Task 3E: Symmetry project for End-to-End Particle Reconstruction

---

● **Dataset Preparation**: Use the vanilla *MNIST* dataset for this purpose. Rotate every sample in steps of 30 degrees and store them in a data format of your choice. Only use the digits 1 and 2 from the dataset if the computational budget is limited.

● **Latent Space Creation**: Build an Auto-Encoder of your choice and train it using the dataset prepared in the previous step.

● **Dataset Distillation**: Devise a strategy to remove the samples that the AE poorly reconstructs to remove outliers.

● **Lie Group Generation**: Using the latent vectors from the AE construct an Infinitesimal operator that represents the rotation group but in the latent space. Refer to the paper for more information on the training scheme.

● **Lie Group Action**: Demonstrate the rotation action of the operator by applying it to an arbitrary latent vector from the dataset and decoding it using the decoder of the AE.

In [1]:
```
import os
import gc
from random import shuffle

import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
```

In [2]:
```
import torch
from torch import nn, optim

from torch.nn import functional as F

from PIL import Image
from torch.utils.data import Dataset, DataLoader, TensorDataset, random_spli

from torchvision.datasets import MNIST
from torchvision import transforms as T
from torchvision import models
```

Phase I: Data Preparation

Aim: Use the vanilla MNIST dataset for this purpose. Rotate every sample in steps of 30 degrees and store them in a data format of your choice. Onl use the digits 1 and 2 from the dataset if the computational budget is limited.

First, I load the *MNIST* dataset.

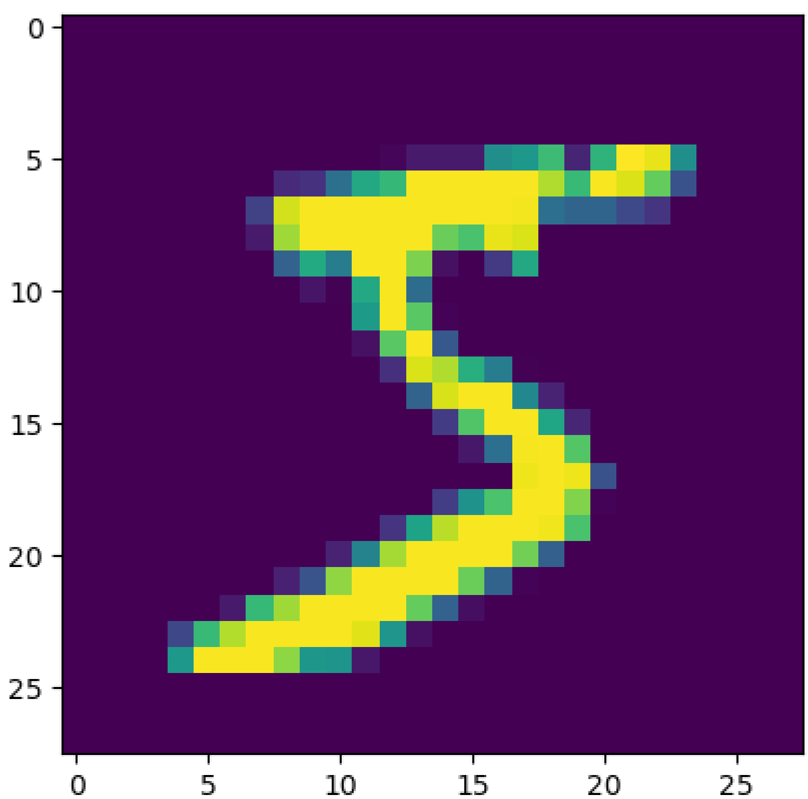
In [77]:
```
# Load data files
train_dataset = MNIST(root = ".", download = True, train = True)
test_dataset = MNIST(root = ".", download = True, train = False)
```

In [4]:
```
# Get image-label pair at index 0
index = 0
train_dataset.__getitem__(index)
```

Out[4]:
```
(<PIL.Image.Image image mode=L size=28x28>, 5)
```

In [6]:
```
plt.imshow(train_dataset[0][0])
```

Out[6]:
```
<matplotlib.image.AxesImage at 0x7f34e3ec1d80>
```



The images in the dataset need to be rotated incrementally by 30 degrees. This will effectively increase the size of the dataset by a factor of 12.

In [7]:
```
360/30
```

Out[7]:
```
12.0
```

In [8]:
```
[30 * n for n in range(1, 12)]
```

Out[8]:
```
[30, 60, 90, 120, 150, 180, 210, 240, 270, 300, 330]
```

It is stated in the document that the rotated images should be stored in a format of choice. I however ellect to go against this specific instruction for the following reasons:

1. **I/O-bound Operations**: Persisting 720,000 images might lead to issues with IO.

In [9]:
```
class MNISTDataset(Dataset):
    def __init__(self, dataset, return_tensor = False):
        super().__init__()

        self.return_tensor = return_tensor

        self.base_rotator = T.Compose(
            [
                T.ToTensor(),
                T.Normalize([0.5,], [0.5,])
            ]
        )

        self.rotators = [
            T.Compose(
                [
                    T.ToTensor(),
                    T.Normalize([0.5,], [0.5,]),
                    T.RandomRotation((30 * factor, 30 * factor)),
                ]
            )

            for factor in range(1, 12)
        ]

        self.dataset = dataset

    def __getitem__(self, index):
        mapped_index = int(index / 12) # Index to base image to be rotated
        rotation_index = index % 12    # Index for angular rotation to be ap

        image, label = self.dataset.__getitem__(mapped_index)
        image = self.rotators[rotation_index - 1](image) if rotation_index >
```

```python
        return (image, label) if self.return_tensor else (T.ToPILImage()(ima


    def __len__(self):
        return self.dataset.__len__() * 12
```

In [78]:
```python
train_dataset = MNISTDataset(dataset = train_dataset)
test_dataset = MNISTDataset(dataset = test_dataset)
```

