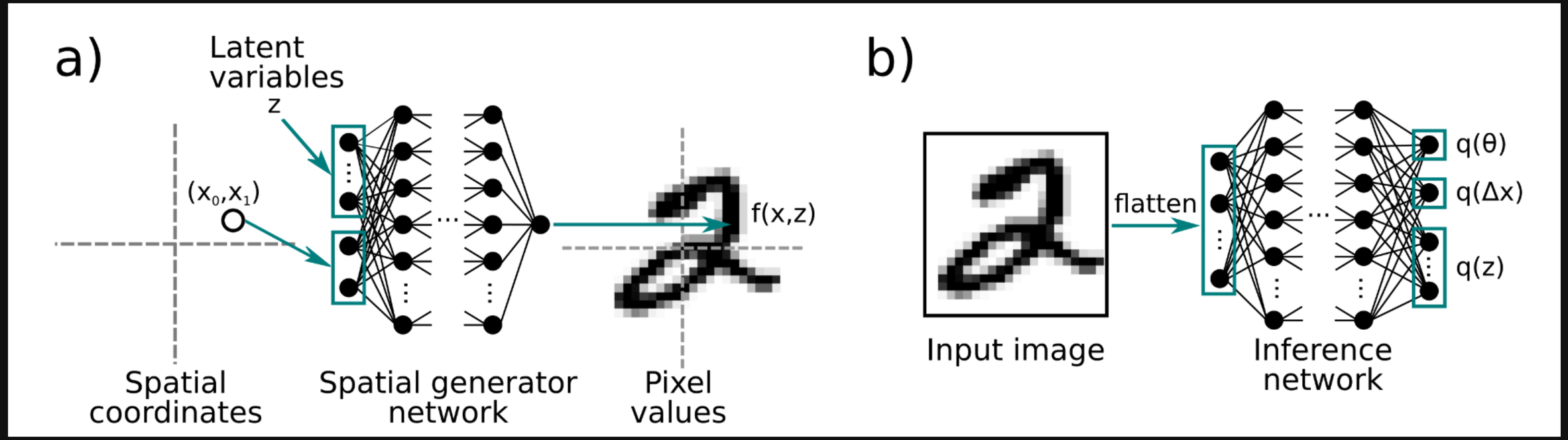


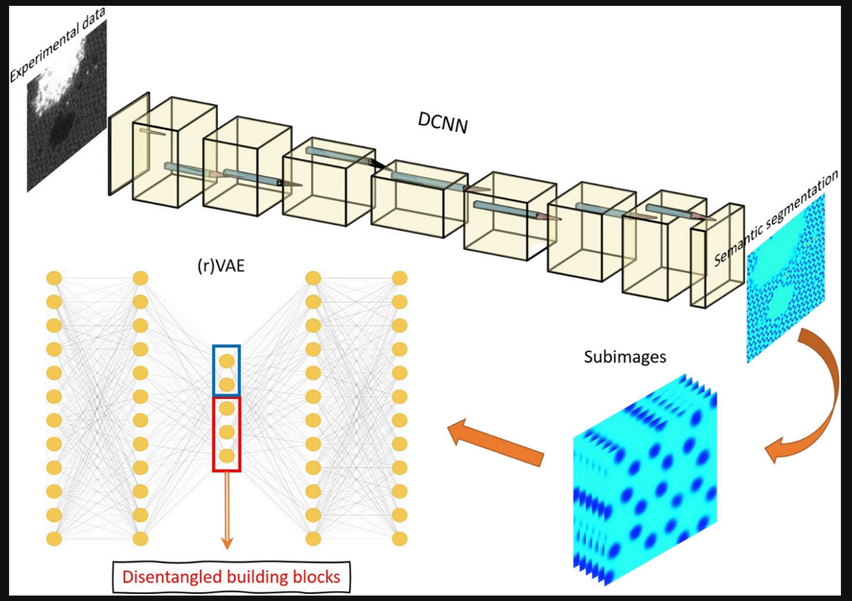
Diagram of the spatial-VAE framework

Rotationally Invariant VAEs Forward

```python
class SpatialGenerator(nn.Module):
    def forward(self, x, z):
        # x is (batch, num_coords, 2)
        # z is (batch, latent_dim)

        if len(x.size()) < 3:
            x = x.unsqueeze(0)
        b = x.size(0)
        n = x.size(1)
        x = x.view(b*n, -1)

        h_x = self.coord_linear(x)
        h_x = h_x.view(b, n, -1)

        h_z = 0

        if len(z.size()) < 2:
            z = z.unsqueeze(0)
        h_z = self.latent_linear(z)
```

Experimental Setup

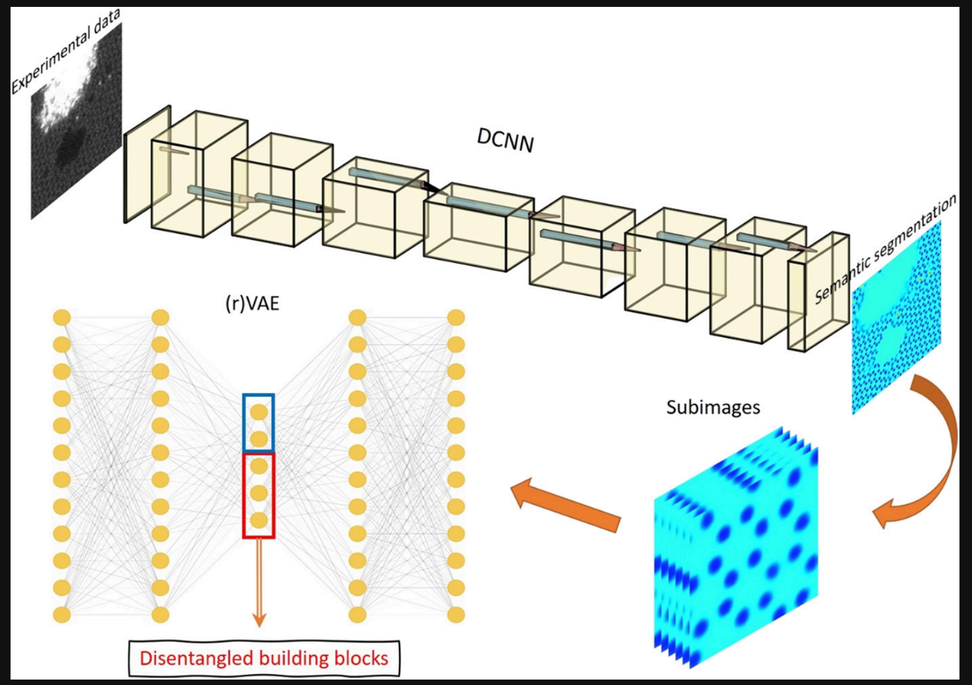
- **Sample**: Silicon-doped graphene.
- **Imaging**: Scanning transmission electron microscopy (STEM).
- **Procedure**:
  - Use DCNN for initial pixel probability maps.
  - Extract atomic positions for VAE analysis.
- **Data**: Multiple snapshots during dynamic processes.



Schematic of the overall approach.

Data Analysis with rVAE

- **Workflow**:
  1. DCNN categorizes pixels into atomic types.
  2. Generate subimages centered on atomic positions.
  3. rVAE seeks the most effective reduced representation.
- **Output**: Identifies key structural elements and their dynamics.



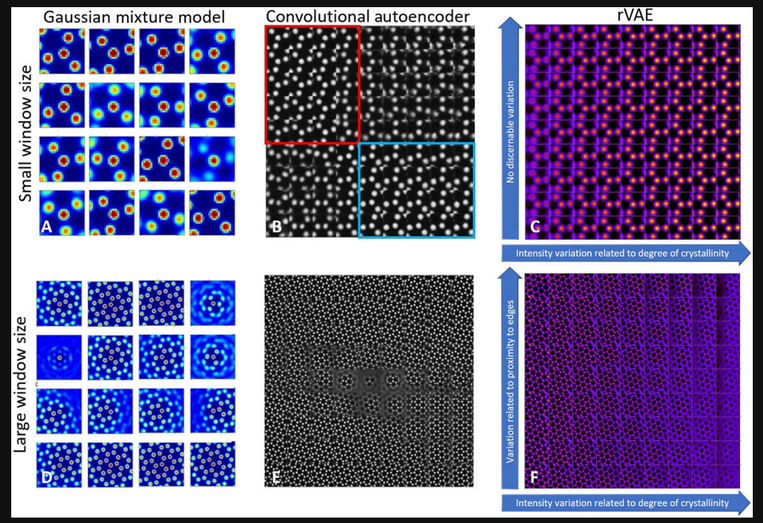
Workflow

Results

- **Findings**:
  - Effective exploration of chemical evolution in the system.
  - rVAE captured rotationally invariant features.

Comparison of methods for construction of elementary descriptors.

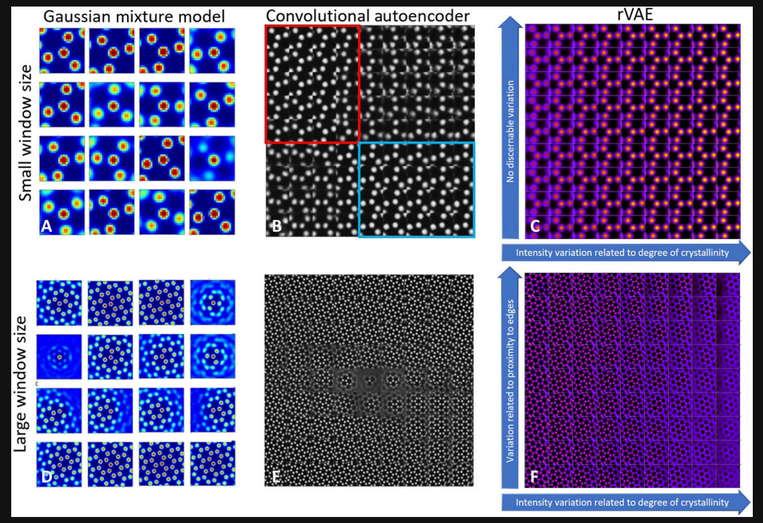
- (A to C) small and (D to F) large windows.
- (A and D) GMM classes of the data that are decomposed into many independent components that are statistical in nature and often do not allow for direct physical interpretation. In (D), this approach performs poorly at capturing rotation and spreads this information across several components. (B and E) Representation in 2D latent space of convolutional AE. Red and blue regions in (B) indicate clear separation of graphene sublattices with remainder of the descriptors encoding lateral shifts, defects, and rotations in a convoluted fashion.