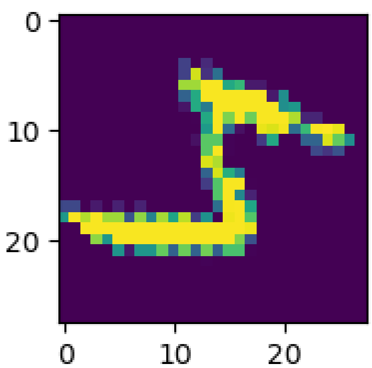
In [79]:
```python
train_dataset[11]
```

Out[79]:
```
(<PIL.Image.Image image mode=L size=28x28>, 5)
```

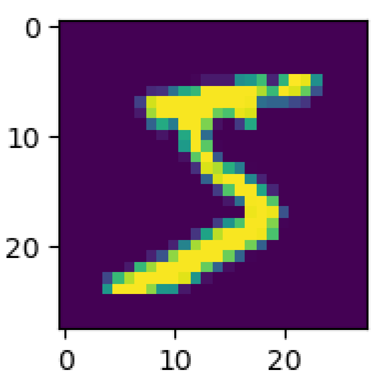
In [80]:
```python
test_dataset[11]
```

Out[80]:
```
(<PIL.Image.Image image mode=L size=28x28>, 7)
```

In [82]:
```python
plt.figure(figsize = (2, 2))
plt.imshow(train_dataset[11][0])
plt.show(); plt.close("all")
```



In [83]:
```python
plt.figure(figsize = (2, 2))
plt.imshow(train_dataset[0][0])
plt.show(); plt.close("all")
```

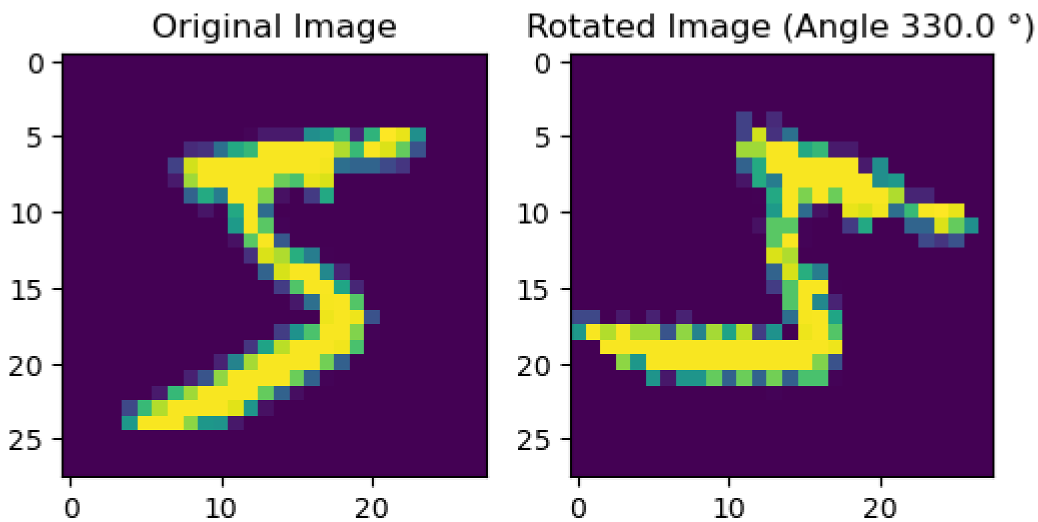


In [15]:
```python
train_dataset.dataset[1]
```

Out[15]:
```
(<PIL.Image.Image image mode=L size=28x28>, 0)
```

In [104…

```
index = 11
rotation_index = index % 12
real_index = int(index / 12)

angle = 0 if rotation_index == 0 else train_dataset.rotators[rotation_index

fig, axes = plt.subplots(nrows = 1, ncols = 2, figsize = (6, 3))

axes[0].imshow(train_dataset.dataset[real_index][0])
# axes[1].imshow(train_dataset[12][0])
axes[1].imshow(train_dataset[index][0])

axes[0].set_title("Original Image")
axes[1].set_title(f"Rotated Image (Angle {angle} °)")

plt.show(); plt.close("all")
```



In [ ]:

```

```

Next, I define dataloaders from the datasets.

In [18]:

```
# Ensure datasets return tensors
train_dataset.return_tensor = True
test_dataset.return_tensor = True
```

In [19]:

```
BATCH_SIZE = 128

train_dl = DataLoader(train_dataset, batch_size = BATCH_SIZE, shuffle = True
test_dl = DataLoader(test_dataset, batch_size = BATCH_SIZE, shuffle = False,
```

In [20]:

```
s = next(iter(train_dl))
```

In [21]:

```
s[0].shape
```

Out[21]: torch.Size([128, 1, 28, 28])

In [22]:
```
s[0].max()
```

Out[22]: tensor(1.)

In [23]:
```
s[0].min()
```

Out[23]: tensor(0.)

In [24]:
```
gc.collect()
```

Out[24]: 5857

In [ ]:

In [ ]:

In [ ]:

In [ ]:

Phase II: Latent Space Creation

Aim: Build an Auto-Encoder of your choice and train it using the dataset prepared in the previous step.

In [25]:
```
class ParticleEncoder(nn.Module):
    def __init__(self, latent_dim, p = .4):
        super(ParticleEncoder, self).__init__()

        self.latent_dim = latent_dim
        self.p = p

        self.l1 = nn.Conv2d(in_channels = 1, out_channels = 8, stride = 1, p
        self.b1 = nn.BatchNorm2d(8)
        self.d1 = nn.Dropout(self.p)

        self.l2 = nn.Conv2d(in_channels = 8, out_channels = 16, stride = 1,
        self.b2 = nn.BatchNorm2d(16)
        self.d2 = nn.Dropout(self.p)

        self.l3 = nn.Conv2d(in_channels = 16, out_channels = 32, stride = 3,
        self.b3 = nn.BatchNorm2d(32)
        self.d3 = nn.Dropout(self.p)

        self.l4 = nn.Conv2d(in_channels = 32, out_channels = 64, stride = 2,
```

```
        self.b4 = nn.BatchNorm2d(64)

        self.linear = nn.Linear(64*6*6, self.latent_dim)

        self.flatten = nn.Flatten()

    def forward(self, x):
        y = self.d1(F.relu(self.b1(self.l1(x))))
        y = self.d2(F.relu(self.b2(self.l2(y))))
        y = self.d3(F.relu(self.b3(self.l3(y))))

        y = F.relu(self.b4(self.l4(y)))

        y = self.flatten(y)
        y = self.linear(y)

        return y
```

In [ ]:

```
```

In [26]:

```
class ParticleDecoder(nn.Module):
    def __init__(self, latent_dim):
        super(ParticleDecoder, self).__init__()

        self.latent_dim = latent_dim

        self.l1 = nn.ConvTranspose2d(in_channels = 64, out_channels = 32, ke
        self.b1 = nn.BatchNorm2d(32)

        self.l2 = nn.ConvTranspose2d(in_channels = 32, out_channels = 16, ke
        self.b2 = nn.BatchNorm2d(16)

        self.l3 = nn.ConvTranspose2d(in_channels = 16, out_channels = 8, ker
        self.b3 = nn.BatchNorm2d(8)

        self.l4 = nn.Conv2d(in_channels = 8, out_channels = 1, kernel_size =

        self.linear = nn.Linear(self.latent_dim, 64*6*6)

    def forward(self, x):
        x = F.relu(self.linear(x))
        y = F.relu(self.b1(self.l1(x.view(-1, 64, 6, 6))))
        y = F.relu(self.b2(self.l2(y)))
        y = F.relu(self.b3(self.l3(y)))

        y = torch.sigmoid(self.l4(y).view(-1, 1, 28, 28))

        return y
```

In [ ]:

```
```

In [27]:

```
class ParticleAutoEncoder(nn.Module):
    def __init__(self, in_dim = 128, out_dim = 128, encoder = None, decoder
        super(ParticleAutoEncoder, self).__init__()
```

```
        self.in_dim, self.out_dim = in_dim, out_dim

        self.encoder = encoder
        self.decoder = decoder

    def forward(self, x):
        x_encoded = self.encoder(x)
        return self.decoder(x_encoded)

    @torch.no_grad()
    def generate_images(self, x_, code_type = 'code'):
        self.eval()
        if code_type == 'code':
            img = self.decoder(x_)
        elif code_type == 'image':
            img = self(x_)
        else:
            raise Exception('Provide a random code [-1, 1] or an image [-1,

        return img

    @torch.no_grad()
    def generate_code(self, image):
        self.eval()
        code = self.encoder(image)
        return code
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [28]:
```
def initialize_weights(model):
    for (name, weights) in filter(lambda x: x[1].requires_grad, model.named_
        if name.split(".")[1] not in ["fc", "conv1"]:
            continue
        try:
            nn.init.kaiming_normal_(weights)
        except:
            nn.init.normal_(weights, 0., 0.05)

    return model

def get_l2_loss(model):
    return sum([x ** 2 for x in model.parameters()])
```

In [29]:
```
def weight_initializer(model):
    for layer in model.children():
        if 'Conv' in str(type(layer)):
            nn.init.normal_(layer.weight, 0, 0.5)
        elif 'Linear' in str(type(layer)):
            nn.init.xavier_normal_(layer.weight, 1.0)

    return model
```

In [30]:
```
latent_dim = 32
```

In [31]:
```
encoder = ParticleEncoder(latent_dim = latent_dim)
decoder = ParticleDecoder(latent_dim = latent_dim)
```

In [32]:
```
model = ParticleAutoEncoder(encoder = encoder.apply(weight_initializer), dec
```

In [33]:
```
model
```

Out[33]:
```
ParticleAutoEncoder(
  (encoder): ParticleEncoder(
    (l1): Conv2d(1, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (b1): BatchNorm2d(8, eps=1e-05, momentum=0.1, affine=True, track_runnin
g_stats=True)
    (d1): Dropout(p=0.4, inplace=False)
    (l2): Conv2d(8, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (b2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
    (d2): Dropout(p=0.4, inplace=False)
    (l3): Conv2d(16, 32, kernel_size=(3, 3), stride=(3, 3), padding=(1, 1))
    (b3): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
    (d3): Dropout(p=0.4, inplace=False)
    (l4): Conv2d(32, 64, kernel_size=(2, 2), stride=(2, 2), padding=(1, 1))
    (b4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
    (linear): Linear(in_features=2304, out_features=32, bias=True)
    (flatten): Flatten(start_dim=1, end_dim=-1)
  )
  (decoder): ParticleDecoder(
    (l1): ConvTranspose2d(64, 32, kernel_size=(3, 3), stride=(3, 3), paddin
g=(1, 1))
    (b1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
    (l2): ConvTranspose2d(32, 16, kernel_size=(2, 2), stride=(2, 2), paddin
g=(2, 2))
    (b2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
    (l3): ConvTranspose2d(16, 8, kernel_size=(2, 2), stride=(2, 2), padding
=(2, 2))
    (b3): BatchNorm2d(8, eps=1e-05, momentum=0.1, affine=True, track_runnin
g_stats=True)
    (l4): Conv2d(8, 1, kernel_size=(2, 2), stride=(2, 2), padding=(2, 2))
    (linear): Linear(in_features=32, out_features=2304, bias=True)
  )
)
```

In [34]:
```
sample_input = torch.randn(8, 1, 28, 28)
```

In [35]:
```
out = model(s[0])
```

In [36]:
```
out.shape
```

Out[36]:
```
torch.Size([128, 1, 28, 28])
```

In [ ]:
```

```

In [37]:
```
for name, p in model.named_parameters():
    print(name)
```

```
encoder.l1.weight
encoder.l1.bias
encoder.b1.weight
encoder.b1.bias
encoder.l2.weight
encoder.l2.bias
encoder.b2.weight
encoder.b2.bias
encoder.l3.weight
encoder.l3.bias
encoder.b3.weight
encoder.b3.bias
encoder.l4.weight
encoder.l4.bias
encoder.b4.weight
encoder.b4.bias
encoder.linear.weight
encoder.linear.bias
decoder.l1.weight
decoder.l1.bias
decoder.b1.weight
decoder.b1.bias
decoder.l2.weight
decoder.l2.bias
decoder.b2.weight
decoder.b2.bias
decoder.l3.weight
decoder.l3.bias
decoder.b3.weight
decoder.b3.bias
decoder.l4.weight
decoder.l4.bias
decoder.linear.weight
decoder.linear.bias
```

Now, I define a function to initialize model weights.

In [38]:
```
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