Comparison of methods for construction of elementary descriptors.

Comparison of methods for construction of elementary descriptors.

- In (E), the larger window introduces variability that is more difficult to interpret. (C and F) Representation in the 2D latent space of rotationally invariant VAE. Because rotational variation is removed from elementary descriptors, remaining variations within data can be described much more efficiently.
- In (C), there are noticeable changes in only one dimension, which can be ascribed to degree of local crystallinity. In (F), the larger window size also captures variations related to proximity of edges. In (B), (C), (E), and (F), the images were generated by applying a corresponding decoder to the uniform grid of discrete point in the latent space.



Comparison of methods for construction of elementary descriptors.

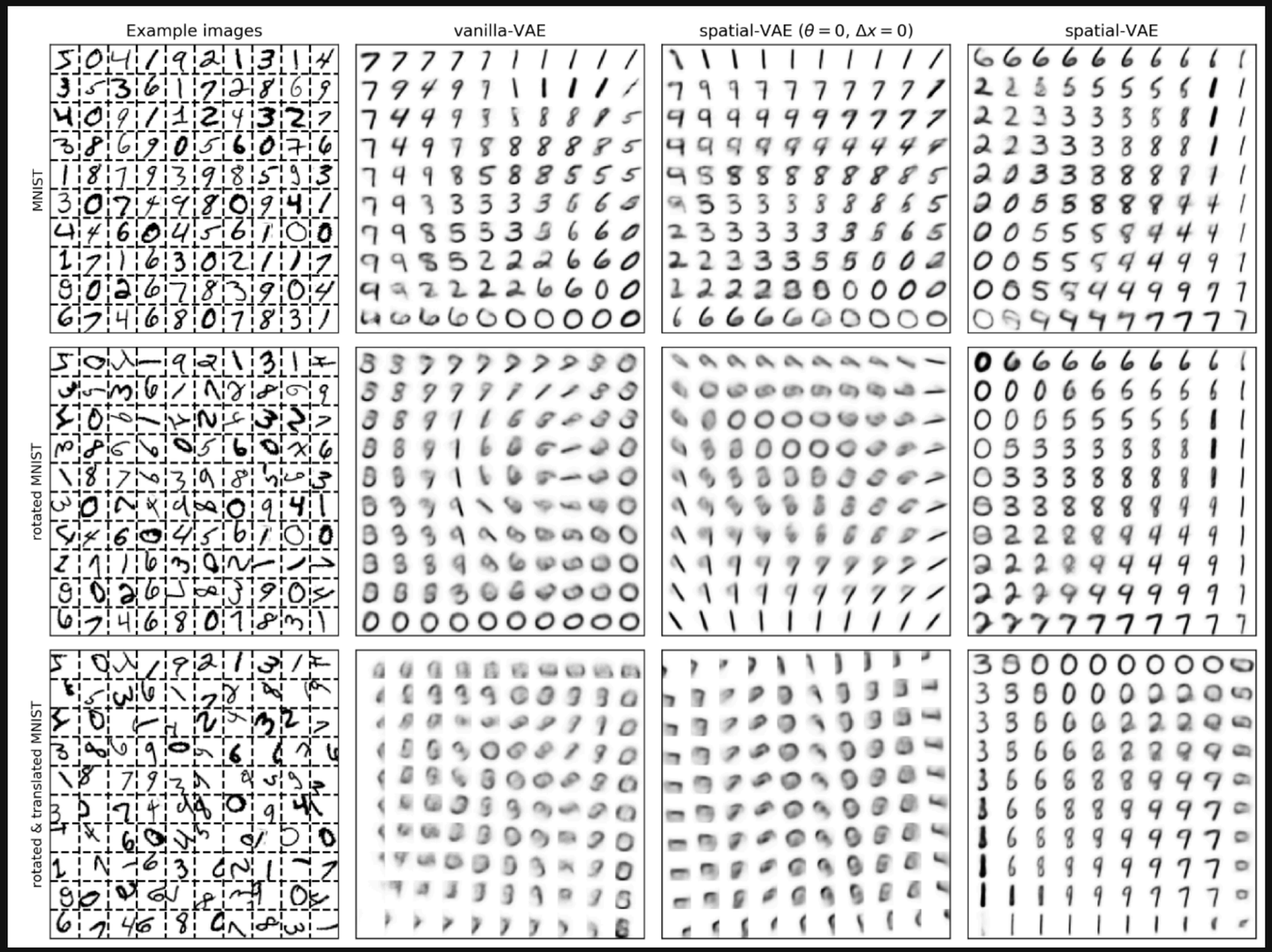
Results

- **Evaluation**:
  - Identified structural changes due to e-beam.
  - Analyzed formation of 5-7 member defect chains.
  - Detected migration of Si atoms to graphene edges.

Comparison with Other Methods

- **GMM Analysis**:
  - Generates independent components.
  - Effective for imaging but less interpretable structurally.
- **Classical AE**:
  - Reduced data to continuous latent variables.
  - Convoluted rotation and structural variations.
- **rVAE**:
  - Separated rotation and structural changes.
  - Provided clear physical interpretation.

Comparison with Other Methods 2



Different embeddings

Conclusion

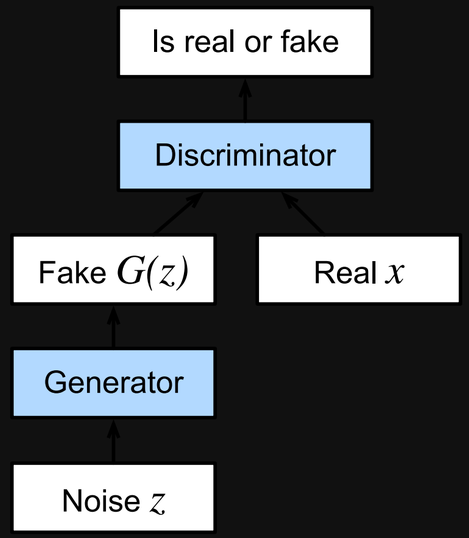
- **Significance**:
  - rVAEs provide a robust framework for analyzing disordered systems.
  - Effective for bottom-up description of dynamic processes.

Generative Adversarial Networks

:label:`sec_basic_gan`

- **Discriminative Learning:**
  - Predicts labels from data examples.
  - Examples: classifiers and regressors.
  - Deep neural networks have revolutionized discriminative learning, achieving human-level accuracy on high-res images.
- **Generative Modeling:**
  - Learns a model to capture data characteristics without labels.
  - Generates synthetic data resembling the training dataset.
  - Example: Generating photorealistic images from a dataset of faces.

- **Deep Neural Networks in Generative Modeling:**
  - Recent advances have enabled discriminative models to assist generative tasks.
  - Example: Recurrent neural network language models.
- **Generative Adversarial Networks (GANs):**
  - Introduced in 2014 by Goodfellow et al.
  - Leverages discriminative models to create generative models.
  - Concept: A good data generator makes fake data indistinguishable from real data (two-sample test).
  - GANs use the two-sample test constructively to train generative models.
  - Aim: Improve the generator until it fools a state-of-the-art classifier.



fig_gan

The GAN architecture is illustrated in :numref:fig_gan.

- **GAN Architecture:**
  - **Generator Network:**
    - Generates data resembling real data.
    - For images: generates images.
    - For speech: generates audio sequences.
  - **Discriminator Network:**
    - Distinguishes fake data from real data.
    - Competes with the generator.
    - Adaptively improves to distinguish better as the generator improves.

- **Discriminator:**
  - Binary classifier: distinguishes real (x) vs. fake data.
  - Outputs scalar prediction $o \in \mathbb{R}$ for input $\mathbf{x}$.
  - Applies sigmoid function: $D(\mathbf{x}) = \frac{1}{1+e^{-o}}$.
  - True data label $y = 1$, fake data label $y = 0$.
  - Minimize cross-entropy loss:

$$\min_D \{-y \log D(\mathbf{x}) - (1-y) \log(1 - D(\mathbf{x}))\}$$

- **Generator:**
  - Draws parameter $\mathbf{z} \in \mathbb{R}^d$ from randomness, e.g., $\mathbf{z} \sim \mathcal{N}(0, 1)$ (latent variable).
  - Generates data: $\mathbf{x}' = G(\mathbf{z})$.
  - Aims to fool the discriminator: $D(G(\mathbf{z})) \approx 1$.
  - Update parameters to maximize cross-entropy loss for $y = 0$:

$$\max_G \{-\log(1 - D(G(\mathbf{z})))\}$$

  - Commonly minimize the loss:

$$\min_G \{-\log(D(G(\mathbf{z})))\}$$

    - This is feeding $\mathbf{x}' = G(\mathbf{z})$ into the discriminator but giving label $y = 1$.

- **Minimax Game:**
  - D and G play a “minimax” game with the objective function:

$$\min_D \max_G \{-E_{x \sim \text{Data}} \log D(\mathbf{x}) - E_{z \sim \text{Noise}} \log(1 - D(G(\mathbf{z})))\}$$

- **Applications:**
  - Many GAN applications are in the context of images.
  - Demonstration: fitting a simpler distribution.
  - Example: Using GANs to estimate parameters for a Gaussian.