In [39]:
```
DEVICE
```

Out[39]:
```
device(type='cuda')
```

In [40]:
```
model = model.to(DEVICE)

# model = initialize_weights(model)
```

In [41]:
```
EPOCHS = 20
l2_lambda = 1e-5

criterion = nn.MSELoss().to(DEVICE)
```

```
# Optimizer hyperparameters
LR = 1e-2
FACTOR = 10
AMSGRAD = False
BETAS = (.9, .999)
```

optimizer = optim.AdamW([{'params': encoder.parameters()}, {'params': decoder.parameters()}], lr = .001, weight_decay = 1e-5) scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, patience = 2, min_lr = 0.0001)

In this notebook, the pretrained weights will be finetuned. This is in contrast to the previous one, where the weights were kept frozen. Also, the learing rate is increased from 1e-4 to 1e-3.

In [42]:
```
opt = optim.AdamW(
    params = [
        {
            "params" : model.encoder.parameters(),
            "lr": LR
        },
        {
            "params" : model.decoder.parameters(),
            "lr": LR
        }
    ],
    lr=LR/FACTOR,
    amsgrad = AMSGRAD,
    betas = BETAS,
    weight_decay = l2_lambda
)
scheduler = optim.lr_scheduler.ReduceLROnPlateau(opt, patience = 2, min_lr =
```

In [43]:
```
def get_l2_loss(model):
    l2_loss = torch.tensor(0.).cuda()
    l2_loss += sum(map(lambda x: x.data.pow(2).sum(), filter(lambda x: x.rec
    return l2_loss
```

In [44]:
```
from sklearn.metrics import accuracy_score
```

In [45]:
```
### Code mean (determined via pretraining)
MEAN = .02
STD = .233
```

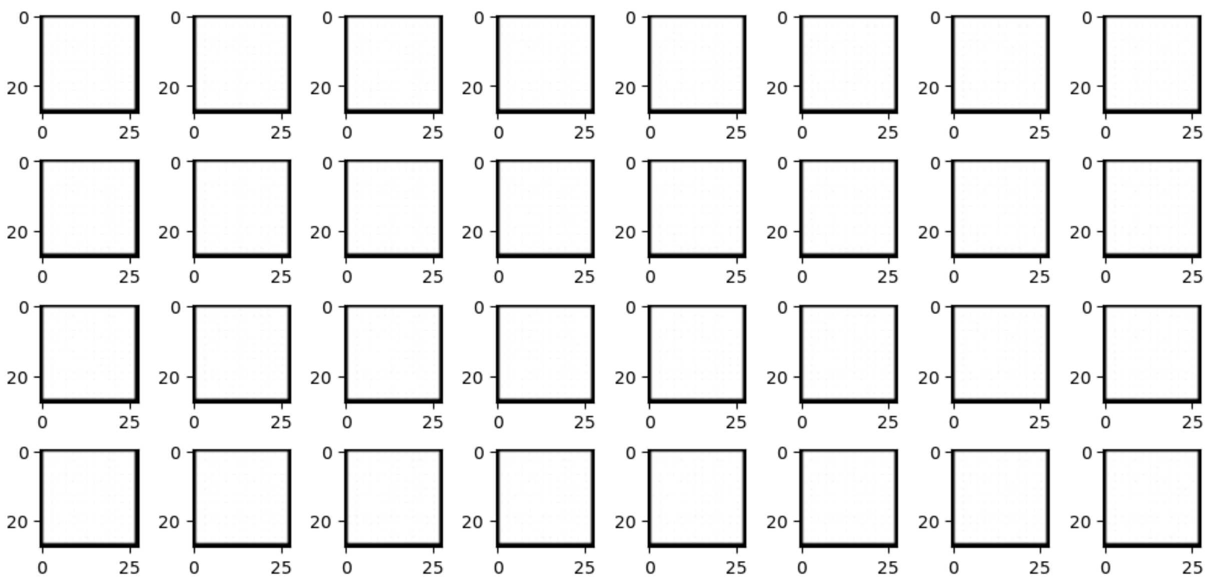
In [46]:
```
def training_loop(epochs, model, optimizer):
    TRAIN_LOSSES, TEST_LOSSES = [], []

    for epoch in range(1, epochs + 1):
        train_losses, test_losses = [], []

        model.train() # Set up training mode

        for batch in iter(train_dl):
```

```
        # X, y = collate_function(batch)
        X, y = batch
        X, y = X.to(DEVICE), y.view(-1).to(DEVICE)

        optimizer.zero_grad()

        y_pred = model(X)

        # Uncomment the line below if the criterion is nn.NLLLoss()
        # y_pred = torch.log_softmax(y_pred, dim = -1)

        # Compare actual targets and predicted targets to get the loss
        train_loss = criterion(y_pred, X) #+ (l2_lambda * get_l2_loss(mo
        # Backpropagate the loss
        train_loss.backward()

        optimizer.step()

        train_losses.append(train_loss.detach().item())

    scheduler.step(train_loss)

    with torch.no_grad(): # Turn off computational graph
        model.eval() # Set model to evaluation mode
        for batch in iter(test_dl):
            # X_, y_ = collate_function(batch)
            X_, y_ = batch
            X_, y_ = X_.to(DEVICE), y_.view(-1).to(DEVICE)

            y_pred_ = model(X_)

            # Uncomment the line below if the criterion is nn.NLLLoss()
            # y_pred_ = torch.log_softmax(y_pred_, dim = -1)

            # Compare actual targets and predicted targets to get the lo
            test_loss = criterion(y_pred_, X_) #+ (l2_lambda * get_l2_lo
            test_losses.append(test_loss.item())


    avg_train_loss = sum(train_losses) / len(train_losses)
    avg_test_loss = sum(test_losses) / len(test_losses)

    print(
        f"Epoch: {epoch} | Train MSE loss: {avg_train_loss: .3f} | Test
    )

    TRAIN_LOSSES.append(avg_train_loss)
    TEST_LOSSES.append(avg_test_loss)

    img = model.generate_images(
        x_ = torch.distributions.Normal(loc = MEAN, scale = STD).sample(
        code_type = 'code'
    )

    fig, ax = plt.subplots(nrows = 4, ncols = 8, figsize = (10, 5))
```

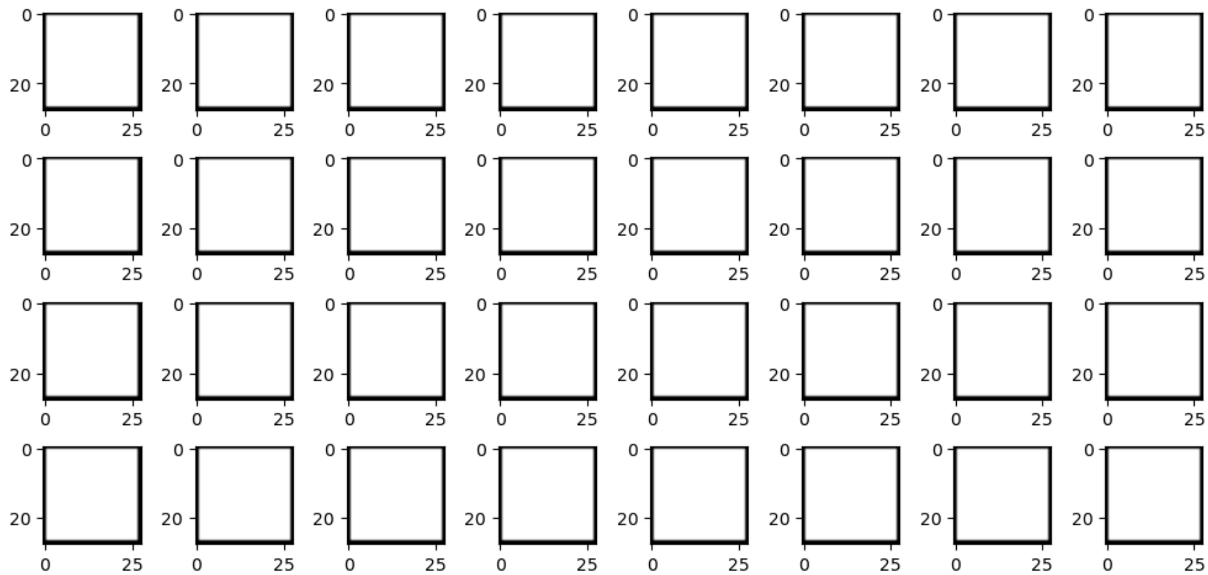
```python
        ix = 0
        for i in range(4):
            for j in range(8):
                ax[i, j].imshow(img[ix].cpu().squeeze(), 'gray')
                ix += 1

        plt.tight_layout(h_pad = 0.01)
        plt.show(); plt.close('all')

        print("\n\n\n" + "="*145)


    # Clear CUDA cache
    torch.cuda.empty_cache()
    torch.clear_autocast_cache()

    return {
        "loss": [TRAIN_LOSSES, TEST_LOSSES],
        "model": model
    }
```

In [47]:
```python
# Train Resnet-18 with finetuning
model_results = training_loop(epochs = EPOCHS, optimizer = opt, model = mode
```

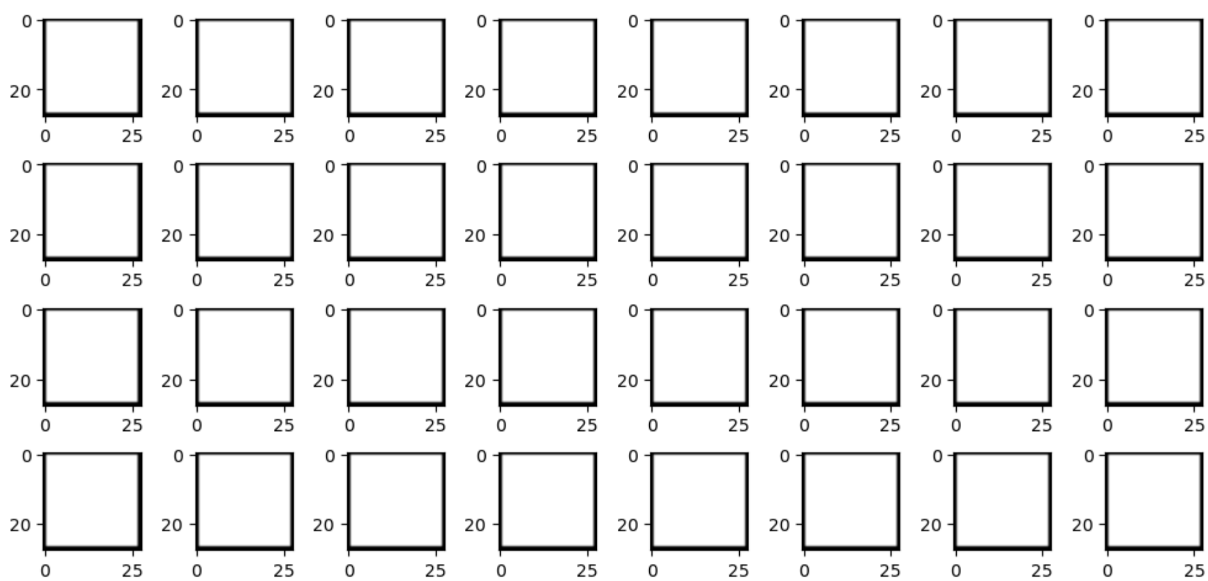
```
/home/harkhymadhe/miniforge3/envs/quantum/lib/python3.10/site-packages/torc
h/nn/modules/conv.py:456: UserWarning: Applied workaround for CuDNN issue, i
nstall nvrtc.so (Triggered internally at /opt/conda/conda-bld/pytorch_170802
5831440/work/aten/src/ATen/native/cudnn/Conv_v8.cpp:80.)
  return F.conv2d(input, weight, bias, self.stride,
Epoch: 1 | Train MSE loss:  0.013 | Test MSE loss:  0.009 |
```