Let’s get started.

```
from d2l import torch as d2l
import torch
from torch import nn
```

Generate Some “Real” Data

Since this is going to be the world’s lamest example, we simply generate data drawn from a Gaussian.

```
X = d2l.normal(0.0, 1, (1000, 2))
A = d2l.tensor([[1, 2], [-0.1, 0.5]])
b = d2l.tensor([1, 2])
data = d2l.matmul(X, A) + b
```

Let’s see what we got. This should be a Gaussian shifted in some rather arbitrary way with mean b and covariance matrix $A^T A$.

```

d2l.set_figsize()
d2l.plt.scatter(d2l.numpy(data[:100, 0]), d2l.numpy(data[:100, 1]));
print(f'The covariance matrix is\n{d2l.matmul(A.T, A)}')
```

```
batch_size = 8
data_iter = d2l.load_array((data,), batch_size)
```

Generator

Our generator network will be the simplest network possible - a single layer linear model. This is since we will be driving that linear network with a Gaussian data generator. Hence, it literally only needs to learn the parameters to fake things perfectly.

```
net_G = nn.Sequential(nn.Linear(2, 2))
```

Discriminator

For the discriminator we will be a bit more discriminating: we will use an MLP with 3 layers to make things a bit more interesting.

```
net_D = nn.Sequential(
    nn.Linear(2, 5), nn.Tanh(),
    nn.Linear(5, 3), nn.Tanh(),
    nn.Linear(3, 1))
```

Training

First we define a function to update the discriminator.

```python
def update_D(X, Z, net_D, net_G, loss, trainer_D):
    """Update discriminator."""
    batch_size = X.shape[0]
    ones = torch.ones((batch_size,), device=X.device)
    zeros = torch.zeros((batch_size,), device=X.device)
    trainer_D.zero_grad()
    real_Y = net_D(X)
    fake_X = net_G(Z)
    # Do not need to compute gradient for `net_G`, detach it from
    # computing gradients.
    fake_Y = net_D(fake_X.detach())
    loss_D = (loss(real_Y, ones.reshape(real_Y.shape)) +
              loss(fake_Y, zeros.reshape(fake_Y.shape))) / 2
    loss_D.backward()
    trainer_D.step()
    return loss_D
```

The generator is updated similarly. Here we reuse the cross-entropy loss but change the label of the fake data from 0 to 1.

```python
def update_G(Z, net_D, net_G, loss, trainer_G):
    """Update generator."""
    batch_size = Z.shape[0]
    ones = torch.ones((batch_size,), device=Z.device)
    trainer_G.zero_grad()
    # We could reuse `fake_X` from `update_D` to save computation
    fake_X = net_G(Z)
    # Recomputing `fake_Y` is needed since `net_D` is changed
    fake_Y = net_D(fake_X)
    loss_G = loss(fake_Y, ones.reshape(fake_Y.shape))
    loss_G.backward()
    trainer_G.step()
    return loss_G
```

Both the discriminator and the generator performs a binary logistic regression with the cross-entropy loss. We use Adam to smooth the training process. In each iteration, we first update the discriminator and then the generator. We visualize both losses and generated examples.

```
def train(net_D, net_G, data_iter, num_epochs, lr_D, lr_G, latent_dim, data):
    loss = nn.BCEWithLogitsLoss(reduction='sum')
    for w in net_D.parameters():
        nn.init.normal_(w, 0, 0.02)
    for w in net_G.parameters():
        nn.init.normal_(w, 0, 0.02)
    trainer_D = torch.optim.Adam(net_D.parameters(), lr=lr_D)
    trainer_G = torch.optim.Adam(net_G.parameters(), lr=lr_G)
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[1, num_epochs], nrows=2, figsize=(5, 5),
                            legend=['discriminator', 'generator'])
    animator.fig.subplots_adjust(hspace=0.3)
    for epoch in range(num_epochs):
        # Train one epoch
        timer = d2l.Timer()
        metric = d2l.Accumulator(3)  # loss_D, loss_G, num_examples
        for (X,) in data_iter:
            batch_size = X.shape[0]
            Z = torch.normal(0, 1, size=(batch_size, latent_dim))
```

Now we specify the hyperparameters to fit the Gaussian distribution.

```
lr_D, lr_G, latent_dim, num_epochs = 0.05, 0.005, 2, 20
# train(net_D, net_G, data_iter, num_epochs, lr_D, lr_G,
#       latent_dim, d2l.numpy(data[:100]))
```

Summary

- Generative adversarial networks (GANs) composes of two deep networks, the generator and the discriminator.
- The generator generates the image as much closer to the true image as possible to fool the discriminator, via maximizing the cross-entropy loss, *i.e.*, $\max \log(D(\mathbf{x}'))$.
- The discriminator tries to distinguish the generated images from the true images, via minimizing the cross-entropy loss, *i.e.*, $\min -y \log D(\mathbf{x}) - (1-y) \log(1-D(\mathbf{x}))$.

Exercises

- Does an equilibrium exist where the generator wins, *i.e.* the discriminator ends up unable to distinguish the two distributions on finite samples?

Deep Convolutional Generative Adversarial Networks

- **Introduction to GANs:**
  - Basic idea: Transform samples from simple distributions (uniform, normal) to match dataset distributions.
  - Previous example: Matching a 2D Gaussian distribution.
- **Photorealistic Image Generation:**
  - Use GANs to generate photorealistic images.
  - Based on deep convolutional GANs (DCGAN) from Radford et al. (2015).

- **DCGAN Architecture:**
  - Leverages convolutional architecture.
  - Successful for discriminative computer vision tasks.
  - Adapted for generative tasks to produce realistic images.

```
from d2l import torch as d2l
import torch
import torchvision
from torch import nn
import warnings
```

The Pokemon Dataset

The dataset we will use is a collection of Pokemon sprites obtained from pokemondb. First download, extract and load this dataset.

```
# d2l.DATA_HUB['pokemon'] = (d2l.DATA_URL + 'pokemon.zip',
#                            'c065c0e2593b8b161a2d7873e42418bf6a21106c')

# data_dir = d2l.download_extract('pokemon')
# print(data_dir)
# pokemon = torchvision.datasets.ImageFolder(data_dir)
pokemon = torchvision.datasets.EMNIST(root='./', split='digits', download=True)
```

We resize each image into 64×64. The `ToTensor` transformation will project the pixel value into $[0, 1]$, while our generator will use the tanh function to obtain outputs in $[-1, 1]$. Therefore we normalize the data with 0.5 mean and 0.5 standard deviation to match the value range.

```
batch_size = 256
transformer = torchvision.transforms.Compose([
    torchvision.transforms.Resize((64, 64)),
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize(0.5, 0.5)
])
pokemon.transform = transformer
data_iter = torch.utils.data.DataLoader(
    pokemon, batch_size=batch_size,
    shuffle=True, num_workers=1)
```

```
warnings.filterwarnings('ignore')
d2l.set_figsize((4, 4))
for X, y in data_iter:
    imgs = X[:20,:,:,:].permute(0, 2, 3, 1)/2+0.5
    d2l.show_images(imgs, num_rows=4, num_cols=5)
    break
```

The Generator

The generator needs to map the noise variable $\mathbf{z} \in \mathbb{R}^d$, a length-$d$ vector, to a RGB image with width and height to be 64×64 . In :numref:sec_fcn we introduced the fully convolutional network that uses transposed convolution layer (refer to :numref:sec_transposed_conv) to enlarge input size. The basic block of the generator contains a transposed convolution layer followed by the batch normalization and ReLU activation.

```
class G_block(nn.Module):
    def __init__(self, out_channels, in_channels=3, kernel_size=4, strides=2,
                 padding=1, **kwargs):
        super(G_block, self).__init__(**kwargs)
        self.conv2d_trans = nn.ConvTranspose2d(in_channels, out_channels,
                                kernel_size, strides, padding, bias=False)
        self.batch_norm = nn.BatchNorm2d(out_channels)
        self.activation = nn.ReLU()

    def forward(self, X):
        return self.activation(self.batch_norm(self.conv2d_trans(X)))
```

In default, the transposed convolution layer uses a $k_h = k_w = 4$ kernel, a $s_h = s_w = 2$ strides, and a $p_h = p_w = 1$ padding. With a input shape of $n_h' \times n_w' = 16 \times 16$, the generator block will double input's width and height.

$$
\begin{aligned}
n_h' \times n_w' &= [(n_h k_h - (n_h - 1)(k_h - s_h) - 2p_h] \times [(n_w k_w - (n_w - 1)(k_w - s_w) - 2p_w] \\
&= [(k_h + s_h(n_h - 1) - 2p_h] \times [(k_w + s_w(n_w - 1) - 2p_w] \\
&= [(4 + 2 \times (16 - 1) - 2 \times 1] \times [(4 + 2 \times (16 - 1) - 2 \times 1] \\
&= 32 \times 32.
\end{aligned}
$$

```
1
2 x = torch.zeros((2, 3, 16, 16))
3 g_blk = G_block(20)
4 g_blk(x).shape
```

If changing the transposed convolution layer to a 4×4 kernel, 1×1 strides and zero padding. With a input size of 1×1, the output will have its width and height increased by 3 respectively.

```
1
2 x = torch.zeros((2, 3, 1, 1))
3 g_blk = G_block(20, strides=1, padding=0)
4 g_blk(x).shape
```