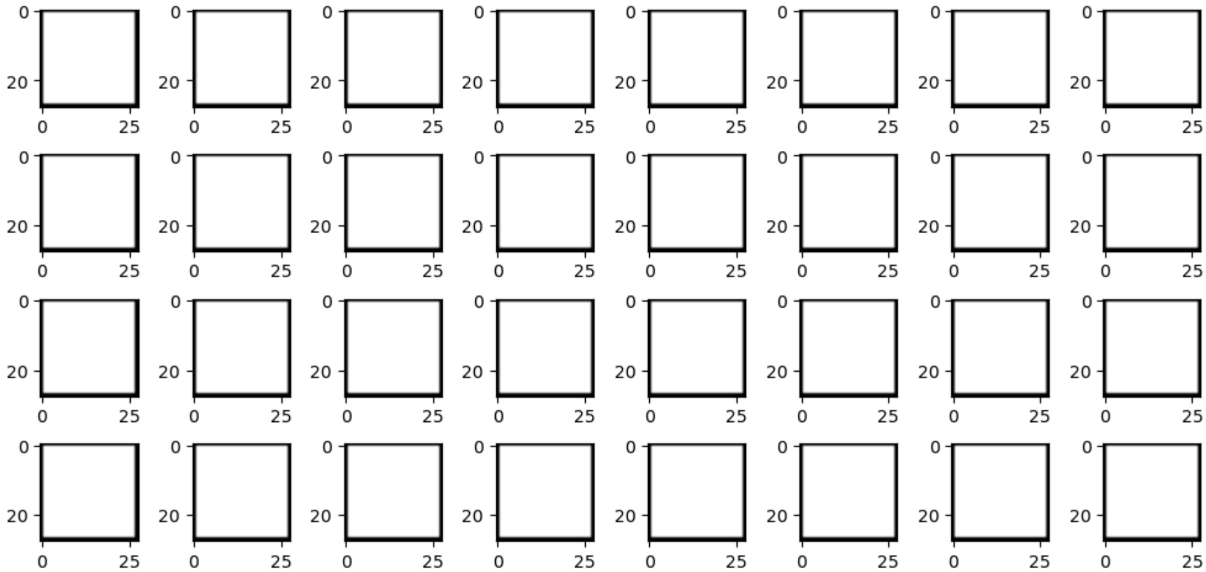
```
============================================================================
=====================================================================
Epoch: 2 | Train MSE loss:  0.008 | Test MSE loss:  0.007 |
```



```
=============================================================================
======================================================================
Epoch: 3 | Train MSE loss:  0.007 | Test MSE loss:  0.007 |
```



```
=============================================================================
======================================================================
Epoch: 4 | Train MSE loss:  0.007 | Test MSE loss:  0.007 |
```


0
20
0
25

==
==

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
/tmp/ipykernel_25579/2859640907.py in <cell line: 2>()
      1 # Train Resnet-18 with finetuning
----> 2 model_results = training_loop(epochs = EPOCHS, optimizer = opt, mode
l = model)

/tmp/ipykernel_25579/2906855426.py in training_loop(epochs, model, optimize
r)
      7         model.train() # Set up training mode
      8 
----> 9         for batch in iter(train_dl):
     10             # X, y = collate_function(batch)
     11             X, y = batch

~/miniforge3/envs/quantum/lib/python3.10/site-packages/torch/utils/data/data
loader.py in __next__(self)
    629                 # TODO(https://github.com/pytorch/pytorch/issues/767
50)
    630                 self._reset()  # type: ignore[call-arg]
--> 631             data = self._next_data()
    632             self._num_yielded += 1
    633             if self._dataset_kind == _DatasetKind.Iterable and \

~/miniforge3/envs/quantum/lib/python3.10/site-packages/torch/utils/data/data
loader.py in _next_data(self)
    673     def _next_data(self):
    674         index = self._next_index()  # may raise StopIteration
--> 675         data = self._dataset_fetcher.fetch(index)  # may raise StopI
teration
    676         if self._pin_memory:
    677             data = _utils.pin_memory.pin_memory(data, self._pin_memo
ry_device)

~/miniforge3/envs/quantum/lib/python3.10/site-packages/torch/utils/data/_uti
ls/fetch.py in fetch(self, possibly_batched_index)
     49                 data = self.dataset.__getitems__(possibly_batched_in
dex)
     50             else:
---> 51                 data = [self.dataset[idx] for idx in possibly_batche
d_index]
     52         else:
     53             data = self.dataset[possibly_batched_index]

~/miniforge3/envs/quantum/lib/python3.10/site-packages/torch/utils/data/_uti
ls/fetch.py in <listcomp>(.0)
     49                 data = self.dataset.__getitems__(possibly_batched_in
dex)
     50             else:
---> 51                 data = [self.dataset[idx] for idx in possibly_batche
d_index]
     52         else:
     53             data = self.dataset[possibly_batched_index]

/tmp/ipykernel_25579/3041457710.py in __getitem__(self, index)
     23 
```

```
     24         image, label = self.dataset.__getitem__(mapped_index)
---> 25         image = self.rotators[rotation_index - 1](image) if rotation
_index > 0 else T.ToTensor()(image)
     26 
     27         return (image, label) if self.return_tensor else (T.ToPILIma
ge()(image), label)

~/miniforge3/envs/quantum/lib/python3.10/site-packages/torchvision/transform
s/transforms.py in __call__(self, img)
     93     def __call__(self, img):
     94         for t in self.transforms:
---> 95             img = t(img)
     96         return img
     97 

~/miniforge3/envs/quantum/lib/python3.10/site-packages/torch/nn/modules/modu
le.py in _wrapped_call_impl(self, *args, **kwargs)
   1509             return self._compiled_call_impl(*args, **kwargs)  # typ
e: ignore[misc]
   1510         else:
-> 1511             return self._call_impl(*args, **kwargs)
   1512 
   1513     def _call_impl(self, *args, **kwargs):

~/miniforge3/envs/quantum/lib/python3.10/site-packages/torch/nn/modules/modu
le.py in _call_impl(self, *args, **kwargs)
   1518                 or _global_backward_pre_hooks or _global_backward_ho
oks
   1519                 or _global_forward_hooks or _global_forward_pre_hook
s):
-> 1520             return forward_call(*args, **kwargs)
   1521 
   1522         try:

~/miniforge3/envs/quantum/lib/python3.10/site-packages/torchvision/transform
s/transforms.py in forward(self, img)
   1368             else:
   1369                 fill = [float(f) for f in fill]
-> 1370         angle = self.get_params(self.degrees)
   1371 
   1372         return F.rotate(img, angle, self.interpolation, self.expand,
self.center, fill)

~/miniforge3/envs/quantum/lib/python3.10/site-packages/torchvision/transform
s/transforms.py in get_params(degrees)
   1350             float: angle parameter to be passed to ``rotate`` for ra
ndom rotation.
   1351         """
-> 1352         angle = float(torch.empty(1).uniform_(float(degrees[0]), flo
at(degrees[1])).item())
   1353         return angle
   1354 

KeyboardInterrupt:
```

In [105…
```python
torch.save(model_results["model"].state_dict(), "autoencoder.pt")
```

In [ ]:
```python
def visualize_results(history, key = None):
    if key is not None:
        TRAIN_RESULTS, TEST_RESULTS = history[key]

        plt.figure(figsize = (10, 3))

        plt.plot(range(EPOCHS), TRAIN_RESULTS, label = f"Training {key.capit
        plt.plot(range(EPOCHS), TEST_RESULTS, label = f"Test {key.capitalize

        plt.xlabel("Epochs")
        plt.ylabel(key.capitalize())

        plt.title(key.capitalize() + " Evolution for Train and Test Splits",

        plt.legend()
        plt.show(); plt.close("all")
    else:
        TRAIN_LOSSES, TEST_LOSSES = history["loss"]
        TRAIN_ACCS, TEST_ACCS = history["accuracy"]

        fig, ax = plt.subplots(1, 2, figsize = (15, 4))

        ax[0].plot(range(EPOCHS), TRAIN_LOSSES, label = "Training Loss")
        ax[0].plot(range(EPOCHS), TEST_LOSSES, label = "Test Loss")

        ax[0].set_xlabel("Epochs")
        ax[0].set_ylabel("Loss")

        ax[0].set_title("Loss Evolution for Train and Test Splits", fontsize

        ax[1].plot(range(EPOCHS), TRAIN_ACCS, label = "Training Accuracy")
        ax[1].plot(range(EPOCHS), TEST_ACCS, label = "Test Accuracy")

        ax[1].set_xlabel("Epochs")
        ax[1].set_ylabel("Accuracy")

        ax[1].set_title("Accuracy Evolution for Train and Test Splits", font

        plt.legend()
        plt.show(); plt.close("all")

    return
```

In [ ]:

In [ ]:
```python
# VGG-13 with finetuning
visualize_results(model_results)
```

In [ ]:

In [ ]:

In [56]:
```
s_ = model.generate_images(s[0].to(DEVICE), code_type = "image")
```

In [57]:
```
criterion(s_, s[0].to(DEVICE))
```

Out[57]:
```
tensor(0.0069, device='cuda:0')
```

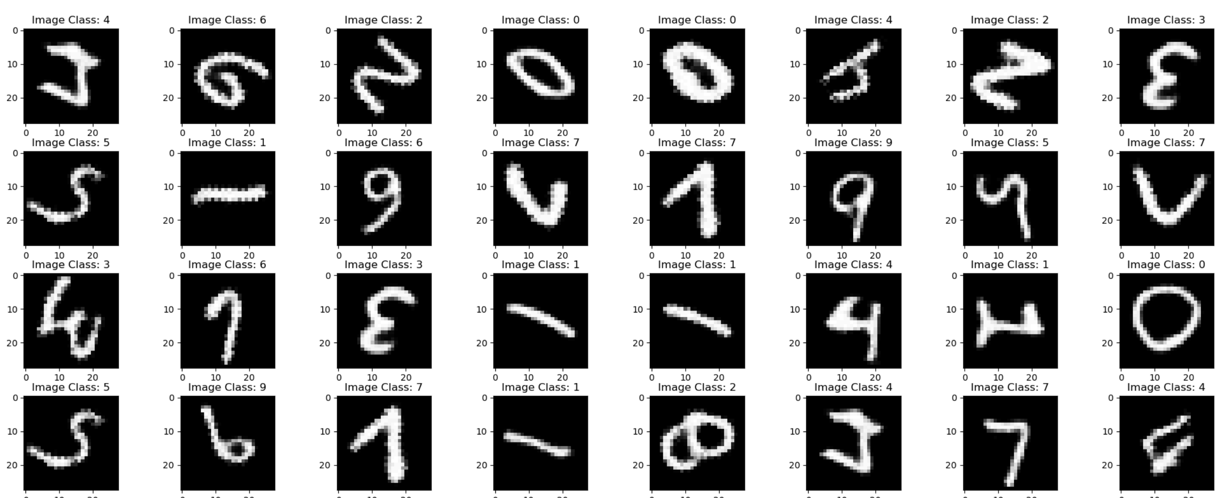
In [72]:
```
s_.max()
```

Out[72]:
```
tensor(0.9995, device='cuda:0')
```

In [73]:
```
s_.min()
```

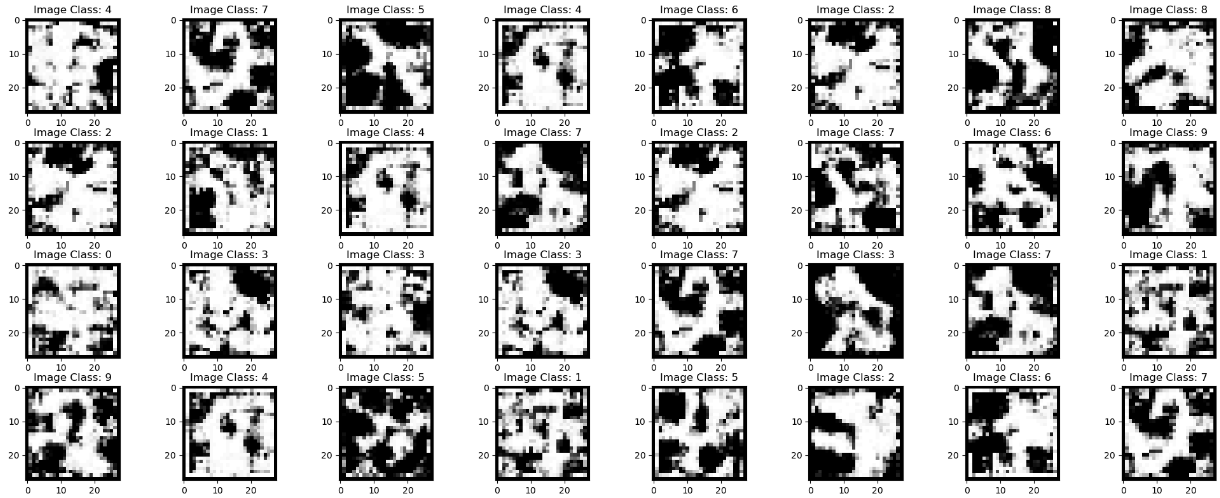
Out[73]:
```
tensor(0., device='cuda:0')
```

In [63]:
```
fig, ax = plt.subplots(nrows = 4, ncols = 8, figsize = (20, 8))