The generator consists of four basic blocks that increase input's both width and height from 1 to 32. At the same time, it first projects the latent variable into 64×8 channels, and then halve the channels each time. At last, a transposed convolution layer is used to generate the output. It further doubles the width and height to match the desired 64×64 shape, and reduces the channel size to 3. The tanh activation function is applied to project output values into the $(-1, 1)$ range.

```
n_G = 64
net_G = nn.Sequential(
    G_block(in_channels=100, out_channels=n_G*8,
            strides=1, padding=0),                  # Output: (64 * 8, 4, 4)
    G_block(in_channels=n_G*8, out_channels=n_G*4), # Output: (64 * 4, 8, 8)
    G_block(in_channels=n_G*4, out_channels=n_G*2), # Output: (64 * 2, 16, 16)
    G_block(in_channels=n_G*2, out_channels=n_G),   # Output: (64, 32, 32)
    nn.ConvTranspose2d(in_channels=n_G, out_channels=3,
                       kernel_size=4, stride=2, padding=1, bias=False),
    nn.Tanh())  # Output: (3, 64, 64)
```

Generate a 100 dimensional latent variable to verify the generator’s output shape.

```

x = torch.zeros((1, 100, 1, 1))
net_G(x).shape
```

Discriminator

The discriminator is a normal convolutional network network except that it uses a leaky ReLU as its activation function. Given $\alpha \in [0, 1]$, its definition is

$$\text{leaky ReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{otherwise} \end{cases}.$$

As it can be seen, it is normal ReLU if $\alpha = 0$, and an identity function if $\alpha = 1$. For $\alpha \in (0, 1)$, leaky ReLU is a nonlinear function that give a non-zero output for a negative input. It aims to fix the “dying ReLU” problem that a neuron might always output a negative value and therefore cannot make any progress since the gradient of ReLU is 0.

```
1
2 alphas = [0, .2, .4, .6, .8, 1]
3 x = d2l.arange(-2, 1, 0.1)
4 Y = [d2l.numpy(nn.LeakyReLU(alpha)(x)) for alpha in alphas]
5 d2l.plot(d2l.numpy(x), Y, 'x', 'y', alphas)
```

The basic block of the discriminator is a convolution layer followed by a batch normalization layer and a leaky ReLU activation. The hyperparameters of the convolution layer are similar to the transpose convolution layer in the generator block.

```
class D_block(nn.Module):
    def __init__(self, out_channels, in_channels=3, kernel_size=4, strides=2,
                 padding=1, alpha=0.2, **kwargs):
        super(D_block, self).__init__(**kwargs)
        self.conv2d = nn.Conv2d(in_channels, out_channels, kernel_size,
                                strides, padding, bias=False)
        self.batch_norm = nn.BatchNorm2d(out_channels)
        self.activation = nn.LeakyReLU(alpha, inplace=True)

    def forward(self, X):
        return self.activation(self.batch_norm(self.conv2d(X)))
```

A basic block with default settings will halve the width and height of the inputs, as we demonstrated in :numref:sec_padding. For example, given a input shape $n_h = n_w = 16$, with a kernel shape $k_h = k_w = 4$, a stride shape $s_h = s_w = 2$, and a padding shape $p_h = p_w = 1$, the output shape will be:

$$\begin{aligned} n'_h \times n'_w &= \lfloor (n_h - k_h + 2p_h + s_h)/s_h \rfloor \times \lfloor (n_w - k_w + 2p_w + s_w)/s_w \rfloor \\ &= \lfloor (16 - 4 + 2 \times 1 + 2)/2 \rfloor \times \lfloor (16 - 4 + 2 \times 1 + 2)/2 \rfloor \\ &= 8 \times 8. \end{aligned}$$

```

x = torch.zeros((2, 3, 16, 16))
d_blk = D_block(20)
d_blk(x).shape
```

```
n_D = 64
net_D = nn.Sequential(
    D_block(n_D),  # Output: (64, 32, 32)
    D_block(in_channels=n_D, out_channels=n_D*2),  # Output: (64 * 2, 16, 16)
    D_block(in_channels=n_D*2, out_channels=n_D*4),  # Output: (64 * 4, 8, 8)
    D_block(in_channels=n_D*4, out_channels=n_D*8),  # Output: (64 * 8, 4, 4)
    nn.Conv2d(in_channels=n_D*8, out_channels=1,
              kernel_size=4, bias=False))  # Output: (1, 1, 1)
```

It uses a convolution layer with output channel 1 as the last layer to obtain a single prediction value.

```
x = torch.zeros((1, 3, 64, 64))
net_D(x).shape
```

Training

Compared to the basic GAN in :numref:`sec_basic_gan`, we use the same learning rate for both generator and discriminator since they are similar to each other. In addition, we change β_1 in Adam (:numref:`sec_adam`) from 0.9 to 0.5. It decreases the smoothness of the momentum, the exponentially weighted moving average of past gradients, to take care of the rapid changing gradients because the generator and the discriminator fight with each other. Besides, the random generated noise `Z`, is a 4-D tensor and we are using GPU to accelerate the computation.

```
def train(net_D, net_G, data_iter, num_epochs, lr, latent_dim,
          device=d2l.try_gpu()):
    loss = nn.BCEWithLogitsLoss(reduction='sum')
    for w in net_D.parameters():
        nn.init.normal_(w, 0, 0.02)
    for w in net_G.parameters():
        nn.init.normal_(w, 0, 0.02)
    net_D, net_G = net_D.to(device), net_G.to(device)
    trainer_hp = {'lr': lr, 'betas': [0.5,0.999]}
    trainer_D = torch.optim.Adam(net_D.parameters(), **trainer_hp)
    trainer_G = torch.optim.Adam(net_G.parameters(), **trainer_hp)
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[1, num_epochs], nrows=2, figsize=(5, 5),
                            legend=['discriminator', 'generator'])
    animator.fig.subplots_adjust(hspace=0.3)
    for epoch in range(1, num_epochs + 1):
        # Train one epoch
        timer = d2l.Timer()
        metric = d2l.Accumulator(3)  # loss_D, loss_G, num_examples
```

We train the model with a small number of epochs just for demonstration. For better performance, the variable `num_epochs` can be set to a larger number.

```
latent_dim, lr, num_epochs = 100, 0.005, 20
# train(net_D, net_G, data_iter, num_epochs, lr, latent_dim)
```

Summary

- DCGAN architecture has four convolutional layers for the Discriminator and four “fractionally-strided” convolutional layers for the Generator.
- The Discriminator is a 4-layer strided convolutions with batch normalization (except its input layer) and leaky ReLU activations.
- Leaky ReLU is a nonlinear function that give a non-zero output for a negative input. It aims to fix the “dying ReLU” problem and helps the gradients flow easier through the architecture.

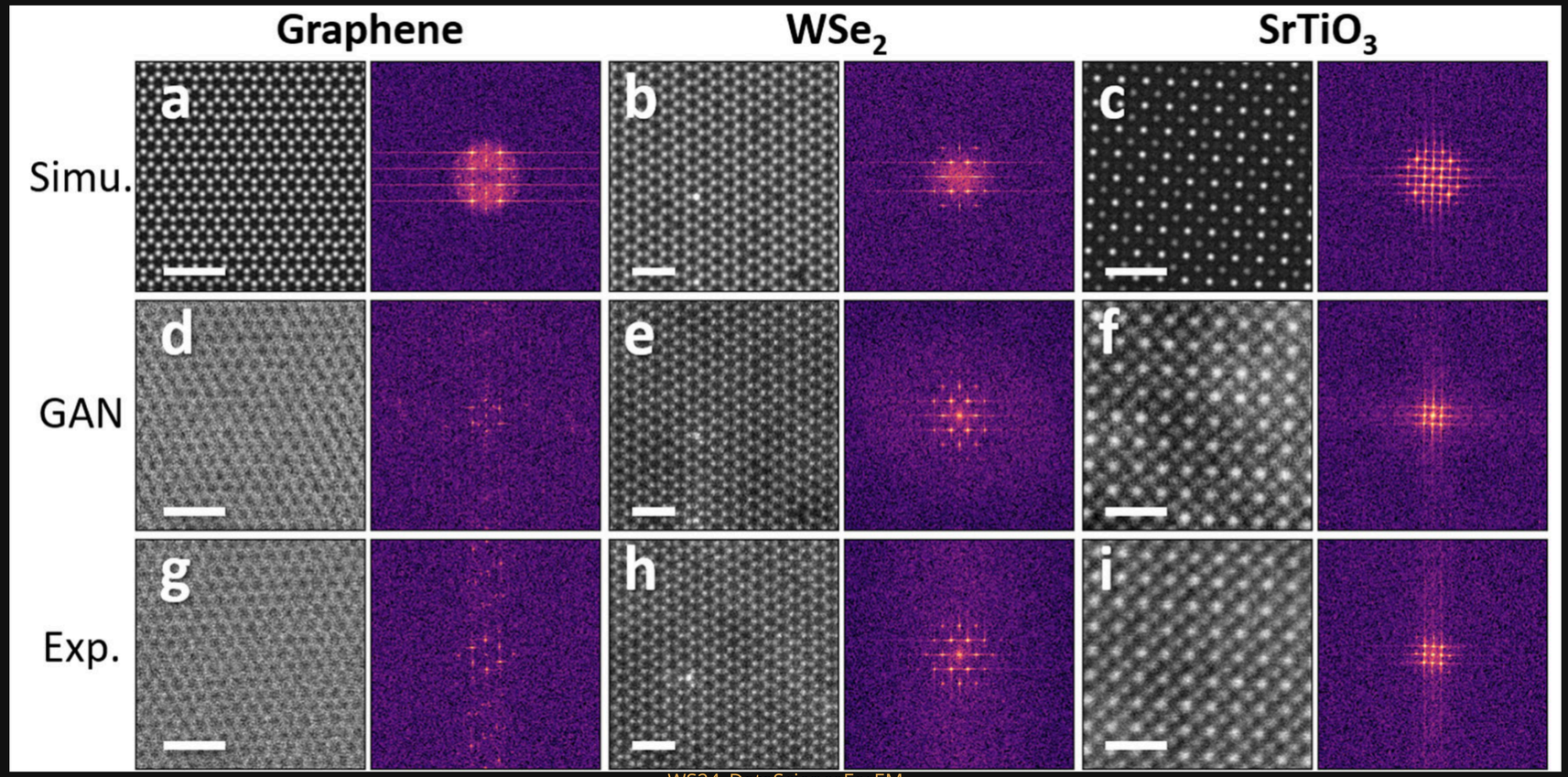
Exercises

1. What will happen if we use standard ReLU activation rather than leaky ReLU?
2. Apply DCGAN on Fashion-MNIST and see which category works well and which does not.

Application examples

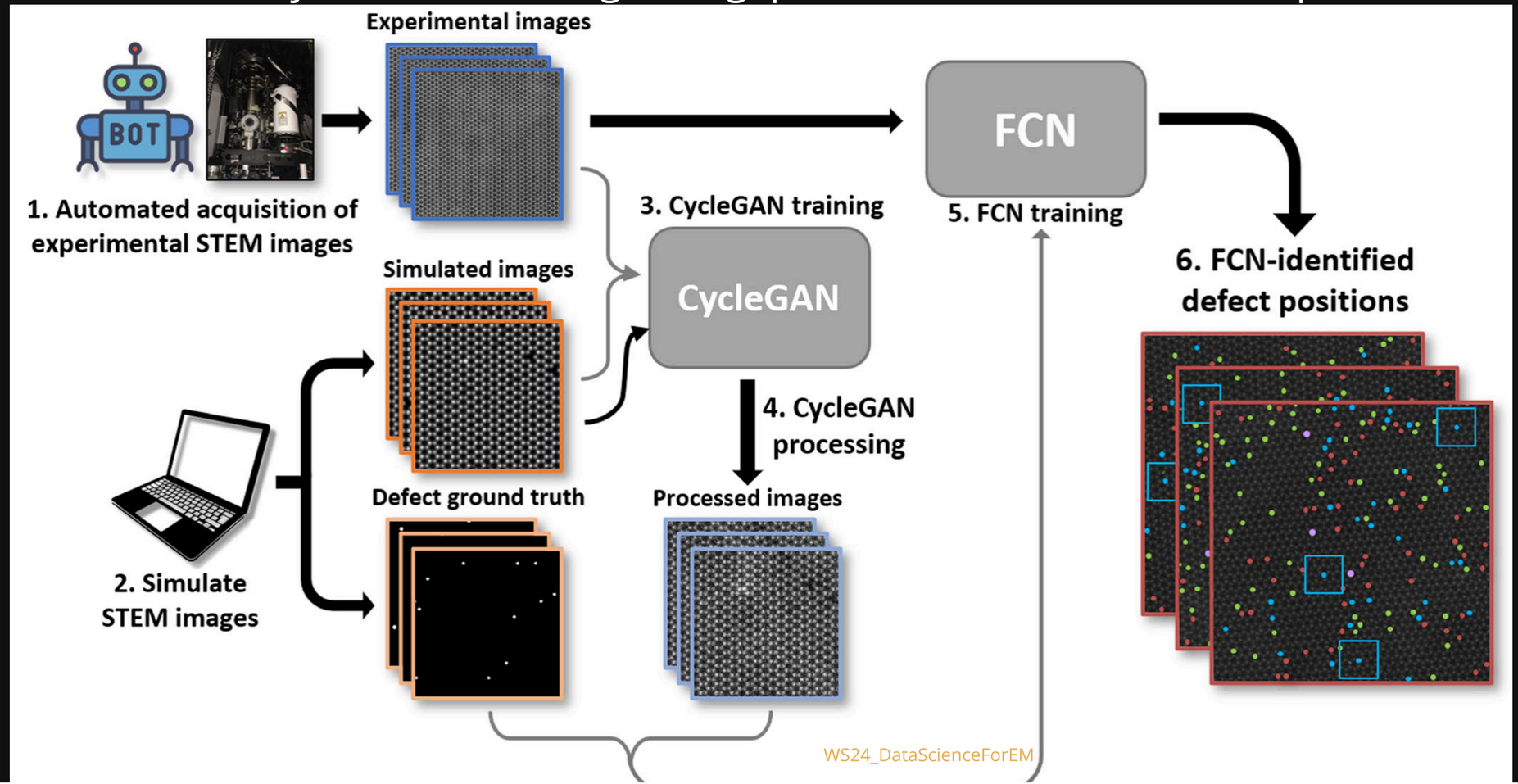
Example 1: Leveraging Generative Adversarial Networks to Create Realistic Scanning Transmission Electron Microscopy Images

Authors: Abid Khan, Chia-Hao Lee, Pinshane Y. Huang, Bryan K. Clark
Published in: npj Computational Materials (2023)
DOI: 10.1038/s41524-023-01042-3



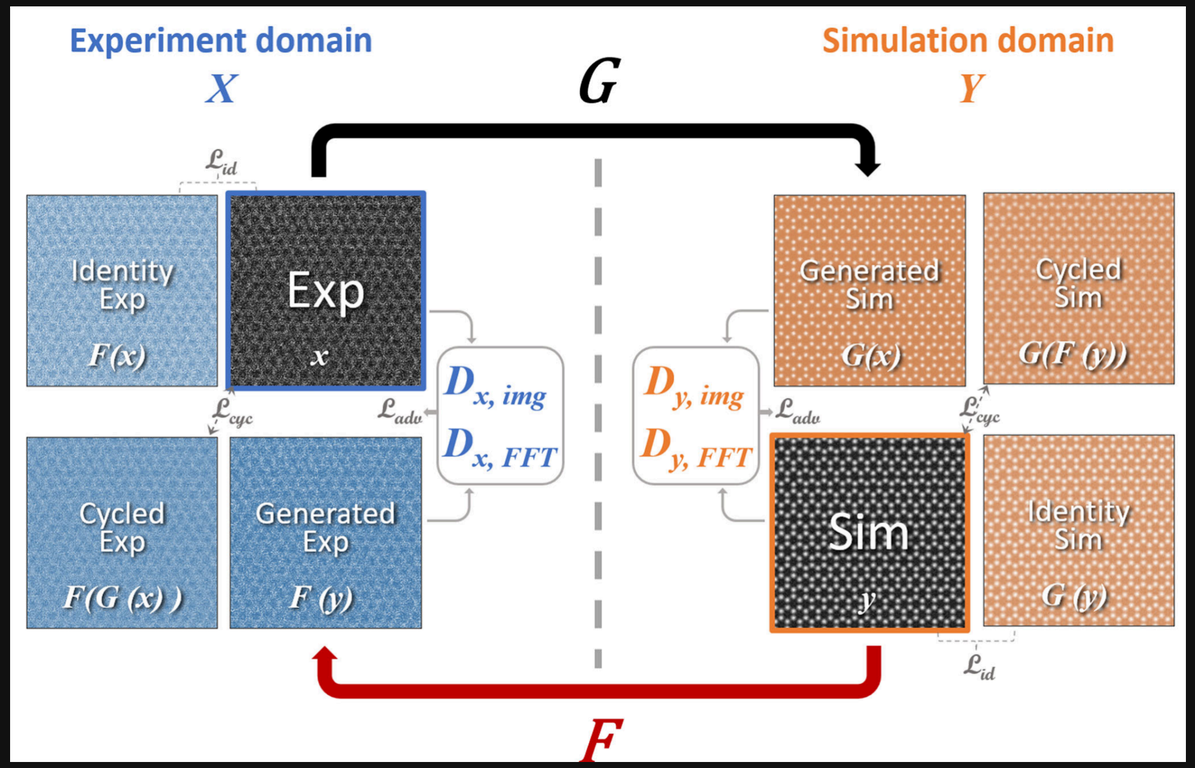
Introduction

- Machine learning (ML) in electron microscopy:
  - Atom localization
  - Defect identification
  - Image denoising
- Challenge: High-quality training data with ground truth for supervised learning.
- Solution: Use CycleGANs to bridge the gap between simulated and experimental STEM images.



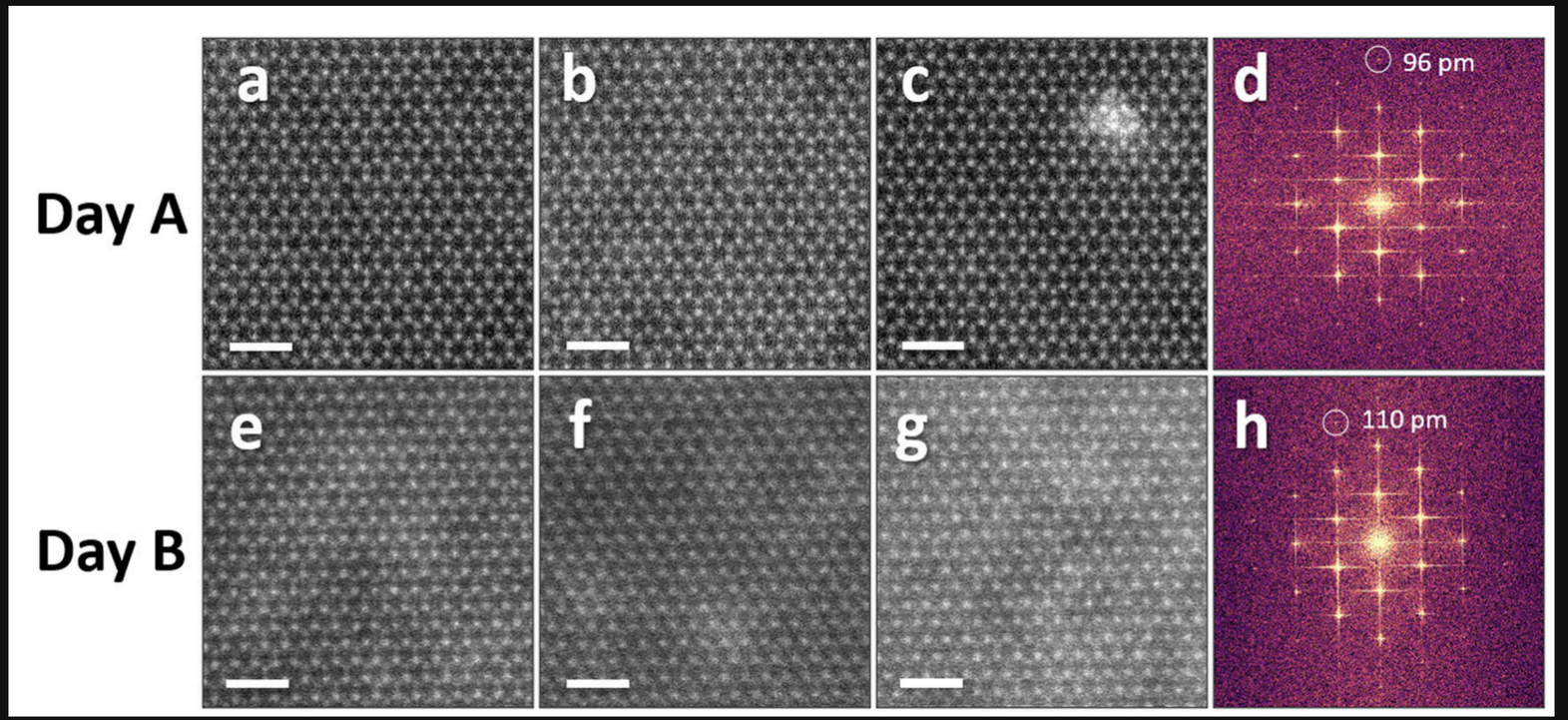
Generating Realistic Images with CycleGANs

- CycleGANs are used for image-to-image translation.
- Training involves both experimental and simulated images.
- CycleGANs produce images that match experimental data in terms of noise, distortions, and other artifacts.
- Results show significant improvement in realism compared to unoptimized simulated images.



variations within experimental STEM data

- Experimental STEM images acquired on Day A (a–c) and B (e–g) showing variations within and between days due to sample contamination and microscope instabilities.
- d, h Power spectra obtained with FFT of a and e demonstrate the variation of image resolution, from 96 pm to 110 pm, determined by the highest transferred spatial frequency marked with white circles.
- The power spectra are displayed in log scale for visibility. Scale bars are equal to 1 nm.



Example of variations within experimental STEM data sets taken on two different days, Day A and Day B.

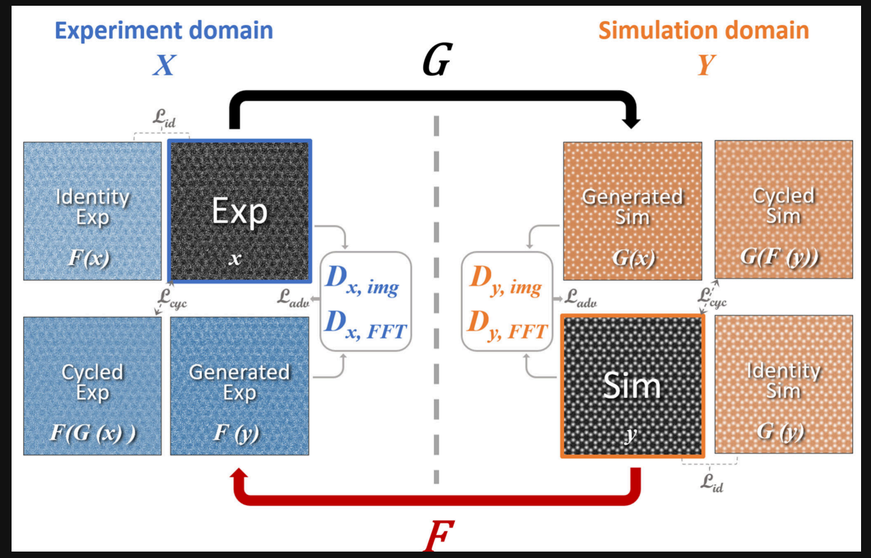
CycleGAN Architecture and Optimization

- CycleGAN translates images between two domains:

  1. the experiment domain X and

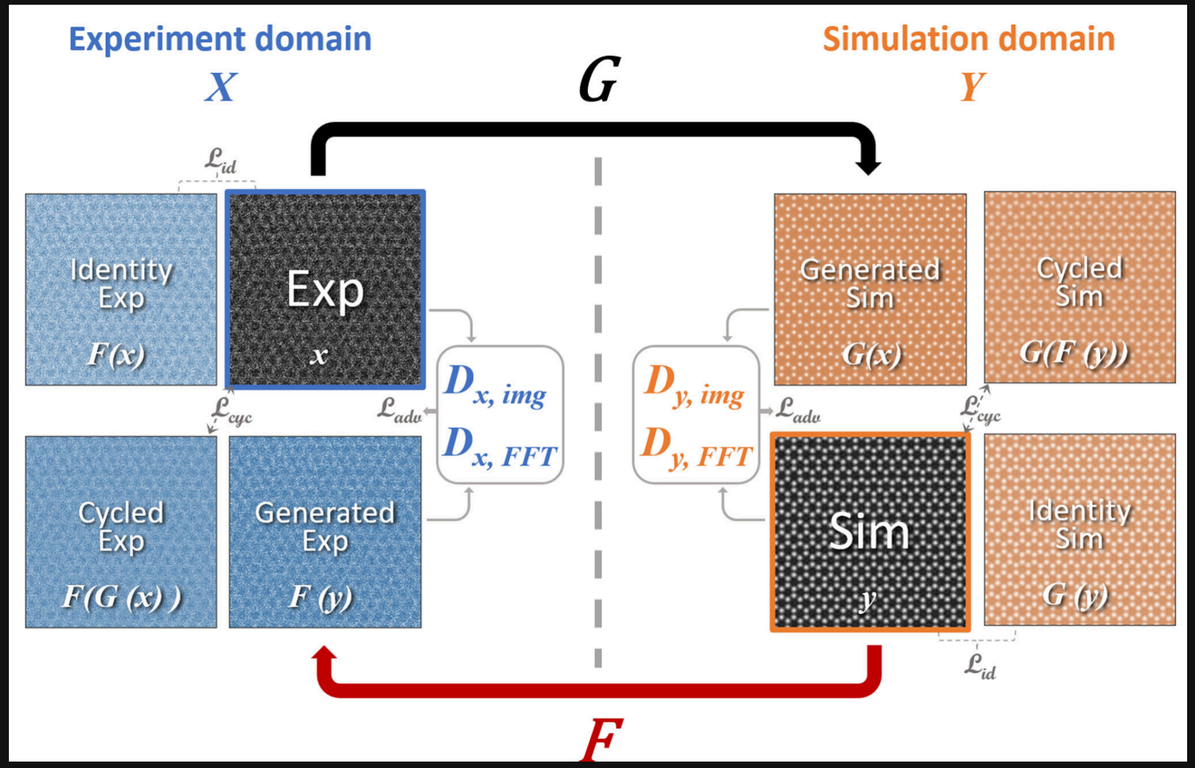
  2. the simulation domain Y, and has two main components: generators and discriminators.

- generator G converts input experimental images x to simulation-like images G(x), and the generator F converts input simulated images y to experiment-like images F(y).
- generated images (F(y) and G(x)) are then fed into four discriminators (Dx,img, Dx,FFT, Dy,img, Dy,FFT) along with the raw images (x and y) to evaluate the quality of generated images.



Schematic of the major components in a CycleGAN.

- Both input images and their FFTs are examined by the discriminators to calculate the adversarial losses (Ladv), which are used to optimize both generators F and G respectively.
- By passing the raw images (x and y) with combinations of both generators, identity images (F(x) and G(y)) and cycled images (F(G(x)) and G(F(y))) are also generated.
- corresponding identity loss Lid and cycle consistency loss Lcyc are added to ensure the identity and cycle consistency mapping of the generators.



Schematic of the major components in a CycleGAN.

Evaluation Metrics

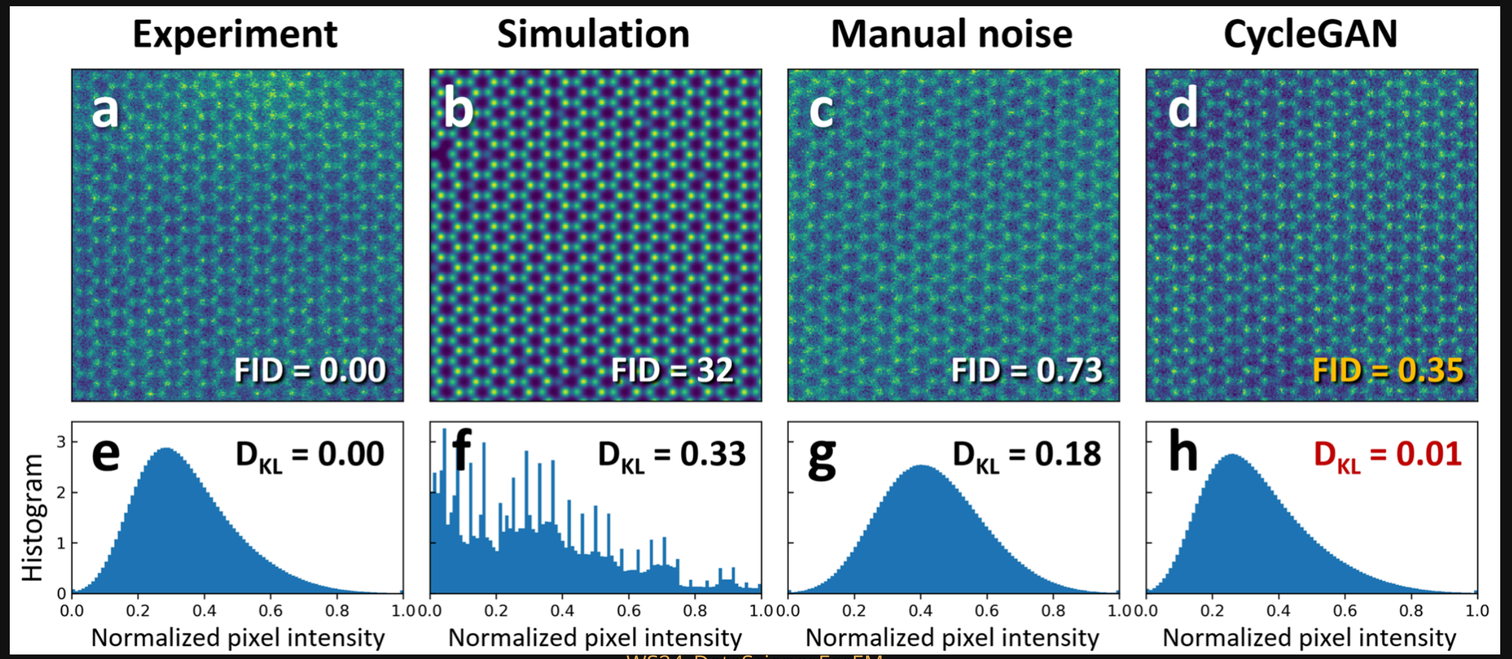
- Quantitative evaluation using:
  - Fréchet Inception Distance (FID)
  - Kullback–Leibler (KL) divergence
- Results:
  - CycleGAN-processed images have the lowest FID scores and KL divergence, indicating high similarity to experimental images.

Fréchet Inception Distance (FID)

- **Purpose**: Measure similarity between generated images and real images.
- **Calculation**:
  - Preprocess the images. Ensure the two images are compatible using basic processing.
  - **Extract feature representations**. Pass the real and generated images through the Inception-v3 model. This transforms the raw pixels into numerical vectors to represent aspects of the images, such as lines, edges and higher-order shapes.
  - **Calculate statistics**. Statistical analysis is performed to determine the mean and covariance matrix of the features in each image.
  - Treats images as multivariate Gaussian distributions
  - $d_F(\mathcal{N}(\mu, \Sigma), \mathcal{N}(\mu', \Sigma'))^2 = \|\mu - \mu'\|_2^2 + \mathrm{tr}\left(\Sigma + \Sigma' - 2(\Sigma\Sigma')^{\frac{1}{2}}\right)$
- **Interpretation**:
  - Lower FID indicates higher similarity and better quality of generated images.

Quantitative measurements of data set quality using FID and KL

- Images are generated from (a) experiments, (b) simulation without noise, (c) simulation with manually optimized noise, and (d) CycleGAN.
- FID score measures the dissimilarity between image data sets, so a smaller FID score implies higher similarity between image data sets.
- e–h Histograms of normalized pixel intensity are calculated for each image data set. Each histogram is normalized so that the probability distribution sums to unity.

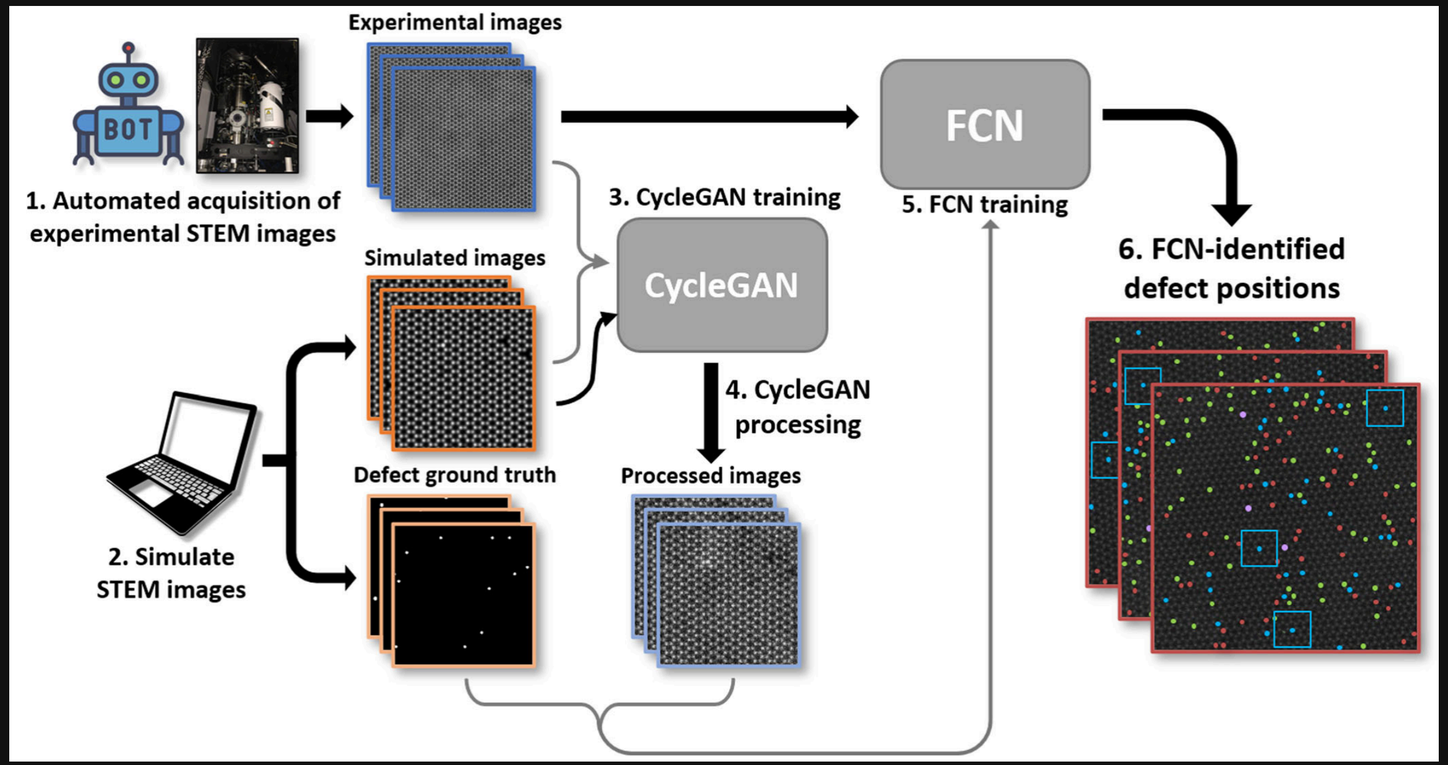


Quantitative measurements of data set quality using FID and KL 2

- KL divergence DKL(P||Q) of each data set with the experimental histogram are labeled as DKL at the top right corner of each histogram, with the lowest non-zero value marked in red.
- both the FID score and KL divergence of intensity histograms are calculated for the entire data set with respect to the experimental data set, where each data set contains roughly 1700 image patches with 256 × 256 pixels.
- CycleGAN generated image set exhibits the best FID score and lowest KL divergence, indicating it is the best match for experimental data.

Defect Identification with CycleGAN and FCN

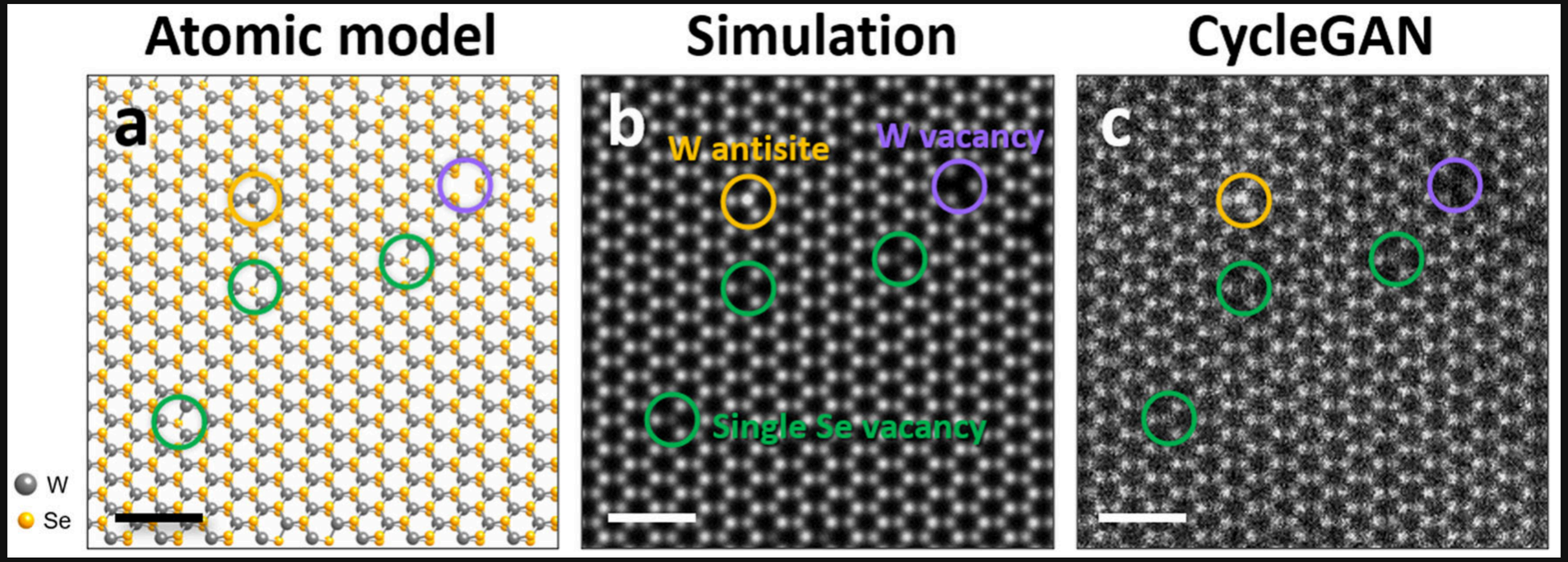
- Workflow:
  1. Acquire experimental STEM images.
  2. Simulate STEM images with known defects.
  3. Train CycleGAN with both experimental and simulated images.



Schematic of the major components in a CycleGAN.

Defect Identification with CycleGAN and FCN

- Workflow:
  4. Generate realistic images using CycleGAN.
  5. Train Fully Convolutional Network (FCN) on CycleGAN-processed images.
  6. Use FCN to identify defects in experimental images.
- Results: High precision and recall in defect identification with minimal human intervention.



The CycleGAN-processed images preserve the defect types and positions from the input simulated images.

Conclusion

- CycleGANs effectively bridge the gap between simulated and experimental STEM images.
- FCNs trained on CycleGAN-processed images achieve high accuracy in defect identification.
- Potential for real-time, automated microscopy data processing.

Example 2: Generation of Highly Realistic Microstructural Images of Alloys from Limited Data with a Style-Based Generative Adversarial Network

Authors: Guillaume Lambard, Kazuhiko Yamazaki, Masahiko Demura
Published in: Scientific Reports (2023)
DOI: 10.1038/s41598-023-27574-8

Introduction

- **Microstructural Characterization**: Essential for understanding material properties.
- **Challenges**: Limited availability of high-quality SEM images.
- **Solution**: Use StyleGAN2 with ADA to generate synthetic SEM images from a small dataset.

StyleGAN2 with ADA

- **StyleGAN2 Architecture**:
  - Adaptive Discriminator Augmentation (ADA)
  - Effective in low data regimes
- **Training**:
  - Dataset: 3000 SEM images of ferrite-martensite dual-phase steel
  - Image size: 512x512 pixels
  - Augmentations: Pixel blitting, geometrical transformations

StyleGAN2 with ADA

- **Problem**: GANs require large datasets to avoid discriminator overfitting.
- **Solution**: Adaptive Discriminator Augmentation (ADA) to stabilize training with limited data.
- **Key Insight**: ADA prevents overfitting without changing loss functions or network architectures.

Adaptive Discriminator Augmentation (ADA)

- **Objective**: Apply augmentations to prevent discriminator overfitting.
- **Mechanism**:
  - Stochastic augmentation of discriminator inputs.
  - Augmentations include geometric and color transformations.
  - Adaptive control based on overfitting heuristics.
- **Benefits**:
  - Effective even with few thousand training images.
  - Maintains image quality and diversity.

Augmentations and Training

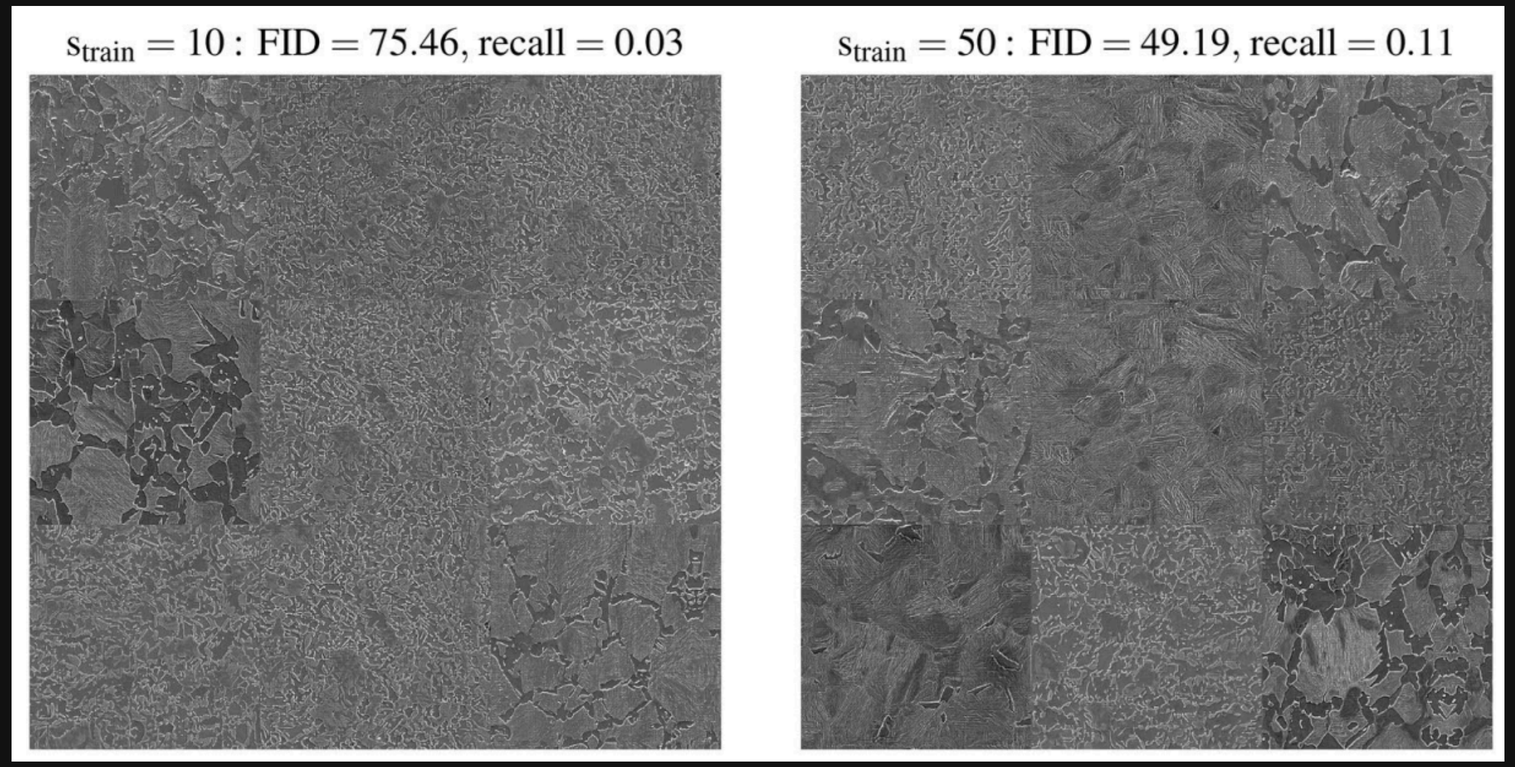
- **Augmentation Strategies**:
  - Pixel blitting and geometrical transformations improved FID by ~77%
  - Other augmentations like color transformations and additive noise were less effective.
- **Target Heuristic (rt)**:
  - Optimal value: 0.5
  - Balances between reducing overfitting and maintaining diversity

Results and Evaluation

- **Evaluation Metrics**:
  - Fréchet Inception Distance (FID)
  - Recall Metric

Results and Evaluation2

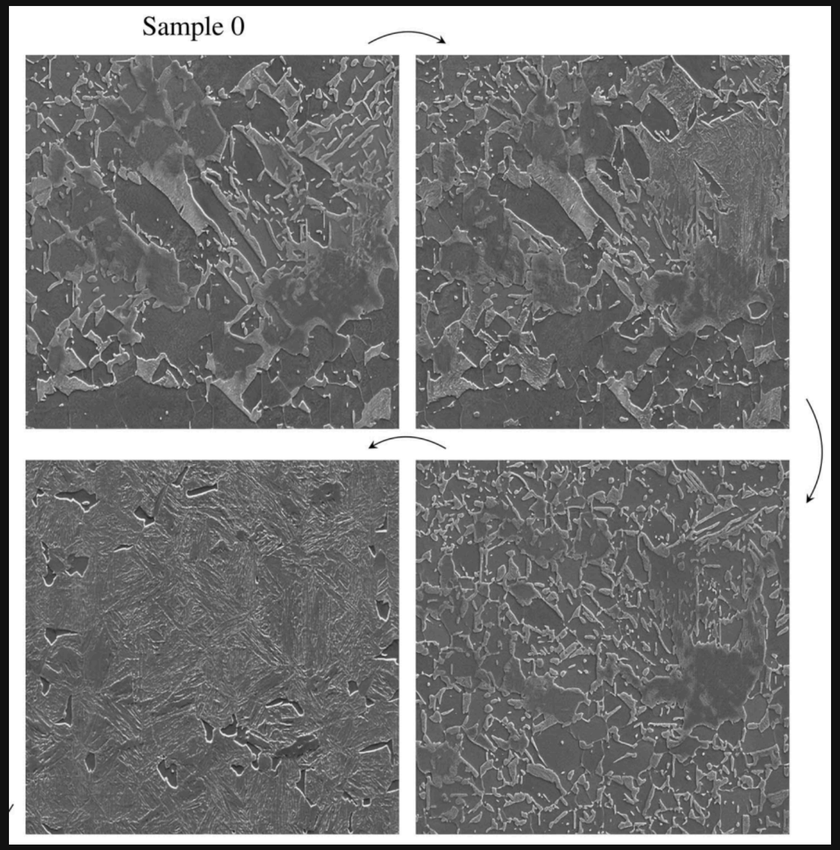
- **Results**:
  - Best FID: 6.59 with 3000 images
  - High-quality and diverse SEM images
  - Successful interpolation between microstructures



Samples of non-curated SEM images generated with the StyleGAN2 with ADA

Interpolation and Diversity

- **Latent Space Interpolation**:
  - Smooth transitions between different microstructures
  - Demonstrates the potential for exploring new microstructural features



Selection of 4 non-curated generated SEM images obtained thanks to a spherical linear interpolation

Interpolation and Diversity 2

- **Generated Images**:
  - High resemblance to real SEM images
  - Captured both coarse and fine microstructural details