ix = 0
for i in range(4):
    for j in range(8):
        ix = np.random.randint(low = 0, high = len(s_), size = (1,)).item()
        ax[i, j].imshow(s_[ix].cpu().squeeze(), 'gray')
        ax[i, j].set_title(f"Image Class: {s[1][ix].item()}")
        ix += 1

plt.tight_layout(h_pad = 0.01)
plt.show(); plt.close('all')
```



In [71]:
```
s__ = model.generate_images(
    x_ = torch.distributions.Normal(loc = -0.9236, scale = 135.1622).sample(
)

fig, ax = plt.subplots(nrows = 4, ncols = 8, figsize = (20, 8))

ix = 0
for i in range(4):
    for j in range(8):
        ix = np.random.randint(low = 0, high = len(s__), size = (1,)).item()
        ax[i, j].imshow(s__[ix].cpu().squeeze(), 'gray')
        ax[i, j].set_title(f"Image Class: {s[1][ix].item()}")
```

```
        ix += 1

plt.tight_layout(h_pad = 0.01)
plt.show(); plt.close('all')
```



In [106…

```
model.eval()
```

Out[106…
```
ParticleAutoEncoder(
  (encoder): ParticleEncoder(
    (l1): Conv2d(1, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (b1): BatchNorm2d(8, eps=1e-05, momentum=0.1, affine=True, track_runnin
g_stats=True)
    (d1): Dropout(p=0.4, inplace=False)
    (l2): Conv2d(8, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (b2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
    (d2): Dropout(p=0.4, inplace=False)
    (l3): Conv2d(16, 32, kernel_size=(3, 3), stride=(3, 3), padding=(1, 1))
    (b3): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
    (d3): Dropout(p=0.4, inplace=False)
    (l4): Conv2d(32, 64, kernel_size=(2, 2), stride=(2, 2), padding=(1, 1))
    (b4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
    (linear): Linear(in_features=2304, out_features=32, bias=True)
    (flatten): Flatten(start_dim=1, end_dim=-1)
  )
  (decoder): ParticleDecoder(
    (l1): ConvTranspose2d(64, 32, kernel_size=(3, 3), stride=(3, 3), paddin
g=(1, 1))
    (b1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
    (l2): ConvTranspose2d(32, 16, kernel_size=(2, 2), stride=(2, 2), paddin
g=(2, 2))
    (b2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
    (l3): ConvTranspose2d(16, 8, kernel_size=(2, 2), stride=(2, 2), padding
=(2, 2))
    (b3): BatchNorm2d(8, eps=1e-05, momentum=0.1, affine=True, track_runnin
g_stats=True)
    (l4): Conv2d(8, 1, kernel_size=(2, 2), stride=(2, 2), padding=(2, 2))
    (linear): Linear(in_features=32, out_features=2304, bias=True)
  )
)
```

Phase III: Data Distillation

Aim: Devise a strategy to remove the samples that the AE poorly reconstructs to remove outliers.

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

Phase IV: Lie Group Generation

Aim: Using the latent vectors from the AE construct an Infinitesimal operator that represents the rotation group but in the latent space.

The architecture and training scheme used for this is shown below:

No description has been provided for this image

In [ ]:

```
class LieGenerator(nn.Module):
    def __init__(self, latent_dim):
        super().__init__()

        self.latent_dim = latent_dim

        self.linear1 = nn.Linear(in_features = self.latent_dim, out_features
        self.bn1 = nn.BatchNorm1d(self.latent_dim * 2)

        self.linear2 = nn.Linear(in_features = self.latent_dim * 2, out_feat
        self.bn2 = nn.BatchNorm1d(self.latent_dim * 2)

        self.linear3 = nn.Linear(in_features = self.latent_dim * 2, out_feat

    def forward(self, x):
        x = self.bn1(self.linear1(x))
        x = F.relu(x)

        x = self.bn2(self.linear2(x))
        x = F.relu(x)

        x = self.linear3(x)

        return F.tanh(x)
```

In [ ]:

```
class LieDiscriminator(nn.Module):
    def __init__(self, latent_dim):
        super().__init__()

        self.latent_dim = latent_dim
```

```
        self.linear1 = nn.Linear(in_features = self.latent_dim, out_features
        self.bn1 = nn.BatchNorm1d(self.latent_dim * 2)

        self.linear2 = nn.Linear(in_features = self.latent_dim * 2, out_feat
        self.bn2 = nn.BatchNorm1d(self.latent_dim * 2)

        self.linear3 = nn.Linear(in_features = self.latent_dim * 2, out_feat

    def forward(self, x):
        x = self.bn1(self.linear1(x))
        x = F.relu(x)

        x = self.bn2(self.linear2(x))
        x = F.relu(x)

        x = self.linear3(x)

        return F.softmax(x)
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

Phase V: Lie Group Action

Aim: Demonstrate the rotation action of the operator by applying it to an arbitrary latent vector from the dataset and decoding it using the decoder of the AE.

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]: