Exercise 3 from GNN

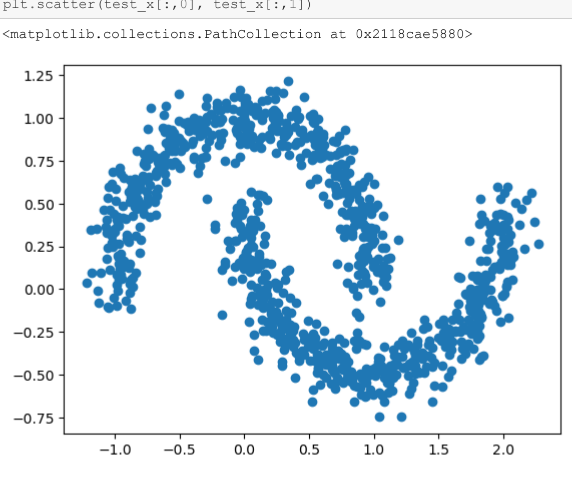
In [1]:
```
import torch
import torch.nn as nn
import math
from sklearn.datasets import make_moons, load_digits
from scipy.stats import special_ortho_group
import matplotlib.pyplot as plt
import random
```

1 Two Moons and Gaussian Mixture Model with INN

In this task we build a RealNVP invertible architecture to try and learn the two moons distribution. We inspect what influence model parameters and learning parameters have on the samples the networks produce. We check if the latent vectors which the network assigns to true data is gaussian distributed. We also evaluate our synthetic data via MMD calculation. Once this is done, we also train our model on a Gaussian mixture distribution and compare the performance to the two moons case.

In [2]:
```
test_x, _ = make_moons(n_samples=1000, shuffle=True, noise=0.1, random_state=42)
plt.scatter(test_x[:,0], test_x[:,1])
```

Out[2]:
```
<matplotlib.collections.PathCollection at 0x2118cae5880>
```



In [3]:
```
def subnet_constructor(input_size, hidden_size, output_size):
    model = nn.Sequential(
                        nn.Linear(input_size, hidden_size),
                        nn.ReLU(),
                        nn.Linear(hidden_size, hidden_size),
                        nn.ReLU(),
                        nn.Linear(hidden_size, output_size)
                        )
    return model

def ortogonal_matrix(dim):
    """A = torch.normal(mean=torch.zeros((dim,dim)), std=torch.ones((dim,dim)))
    Q, _ = torch.linalg.qr(A)
    if dim == 2:
        Q[1,1] = -Q[0,0]
        Q[0,1] = -Q[1,0]
    print(torch.linalg.det(Q))
    return Q"""
    Q = special_ortho_group.rvs(dim)
    return torch.Tensor(Q)

class coupling_block(nn.Module):
    def __init__(self, input_size, hidden_size):
        super().__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size

        self.split1 = math.floor(self.input_size/2)
        self.split2 = self.input_size - self.split1

        self.subnet = subnet_constructor(self.split1, self.hidden_size, 2*self.split2)

    def forward(self, x, rev=False):
        x1, x2 = x[..., :self.split1], x[..., self.split1:]

        params = self.subnet(x1)
        s, t = params[...,:self.split2], params[...,self.split2:]
        s = torch.tanh(s)
        ljd = torch.sum(s, -1)


        if not rev:
            s = torch.exp(s)
            x2 = s*x2 + t
            return torch.cat([x1,x2], -1), ljd
        if rev:
            s = torch.exp(-s)
            x2 = s * (x2-t)
            return torch.cat([x1,x2], -1)

class realNVP(nn.Module):
    def __init__(self, input_size, hidden_size, n_blocks):
        super().__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.n_blocks = n_blocks

        self.coupling_blocks = nn.ModuleList([coupling_block(input_size, hidden_size) for _ in range(n_
blocks)])
        self.orthogonal_matrices = [ortogonal_matrix(input_size) for _ in range(n_blocks-1)]

    def forward(self, x, rev=False):
        if rev: return self._inverse(x)
        return self._forward(x)

    def _forward(self, x):
        ljd = torch.zeros((x.shape[0]))
        for l in range(self.n_blocks-1):
            x, partial_ljd = self.coupling_blocks[l](x)
            ljd += partial_ljd
            x = torch.matmul(x, self.orthogonal_matrices[l])
        x, partial_ljd = self.coupling_blocks[-1](x)
        ljd += partial_ljd
        return x, ljd

    def _inverse(self, x):
        for l in range (self.n_blocks-1, 0, -1):
            x = self.coupling_blocks[l](x, rev=True)
            x = torch.matmul(x, self.orthogonal_matrices[l-1].T)
        x = self.coupling_blocks[0](x, rev=True)
        return x

    def sample(self, num_samples):
        z = torch.normal(mean=torch.zeros((num_samples, self.input_size)), std=torch.ones((num_samples,
 self.input_size)))
        return self._inverse(z)
```

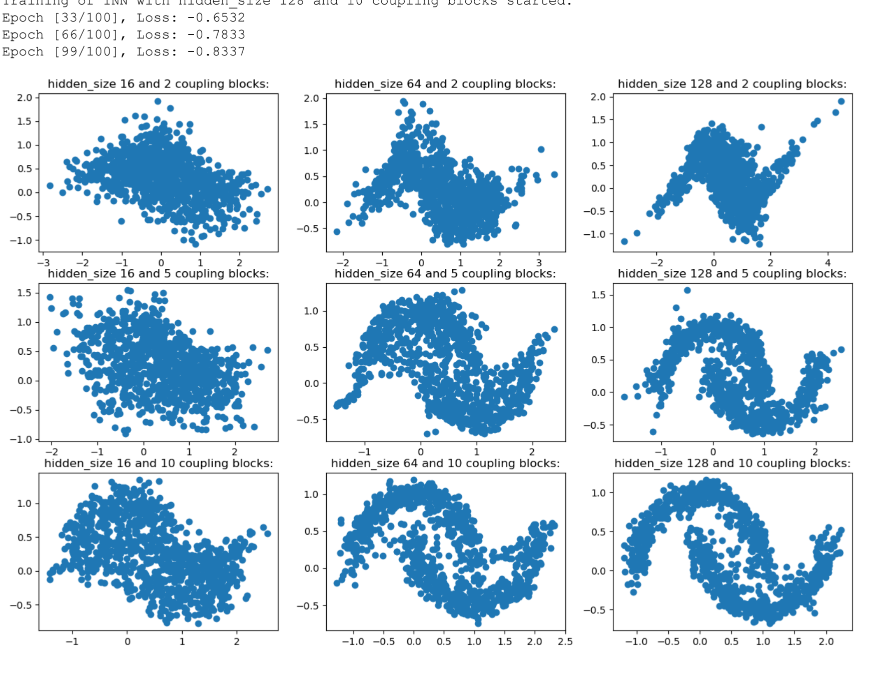
In [4]:
```
def train_inn(model, batchsize=1000, epochs=1000, lr=0.001):
    optimizer = torch.optim.Adam(params=model.parameters(), lr=lr)
    for epoch in range(epochs):
        optimizer.zero_grad()
        x_data, y_data = make_moons(n_samples=batchsize, shuffle=True, noise=0.1, random_state=42)
        x_data, y_data = torch.Tensor(x_data), torch.Tensor(y_data)

        z, ljd = model(x_data)

        loss = torch.sum(0.5*torch.sum(z**2, -1)-ljd) / batchsize
        loss.backward()

        optimizer.step()

        if (epoch+1) % (epochs//3) == 0:
            print(f"Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}")
```

In [5]:
```
def plot_samples(model, title="Generated samples from INN", axs=None):
    samples = model.sample(1000)
    samples = samples.detach().numpy()
    if axs is None:
        fig, axs = plt.subplots(1,1)
    axs.scatter(samples[:,0], samples[:,1])
    axs.set_title(title)
```

First we will compare the effect of changing the model parameters (width of the coupling nets and the number of coupling blocks), while training for a fixed number of 100 epochs and a fixed learning rate.

In [6]:
```
n_blocks = [2, 5, 10]
hidden_sizes = [16, 64, 128]

fig, axs = plt.subplots(3,3,figsize=(15,10))

for n in range(3):
    for h in range(3):
        model = realNVP(2, hidden_sizes[h], n_blocks[n])
        print(f"Training of INN with hidden_size {hidden_sizes[h]} and {n_blocks[n]} coupling blocks st
arted.")
        train_inn(model, epochs=100, lr=0.001)

        plot_samples(model, title=f"hidden_size {hidden_sizes[h]} and {n_blocks[n]} coupling blocks:",
axs=axs[n, h])
```

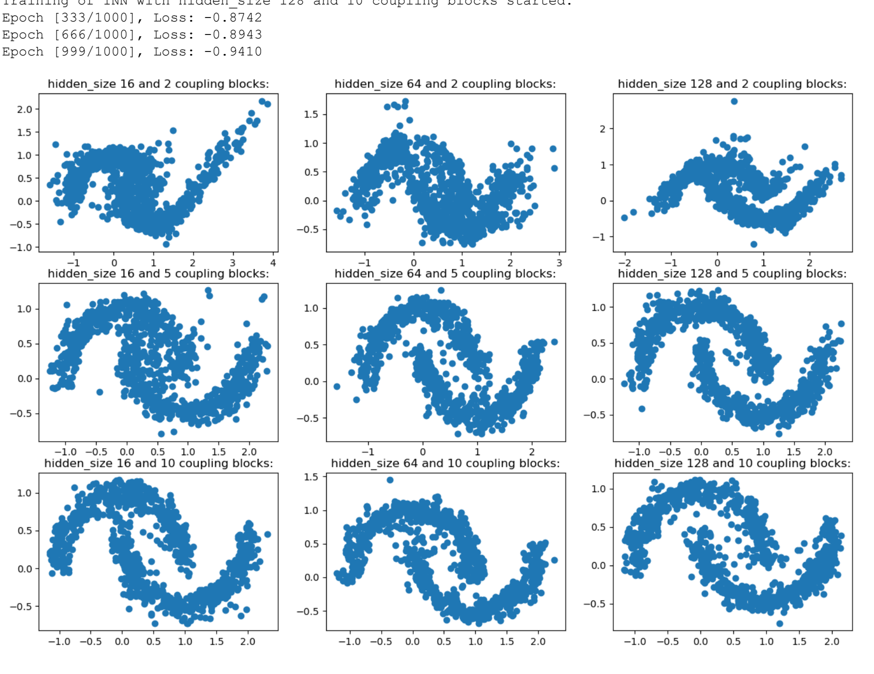
```
Training of INN with hidden_size 16 and 2 coupling blocks started.
Epoch [33/100], Loss: 0.2504
Epoch [66/100], Loss: 0.1504
Epoch [99/100], Loss: 0.0978
Training of INN with hidden_size 64 and 2 coupling blocks started.
Epoch [33/100], Loss: 0.0403
Epoch [66/100], Loss: -0.1422
Epoch [99/100], Loss: -0.2243
Training of INN with hidden_size 128 and 2 coupling blocks started.
Epoch [33/100], Loss: -0.1602
Epoch [66/100], Loss: -0.2389
Epoch [99/100], Loss: -0.2700
Training of INN with hidden_size 16 and 5 coupling blocks started.
Epoch [33/100], Loss: 0.3824
Epoch [66/100], Loss: 0.0974
Epoch [99/100], Loss: -0.0433
Training of INN with hidden_size 64 and 5 coupling blocks started.
Epoch [33/100], Loss: -0.2528
Epoch [66/100], Loss: -0.4679
Epoch [99/100], Loss: -0.5681
Training of INN with hidden_size 128 and 5 coupling blocks started.
Epoch [33/100], Loss: -0.4114
Epoch [66/100], Loss: -0.6675
Epoch [99/100], Loss: -0.7544
Training of INN with hidden_size 16 and 10 coupling blocks started.
Epoch [33/100], Loss: 0.0796
Epoch [66/100], Loss: -0.1491
Epoch [99/100], Loss: -0.3317
Training of INN with hidden_size 64 and 10 coupling blocks started.
Epoch [33/100], Loss: -0.5040
Epoch [66/100], Loss: -0.7008
Epoch [99/100], Loss: -0.7645
Training of INN with hidden_size 128 and 10 coupling blocks started.
Epoch [33/100], Loss: -0.6532
Epoch [66/100], Loss: -0.7833
Epoch [99/100], Loss: -0.8337
```



Now we will see how the models perform if we increase training length to 1000 epochs.

In [7]:
```
n_blocks = [2, 5, 10]
hidden_sizes = [16, 64, 128]

fig, axs = plt.subplots(3,3,figsize=(15,10))

for n in range(3):
    for h in range(3):
        model = realNVP(2, hidden_sizes[h], n_blocks[n])
        print(f"Training of INN with hidden_size {hidden_sizes[h]} and {n_blocks[n]} coupling blocks st
arted.")
        train_inn(model, epochs=1000, lr=0.001)

        plot_samples(model, title=f"hidden_size {hidden_sizes[h]} and {n_blocks[n]} coupling blocks:",
axs=axs[n, h])
```

```
Training of INN with hidden_size 16 and 2 coupling blocks started.
Epoch [333/1000], Loss: -0.2168
Epoch [666/1000], Loss: -0.3991
Epoch [999/1000], Loss: -0.4637
Training of INN with hidden_size 64 and 2 coupling blocks started.
Epoch [333/1000], Loss: -0.3950
Epoch [666/1000], Loss: -0.4374
Epoch [999/1000], Loss: -0.4461
Training of INN with hidden_size 128 and 2 coupling blocks started.
Epoch [333/1000], Loss: -0.5343
Epoch [666/1000], Loss: -0.5535
Epoch [999/1000], Loss: -0.5643
Training of INN with hidden_size 16 and 5 coupling blocks started.
Epoch [333/1000], Loss: -0.5159
Epoch [666/1000], Loss: -0.7038
Epoch [999/1000], Loss: -0.7540
Training of INN with hidden_size 64 and 5 coupling blocks started.
Epoch [333/1000], Loss: -0.7834
Epoch [666/1000], Loss: -0.8195
Epoch [999/1000], Loss: -0.8430
Training of INN with hidden_size 128 and 5 coupling blocks started.
Epoch [333/1000], Loss: -0.8407
Epoch [666/1000], Loss: -0.8804
Epoch [999/1000], Loss: -0.8752
Training of INN with hidden_size 16 and 10 coupling blocks started.
Epoch [333/1000], Loss: -0.7774
Epoch [666/1000], Loss: -0.8741
Epoch [999/1000], Loss: -0.9061
Training of INN with hidden_size 64 and 10 coupling blocks started.
Epoch [333/1000], Loss: -0.8839
Epoch [666/1000], Loss: -0.9264
Epoch [999/1000], Loss: -0.9492
Training of INN with hidden_size 128 and 10 coupling blocks started.
Epoch [333/1000], Loss: -0.8742
Epoch [666/1000], Loss: -0.8943
Epoch [999/1000], Loss: -0.9410
```

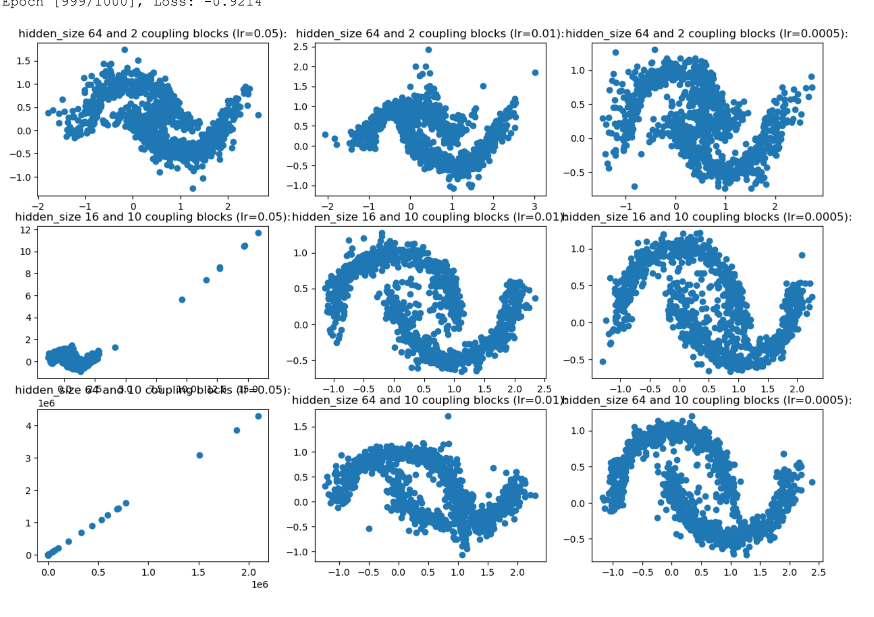


Interestingly, increasing the width of the coupling nets beyond 16 does not yield further improvements, but rather decreases the quality due to over-fitting. 10 seems to be a good choice for the number of coupling blocks.

We can also inspect changing the learning rate for training of different models.

In [8]:
```
learning_rates = [0.05, 0.01, 0.0005]
hidden_sizes = [64, 16, 64]
n_blocks = [2, 10, 10]
fig, axs = plt.subplots(3,3,figsize=(15,10))

for m in range(3):
    for lr in range(3):
        model = realNVP(2, hidden_sizes[m], n_blocks[m])
        print(f"Training of INN with hidden_size {hidden_sizes[m]} and {n_blocks[m]} coupling blocks st
arted with lr={learning_rates[lr]}.")
        train_inn(model, epochs=1000, lr=learning_rates[lr])
        plot_samples(model, title=f"hidden_size {hidden_sizes[m]} and {n_blocks[m]} coupling blocks (lr
={learning_rates[lr]}):", axs=axs[m, lr])
```

```
Training of INN with hidden_size 64 and 2 coupling blocks started with lr=0.05.
Epoch [333/1000], Loss: -0.5518
Epoch [666/1000], Loss: -0.5488
Epoch [999/1000], Loss: -0.5723
Training of INN with hidden_size 64 and 2 coupling blocks started with lr=0.01.
Epoch [333/1000], Loss: -0.5541
Epoch [666/1000], Loss: -0.5710
Epoch [999/1000], Loss: -0.5776
Training of INN with hidden_size 64 and 2 coupling blocks started with lr=0.0005.
Epoch [333/1000], Loss: -0.4852
Epoch [666/1000], Loss: -0.6299
Epoch [999/1000], Loss: -0.6443
Training of INN with hidden_size 16 and 10 coupling blocks started with lr=0.05.
Epoch [333/1000], Loss: -0.4545
Epoch [666/1000], Loss: 0.3741
Epoch [999/1000], Loss: -0.5322
Training of INN with hidden_size 16 and 10 coupling blocks started with lr=0.01.
Epoch [333/1000], Loss: -0.8798
Epoch [666/1000], Loss: -0.8606
Epoch [999/1000], Loss: -0.9354
Training of INN with hidden_size 16 and 10 coupling blocks started with lr=0.0005.
Epoch [333/1000], Loss: -0.7088
Epoch [666/1000], Loss: -0.8185
Epoch [999/1000], Loss: -0.8553
Training of INN with hidden_size 64 and 10 coupling blocks started with lr=0.05.
Epoch [333/1000], Loss: 6.2154
Epoch [666/1000], Loss: 5.8692
Epoch [999/1000], Loss: 3.8929
Training of INN with hidden_size 64 and 10 coupling blocks started with lr=0.01.
Epoch [333/1000], Loss: -0.7084
Epoch [666/1000], Loss: -0.5633
Epoch [999/1000], Loss: -0.7605
Training of INN with hidden_size 64 and 10 coupling blocks started with lr=0.0005.
Epoch [333/1000], Loss: -0.8403
Epoch [666/1000], Loss: -0.8982
Epoch [999/1000], Loss: -0.9214
```



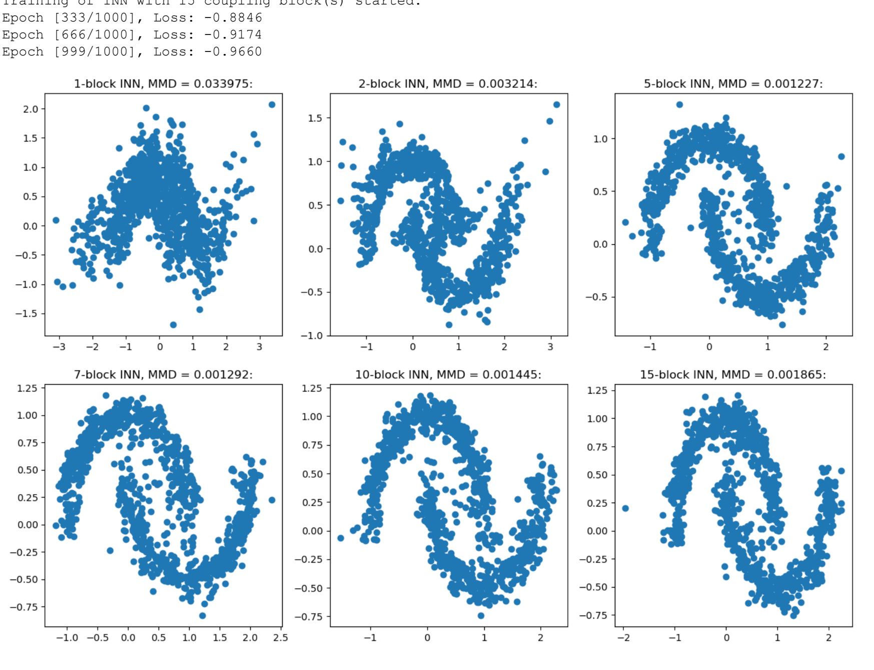
A learning rate which is too high can cause training instabilities as we see here. The default lr=0.001 seems to be a good choice.

Comparison of MMD scores with the function from the last sample solution:

In [9]:
```
def mmd_inverse_multi_quadratic(x, y, bandwidths=None):
    batch_size = x.size()[0]
    # compute the kernel matrices for each combination of x, y
    # (cleverly using broadcasting to do this efficiently)
    xx, yy, xy = torch.mm(x,x.t()), torch.mm(y,y.t()), torch.mm(x,y.t())
    rx = (xx.diag().unsqueeze(0).expand_as(xx))
    ry = (yy.diag().unsqueeze(0).expand_as(yy))
    # compute the sum of kernels at different bandwidths
    K, L, P = 0, 0, 0
    if bandwidths is None:
        bandwidths = [0.4, 0.8, 1.6]
    for sigma in bandwidths:
        s = 1.0 / sigma**2
        K += 1.0 / (1.0 + s * (rx.t() + rx - 2.0*xx))
        L += 1.0 / (1.0 + s * (ry.t() + ry - 2.0*yy))
        P += 1.0 / (1.0 + s * (rx.t() + ry - 2.0*xy))

    beta = 1./(batch_size*(batch_size-1)*len(bandwidths))
    gamma = 2./(batch_size**2 * len(bandwidths))
    return beta * (torch.sum(K)+torch.sum(L)) - gamma * torch.sum(P)
```

In [10]:
```
x, _ = make_moons(n_samples=1000, shuffle=True, noise=0.1, random_state=42)
x = torch.Tensor(x)

# use n_blocks as variable parameter for model quality
n_blocks = (1, 2, 5, 7, 10, 15)

fig, axs = plt.subplots(2,3,figsize=(15,10))
axs = axs.flatten()

for n in range(6):
    model = realNVP(2, 64, n_blocks[n])
    print(f"Training of INN with {n_blocks[n]} coupling block(s) started.")
    train_inn(model)

    samples = model.sample(1000)
    mmd = mmd_inverse_multi_quadratic(samples, x)

    plot_samples(model, title=f"{n_blocks[n]}-block INN, MMD = {mmd:.6f}:", axs=axs[n])
```

```
Training of INN with 1 coupling block(s) started.
Epoch [333/1000], Loss: -0.0052
Epoch [666/1000], Loss: -0.0167
Epoch [999/1000], Loss: -0.0186
Training of INN with 2 coupling block(s) started.
Epoch [333/1000], Loss: -0.6141
Epoch [666/1000], Loss: -0.6483
Epoch [999/1000], Loss: -0.6556
Training of INN with 5 coupling block(s) started.
Epoch [333/1000], Loss: -0.8566
Epoch [666/1000], Loss: -0.8859
Epoch [999/1000], Loss: -0.8986
Training of INN with 7 coupling block(s) started.
Epoch [333/1000], Loss: -0.8649
Epoch [666/1000], Loss: -0.9079
Epoch [999/1000], Loss: -0.9407
Training of INN with 10 coupling block(s) started.
Epoch [333/1000], Loss: -0.8840
Epoch [666/1000], Loss: -0.9127
Epoch [999/1000], Loss: -0.9369
Training of INN with 15 coupling block(s) started.
Epoch [333/1000], Loss: -0.8846
Epoch [666/1000], Loss: -0.9174
Epoch [999/1000], Loss: -0.9660
```



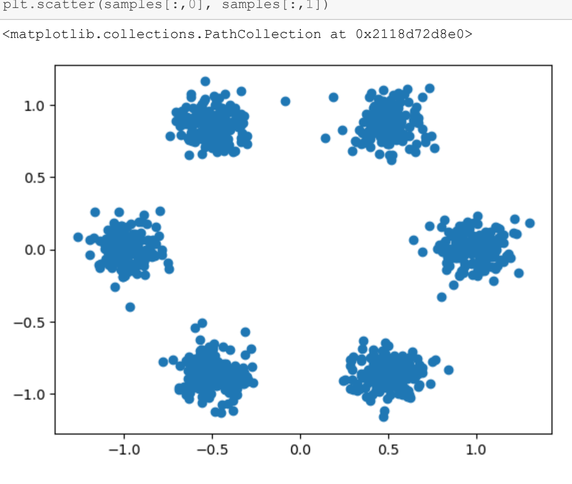
As expected, visually better samples correspond to lower MMD values. The decrease of quality for 10 coupling blocks seems to be a random artefact.

Gaussian Mixture Model (GMM):

In [11]:
```
def make_gmm_samples(n_samples):
    a = 0.5*math.sqrt(3)
    centers = torch.Tensor([[-1, 0],
              [-0.5, -a],
              [0.5, -a],
              [1, 0],
              [0.5, a],
              [-0.5, a]])
    stddev = torch.Tensor([0.1, 0.1])
    samples = []
    indices = []
    for i in range(n_samples):
        j = random.randint(0,5)
        indices.append(j)
        samples.append(torch.normal(centers[j], stddev))
    return torch.stack(samples), torch.Tensor(indices)
```

In [12]:
```
samples, _ = make_gmm_samples(1000)
plt.scatter(samples[:,0], samples[:,1])
```

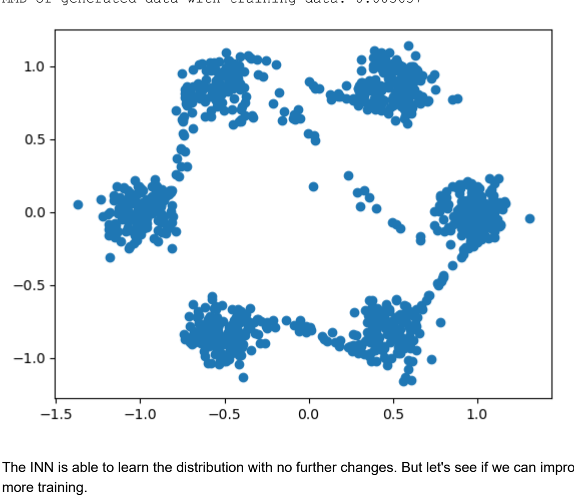
Out[12]:
```
<matplotlib.collections.PathCollection at 0x2118d72d8e0>
```



In [13]:
```
def train_gmm_inn(model, batchsize=1000, epochs=1000, lr=0.001):
    optimizer = torch.optim.Adam(params=model.parameters(), lr=lr)
    for epoch in range(epochs):
        optimizer.zero_grad()
        x_data, _ = make_gmm_samples(batchsize)

        z, ljd = model(x_data)

        loss = torch.sum(0.5*torch.sum(z**2, -1)-ljd) / batchsize
        loss.backward()

        optimizer.step()

        if (epoch+1) % (epochs//3) == 0:
            print(f"Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}")
```

In [14]:
```
gmm_model = realNVP(2, 64, 10)
train_gmm_inn(gmm_model)

samples = gmm_model.sample(1000)

plt.scatter(samples.detach().numpy()[:,0], samples.detach().numpy()[:,1])

true_data, _ = make_gmm_samples(1000)
print(f"MMD of generated data with training data: {mmd_inverse_multi_quadratic(samples, true_data):.6f}
")
```

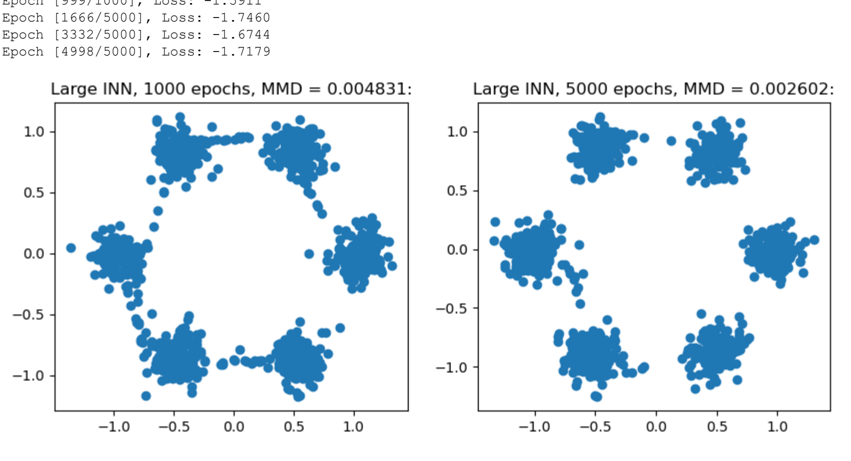
```
Epoch [333/1000], Loss: -1.4501
Epoch [666/1000], Loss: -1.5960
Epoch [999/1000], Loss: -1.5911
MMD of generated data with training data: 0.003657
```



The INN is able to learn the distribution with no further changes. But let's see if we can improve the performance with a larger network and more training.

In [15]:
```
n_epochs = [1000, 5000]
true_data, _ = make_gmm_samples(1000)

figs, axs = plt.subplots(1,2, figsize=(10, 4))
for e in range(2):
    larger_gmm_model=realNVP(2, 128, 15)
    train_gmm_inn(larger_gmm_model, epochs=n_epochs[e])
    samples = larger_gmm_model.sample(1000)

    axs[e].scatter(samples[:,0].detach().numpy(), samples[:,1].detach().numpy())
    axs[e].set_title(f"Large INN, {n_epochs[e]} epochs, MMD = {mmd_inverse_multi_quadratic(samples, tru
e_data):.6f}:")
```

```
Epoch [333/1000], Loss: -1.4595
Epoch [666/1000], Loss: -1.6455
Epoch [999/1000], Loss: -1.5911
Epoch [1666/5000], Loss: -1.7460
Epoch [3332/5000], Loss: -1.6744
Epoch [4998/5000], Loss: -1.7179
```



The larger architecture seems to be sufficient for an increase in visual quality, but further increasing the number of training epochs seems to have an effect on the amount of outliers. The MMD is not improved though.

2 Two Moons and GMM with a Conditional INN

In this exercise we are going to add functionlity to our RealNVP model, to turn it into a conditional INN. We will then compare how the cINN performs when learning condtional distributions $p(x|y)$ in comparison to learning the marginalized distribution $p(x) = \sum_y p(x|y)$. We will inspect the two moons case, the full GMM case and the GMM case where we cluster together 2 and 4 of the modes to make it a binary classification task.

We reuse most of the code for our classes coupling_block and realNVP.

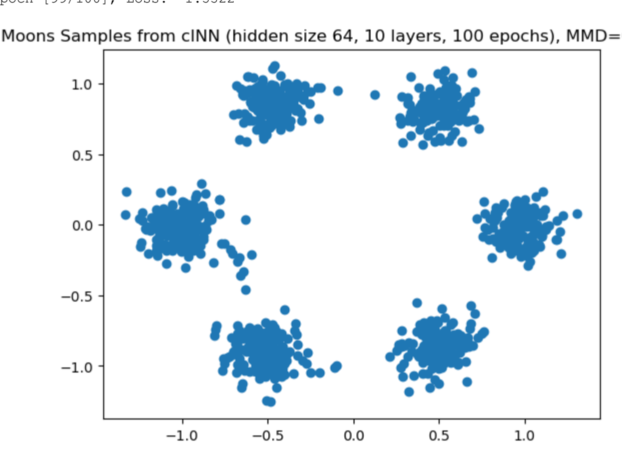
In [16]:
```
class conditional_coupling_block(nn.Module):
    def __init__(self, input_size, hidden_size, condition_size):
        super().__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.condition_size = condition_size

        self.split1 = math.floor(self.input_size/2)
        self.split2 = self.input_size - self.split1

        self.subnet = subnet_constructor(self.split1 + self.condition_size, self.hidden_size, 2*self.sp
lit2)

    def forward(self, x, cond, rev=False):
        x1, x2 = x[..., :self.split1], x[..., self.split1:]

        params = self.subnet(torch.cat([x1, cond], -1))
        s, t = params[...,:self.split2], params[...,self.split2:]
        s = torch.tanh(s)
        ljd = torch.sum(s, -1)


        if not rev:
            s = torch.exp(s)
            x2 = s*x2 + t
            return torch.cat([x1,x2], -1), ljd
        if rev:
            s = torch.exp(-s)
            x2 = s * (x2-t)
            return torch.cat([x1,x2], -1)

class conditional_realNVP(nn.Module):
    def __init__(self, input_size, hidden_size, n_blocks, condition_size):
        super().__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.n_blocks = n_blocks
        self.condition_size = condition_size

        self.coupling_blocks = nn.ModuleList([conditional_coupling_block(input_size, hidden_size, self.
condition_size) for _ in range(n_blocks)])
        self.orthogonal_matrices = [ortogonal_matrix(input_size) for _ in range(n_blocks-1)]

    def forward(self, x, cond, rev=False):
        if rev: return self._inverse(x, cond)
        return self._forward(x, cond)

    def _forward(self, x, cond):
        cond = nn.functional.one_hot(cond.to(torch.int64), num_classes=self.condition_size)
        ljd = torch.zeros((x.shape[0]))
        for l in range(self.n_blocks-1):
            x, partial_ljd = self.coupling_blocks[l](x, cond)
            ljd += partial_ljd
            x = torch.matmul(x, self.orthogonal_matrices[l])
        x, partial_ljd = self.coupling_blocks[-1](x, cond)
        ljd += partial_ljd
        return x, ljd

    def _inverse(self, x, cond):
        cond = nn.functional.one_hot(cond.to(torch.int64), num_classes=self.condition_size)
        for l in range (self.n_blocks-1, 0, -1):
            x = self.coupling_blocks[l](x, cond, rev=True)
            x = torch.matmul(x, self.orthogonal_matrices[l-1].T)
        x = self.coupling_blocks[0](x, cond, rev=True)
        return x

    def sample(self, num_samples, cond=None):
        samples = []
        if cond is None:
            for c in range(self.condition_size):
                z = torch.normal(mean=torch.zeros((num_samples, self.input_size)), std=torch.ones((num_
samples, self.input_size)))
                samples.append(self._inverse(z, cond=c*torch.ones(num_samples)))
        else:
            z = torch.normal(mean=torch.zeros((num_samples, self.input_size)), std=torch.ones((num_samp
les, self.input_size)))
            samples.append(self._inverse(z, cond=cond*torch.ones(num_samples)))
        return torch.cat(samples, 0)
```

In [17]:
```
def train_cinn_moons(model, batchsize=1000, epochs=1000, lr=0.001):
    optimizer = torch.optim.Adam(params=model.parameters(), lr=lr)
    for epoch in range(epochs):
        optimizer.zero_grad()
        x_data, y_data = make_moons(n_samples=batchsize, shuffle=True, noise=0.1, random_state=42)
        x_data, y_data = torch.Tensor(x_data), torch.Tensor(y_data)

        z, ljd = model(x_data, y_data)

        loss = torch.sum(0.5*torch.sum(z**2, -1)-ljd) / batchsize
        loss.backward()

        optimizer.step()

        if (epoch+1) % (epochs//3) == 0:
            print(f"Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}")
```

In [18]:

```
cINN_moons = conditional_realNVP(2, 64, 10, 2)
train_cinn_moons(cINN_moons, epochs=100)

moons_samples = cINN_moons.sample(1000)
true_moons_data, _ = make_moons(n_samples=2000, shuffle=True, noise=0.1, random_state=42)
true_moons_data = torch.Tensor(true_moons_data)

plt.scatter(samples[:,0].detach().numpy(), samples[:,1].detach().numpy())
mmd_score = mmd_inverse_multi_quadratic(moons_samples, true_moons_data)
plt.title(f"Moons Samples from cINN (hidden size 64, 10 layers, 100 epochs), MMD={mmd_score:.6f}")
plt.show()
```

```
Epoch [33/100], Loss: -1.0643
Epoch [66/100], Loss: -1.4710
Epoch [99/100], Loss: -1.5522
```



The conditional can learn the two moons distribution without problems.

Now we will try to learn the GMM with our cINN. We will add a function to cluster the top two modes into one class and the others into another class, so that we can train on the 6-class case and on a binary version of the GMM.

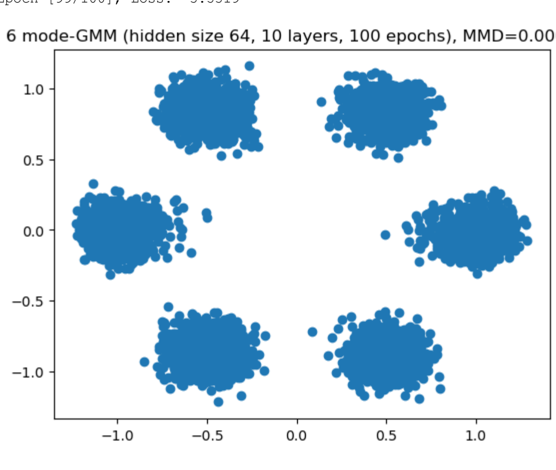
In [19]:

```
def cluster_mode_labels(cond):
    is_cluster_1 = torch.logical_or(cond == 4, cond == 5)
    return is_cluster_1.to(torch.int)

def train_cinn_gmm(model, batchsize=1000, epochs=1000, lr=0.001, binary=False):
    optimizer = torch.optim.Adam(params=model.parameters(), lr=lr)
    for epoch in range(epochs):
        optimizer.zero_grad()
        x_data, y_data = make_gmm_samples(1000)
        if binary: y_data = cluster_mode_labels(y_data)

        z, ljd = model(x_data, y_data)

        loss = torch.sum(0.5*torch.sum(z**2, -1)-ljd) / batchsize
        loss.backward()

        optimizer.step()

        if (epoch+1) % (epochs//3) == 0:
            print(f"Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}")
```

In [20]:

```
cINN_for_GMM = conditional_realNVP(2, 64, 10, 6)
train_cinn_gmm(cINN_for_GMM, epochs=100)

gmm_samples = cINN_for_GMM.sample(1000)
true_gmm_data, _ = make_gmm_samples(6000)

plt.scatter(gmm_samples[:,0].detach().numpy(), gmm_samples[:,1].detach().numpy())
mmd_score = mmd_inverse_multi_quadratic(gmm_samples, true_gmm_data)
plt.title(f"6 mode-GMM (hidden size 64, 10 layers, 100 epochs), MMD={mmd_score:.6f}")
plt.show()
```

```
Epoch [33/100], Loss: -2.7099
Epoch [66/100], Loss: -3.5366
Epoch [99/100], Loss: -3.5319
```

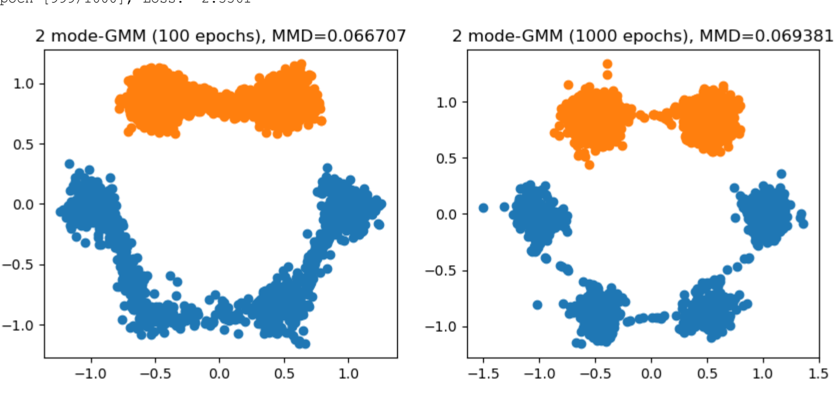


The cINN also has no problems with learning the conditional probabilities of the Gaussian Mixture Model. Now we will try the 2 mode case.

In [23]:

```
epochs = [100, 1000]

figs, axs = plt.subplots(1,2,figsize=(10,4))
axs = axs.flatten()

for e in range(2):
    cINN_for_binary_GMM = conditional_realNVP(2, 64, 10, 2)
    print(f"Training cINN for {epochs[e]} epochs.")
    train_cinn_gmm(cINN_for_binary_GMM, epochs=epochs[e], binary=True)

    binary_gmm_samples_1 = cINN_for_binary_GMM.sample(1000, cond=0)
    binary_gmm_samples_2 = cINN_for_binary_GMM.sample(2000, cond=1)
    binary_gmm_samples = torch.cat([binary_gmm_samples_1, binary_gmm_samples_2],0)
    true_gmm_data, _ = make_gmm_samples(3000)

    axs[e].scatter(binary_gmm_samples_1[:,0].detach().numpy(), binary_gmm_samples_1[:,1].detach().numpy
())
    axs[e].scatter(binary_gmm_samples_2[:,0].detach().numpy(), binary_gmm_samples_2[:,1].detach().numpy
())
    mmd_score = mmd_inverse_multi_quadratic(binary_gmm_samples, true_gmm_data)
    axs[e].set_title(f"2 mode-GMM ({epochs[e]} epochs), MMD={mmd_score:.6f}")
plt.show()
```

```
Training cINN for 100 epochs.
Epoch [33/100], Loss: -1.5827
Epoch [66/100], Loss: -1.9728
Epoch [99/100], Loss: -2.0259
Training cINN for 1000 epochs.
Epoch [333/1000], Loss: -2.2230
Epoch [666/1000], Loss: -2.3509
Epoch [999/1000], Loss: -2.3361
```



The performance is worse in this case. It is noticeable that there is more samples in between the modes, because the cINN does not learn them "independently" of one another. The MMD is significantly worse, and more training is required to achieve samples of quality comparable to that of the 6-mode GMM model. Interestingly, the model that was trained longer gives us a distribution which looks a lot better, but the MMD barely improved compared to the model which was trained less.

Finally we will compare the performance of a conditional INN (without actually using the condition functionality) with the performance of a comparable non-conditional INN, using the GMM distribution. To train the cINN on the marginal distribution $p(x)$ we need to write a new training function, with which the cINN receives no information about labels during training.

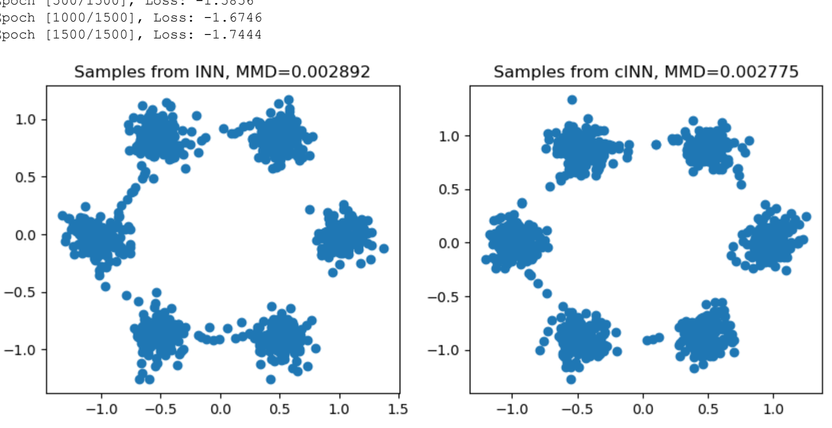
In [24]:

```
def train_cinn(model, batchsize=1000, epochs=1000, lr=0.001):
    optimizer = torch.optim.Adam(params=model.parameters(), lr=lr)
    for epoch in range(epochs):
        optimizer.zero_grad()
        x_data, _ = make_gmm_samples(batchsize)

        z, ljd = model(x_data, torch.zeros(batchsize))

        loss = torch.sum(0.5*torch.sum(z**2, -1)-ljd) / batchsize
        loss.backward()

        optimizer.step()

        if (epoch+1) % (epochs//3) == 0:
            print(f"Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}")
```

In [25]:

```
inn = realNVP(2, 64, 15)
cinn = conditional_realNVP(2, 64, 15, 1)

print("Training INN.")
train_gmm_inn(inn, epochs=1500)
print("Training cINN.")
train_cinn(cinn, epochs=1500)

true_gmm_data, _ = make_gmm_samples(1000)
inn_samples = inn.sample(1000)
cinn_samples = cinn.sample(1000)

figs, axs = plt.subplots(1,2,figsize=(10,4))
axs = axs.flatten()

axs[0].scatter(inn_samples[:,0].detach().numpy(), inn_samples[:,1].detach().numpy())
axs[0].set_title(f"Samples from INN, MMD={mmd_inverse_multi_quadratic(inn_samples, true_gmm_data):.6f}"
)
axs[1].scatter(cinn_samples[:,0].detach().numpy(), cinn_samples[:,1].detach().numpy())
axs[1].set_title(f"Samples from cINN, MMD={mmd_inverse_multi_quadratic(cinn_samples, true_gmm_data):.6f
}")

plt.show()
```

```
Training INN.
Epoch [500/1500], Loss: -1.6785
Epoch [1000/1500], Loss: -1.6900
Epoch [1500/1500], Loss: -1.6857
Training cINN.
Epoch [500/1500], Loss: -1.5856
Epoch [1000/1500], Loss: -1.6746
Epoch [1500/1500], Loss: -1.7444
```



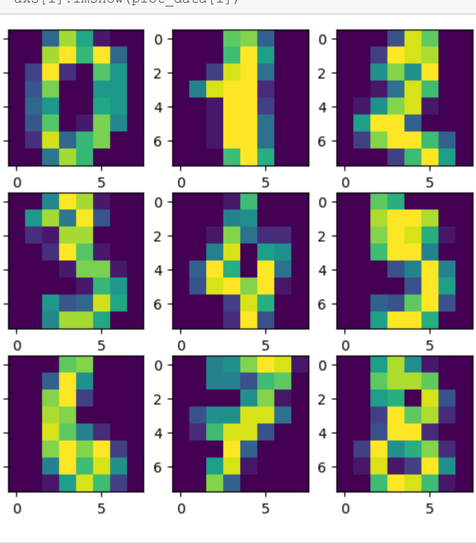
When training for the same duration and when using the same network hyperparameters, the INN performs better than the cINN.

3 Higher-dimensional data with an INN

(3.1) We start by loading the digits dataset and defining a INN which can learn its distribution.

In [26]:

```
digits_dataset = load_digits()
data = digits_dataset["data"]
plot_data = data.reshape((-1,8,8))
targets = digits_dataset["target"]

figs, axs = plt.subplots(3,3,figsize=(6,6))
axs = axs.flatten()

for i in range(9):
    axs[i].imshow(plot_data[i])
```



In [27]:

```
def train_digits_inn(model, epochs=1000, lr=0.001, digit=None):
    optimizer = torch.optim.Adam(params=model.parameters(), lr=lr)
    if digit is None: print("Training INN on all digits.")
    else: print(f"Training INN only only the digit {digit}.")
    for epoch in range(epochs):
        optimizer.zero_grad()
        digits_dataset = load_digits()
        data = digits_dataset["data"]
        if digit is not None:
            target = digits_dataset["target"]
            digit_idx = target == digit
            data = data[digit_idx]
        shuffle_idx = torch.randperm(data.shape[0])
        data = torch.Tensor(data[shuffle_idx]) # shuffle data

        z, ljd = model(data)

        loss = torch.sum(0.5*torch.sum(z**2, -1)-ljd) / data.shape[0]
        loss.backward()

        optimizer.step()

        if (epoch+1) % (epochs//10) == 0:
            print(f"Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}")
```

Here we added the option to train only on a single digit, to check how the performance differs in comparison to learning the whole distribution.

In [28]:

```
digits_inn = realNVP(64, 128, 8)
digit5_inn = realNVP(64, 128, 8)
train_digits_inn(digits_inn, epochs=2000)
train_digits_inn(digit5_inn, epochs=2000, digit=5)
```

```
Training INN on all digits.
Epoch [200/2000], Loss: 90.6462
Epoch [400/2000], Loss: 71.0559
Epoch [600/2000], Loss: 62.3353
Epoch [800/2000], Loss: 57.7789
Epoch [1000/2000], Loss: 54.5608
Epoch [1200/2000], Loss: 49.0357
Epoch [1400/2000], Loss: 45.7566
Epoch [1600/2000], Loss: 43.4063
Epoch [1800/2000], Loss: 42.3132
Epoch [2000/2000], Loss: 38.8255
Training INN only only the digit 5.
Epoch [200/2000], Loss: 52.8084
Epoch [400/2000], Loss: 31.4523
Epoch [600/2000], Loss: 14.1951
Epoch [800/2000], Loss: 2.9448
Epoch [1000/2000], Loss: -4.5249
Epoch [1200/2000], Loss: -11.9988
Epoch [1400/2000], Loss: -2.7596
Epoch [1600/2000], Loss: -22.1305
Epoch [1800/2000], Loss: -4.9696
Epoch [2000/2000], Loss: -19.9814
```

Without going into as much details of the hyperparamter optimization as in exercise 1, it is important to notice that network parameters that are too low do not work well due to not being powerful enough, but large network sizes and numbers of layers can indeed lead to overfitting in this case.

Let's check if the latent vectors are gaussian distributed, by plotting various 2d representations of the latent vectors of the first 500 datapoints.

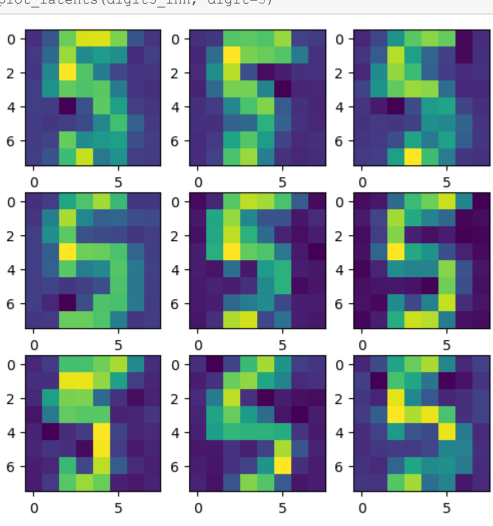
In [29]:

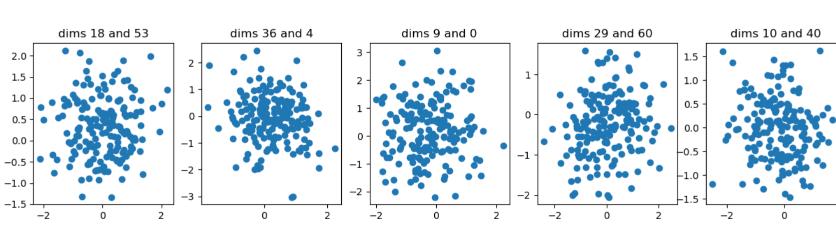
```
def plot_latents(model, digit=None):
    dims = ((18, 53), (36, 4), (9, 0), (29, 60), (10, 40))

    digits_dataset = load_digits()
    data = torch.Tensor(digits_dataset["data"])
    if digit is not None:
        target = torch.Tensor(digits_dataset["target"])
        digit_idx = target == digit
        data = data[digit_idx]

    figs, axs = plt.subplots(1, 5, figsize=(15,3))
    axs = axs.flatten()

    z, _ = model(data[:500])
    for p in range(5):
        axs[p].scatter(z[:,dims[p][0]].detach().numpy(), z[:,dims[p][1]].detach().numpy())
        axs[p].set_title(f"dims {dims[p][0]} and {dims[p][1]}")
```

Let's look at the latents and the samples from the INN which was only trained on the digit 5.

In [30]:

```
fives_samples = digit5_inn.sample(9)

figs, axs = plt.subplots(3,3,figsize=(6,6))
axs = axs.flatten()

for i in range(9):
    axs[i].imshow(fives_samples[i].reshape((8,8)).detach().numpy())

plot_latents(digit5_inn, digit=5)
```

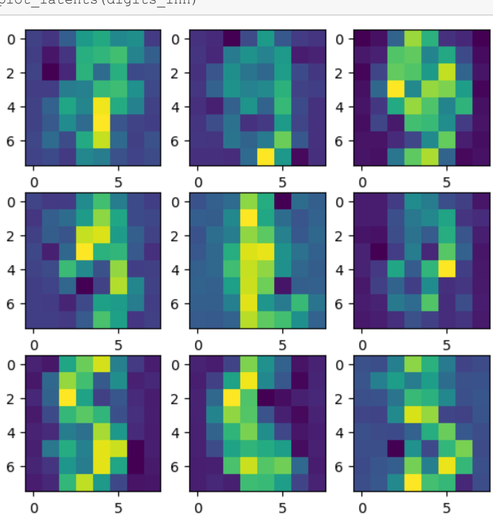


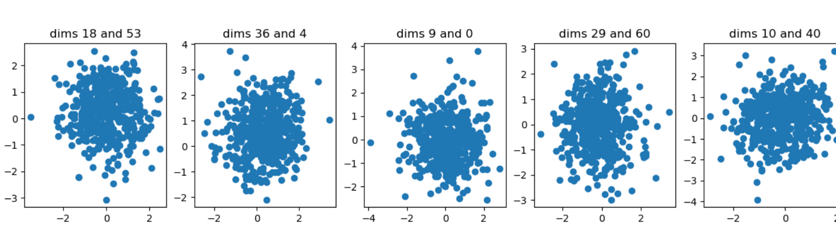


This already looks quite good. Now let's look at the INN trained on the whole dataset.

In [31]:

```
digits_samples = digits_inn.sample(9)

figs, axs = plt.subplots(3,3,figsize=(6,6))
axs = axs.flatten()

for i in range(9):
    axs[i].imshow(digits_samples[i].reshape((8,8)).detach().numpy())

plot_latents(digits_inn)
```





This model also seems to have learned the distribution, but the samples are not satisfactory yet. Finally we can calculate the MMD scores.

In [32]:

```
digits_dataset = load_digits()
data = torch.Tensor(digits_dataset["data"])
target = torch.Tensor(digits_dataset["target"])
fives_idx = target == 5
fives_data = data[fives_idx]

digits_samples_mmd = digits_inn.sample(1797)
fives_samples_mmd = digit5_inn.sample(182)

digits_mmd = mmd_inverse_multi_quadratic(data, digits_samples_mmd)
fives_mmd = mmd_inverse_multi_quadratic(fives_data, fives_samples_mmd)

print(f"MMD score for fives: {fives_mmd:.6f}")
print(f"MMD score for all digits: {digits_mmd:.6f}")
```

```
MMD score for fives: 0.011254
MMD score for all digits: 0.001167
```

(3.2) Now we introduce an artificial bottleneck by allowing the network to only reconstruct given images from the first 8 dimensions of the latent vector. Therefore we have to define a new network with a modified _inverse method, and adjust the training loop to consider the reconstruction loss. We also have to modify the .sample method, adding functionality to draw samples with fixed values for the first k dimensions, and a way to set all non-bottleneck dimensions to zero.

In [33]:

```
class bottleneck_realNVP(nn.Module):
    def __init__(self, input_size, hidden_size, n_blocks, bottleneck):
        super().__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.n_blocks = n_blocks
        self.bottleneck = bottleneck

        self.coupling_blocks = nn.ModuleList([coupling_block(input_size, hidden_size) for _ in range(n_
blocks)])
        self.orthogonal_matrices = [ortogonal_matrix(input_size) for _ in range(n_blocks-1)]

    def forward(self, x, rev=False):
        if rev: return self._inverse(x)
        return self._forward(x)

    def _forward(self, x):
        ljd = torch.zeros((x.shape[0]))
        for l in range(self.n_blocks-1):
            x, partial_ljd = self.coupling_blocks[l](x)
            ljd += partial_ljd
            x = torch.matmul(x, self.orthogonal_matrices[l])
        x, partial_ljd = self.coupling_blocks[-1](x)
        ljd += partial_ljd
        return x, ljd

    def _inverse(self, x):
        x_trunc = x[..., :self.bottleneck]
        x = torch.cat([x_trunc, torch.zeros((x.shape[0], self.input_size-self.bottleneck))], -1)
        for l in range (self.n_blocks-1, 0, -1):
            x = self.coupling_blocks[l](x, rev=True)
            x = torch.matmul(x, self.orthogonal_matrices[l-1].T)
        x = self.coupling_blocks[0](x, rev=True)
        return x

    def sample(self, num_samples, fixed_non_bottleneck=True):
        if not fixed_non_bottleneck:
            z = torch.normal(mean=torch.zeros((num_samples, self.input_size)), std=torch.ones((num_samp
les, self.input_size)))
        if fixed_non_bottleneck:
            z0 = torch.normal(mean=torch.zeros((num_samples, self.bottleneck)), std=torch.ones((num_sam
ples, self.bottleneck)))
            z = torch.cat([z0, torch.zeros((num_samples, self.input_size - self.bottleneck))], -1)
        return self._inverse(z)
```

In [34]:

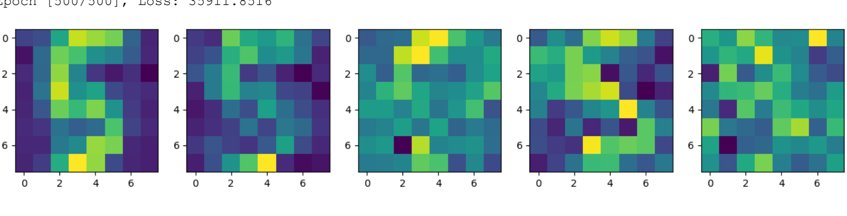
```
def train_bn_inn(model, epochs=1000, lr=0.001, digit=None, l=1):
    optimizer = torch.optim.Adam(params=model.parameters(), lr=lr)
    if digit is None: print("Training INN on all digits.")
    else: print(f"Training INN only with the digit {digit}.")
    for epoch in range(epochs):
        optimizer.zero_grad()
        digits_dataset = load_digits()
        data = digits_dataset["data"]
        if digit is not None:
            target = digits_dataset["target"]
            digit_idx = target == digit
            data = data[digit_idx]
        shuffle_idx = torch.randperm(data.shape[0])
        data = torch.Tensor(data[shuffle_idx]) # shuffle data

        z, ljd = model(data)
        nll_loss = torch.sum(0.5*torch.sum(z**2, -1)-ljd) / data.shape[0]

        x_rec = model(z, rev=True)
        diff = x_rec - data
        mse_loss = torch.sum(diff**2) / data.shape[0]

        loss = nll_loss + l*mse_loss

        loss.backward()

        optimizer.step()

        if (epoch+1) % (epochs//5) == 0:
            print(f"Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}")
```

Training with the reconstruction error introduces a new hyperparameter lambda (or l) which weights the losses against each other. We will look for the best value for l for the single-digit case first.

In [35]:

```
l_values = (0.01, 0.1, 1., 10., 100.)

figs, axs = plt.subplots(1,5,figsize=(15,3))
axs = axs.flatten()

for l in range(5):
    bn_fives_inn = bottleneck_realNVP(64, 128, 8, 8)
    train_bn_inn(bn_fives_inn, epochs=500, digit=5, l=l_values[l])

    sample = bn_fives_inn.sample(1)
    axs[l].imshow(sample.reshape((8,8)).detach().numpy())
```

```
Training INN only with the digit 5.
Epoch [100/500], Loss: 158.1087
Epoch [200/500], Loss: 104.4336
Epoch [300/500], Loss: 85.4504
Epoch [400/500], Loss: 72.2620
Epoch [500/500], Loss: 62.3509
Training INN only with the digit 5.
Epoch [100/500], Loss: 218.0928
Epoch [200/500], Loss: 160.3393
Epoch [300/500], Loss: 135.5841
Epoch [400/500], Loss: 121.4825
Epoch [500/500], Loss: 110.0759
Training INN only with the digit 5.
Epoch [100/500], Loss: 799.7762
Epoch [200/500], Loss: 489.2112
Epoch [300/500], Loss: 390.4735
Epoch [400/500], Loss: 338.8766
Epoch [500/500], Loss: 300.6317
Training INN only with the digit 5.
Epoch [100/500], Loss: 9168.9795
Epoch [200/500], Loss: 4577.6582
Epoch [300/500], Loss: 3466.3394
Epoch [400/500], Loss: 2865.8110
Epoch [500/500], Loss: 2433.5530
Training INN only with the digit 5.
Epoch [100/500], Loss: 170519.1875
Epoch [200/500], Loss: 66946.7188
Epoch [300/500], Loss: 48661.0312
Epoch [400/500], Loss: 40560.2227
Epoch [500/500], Loss: 35911.8516
```



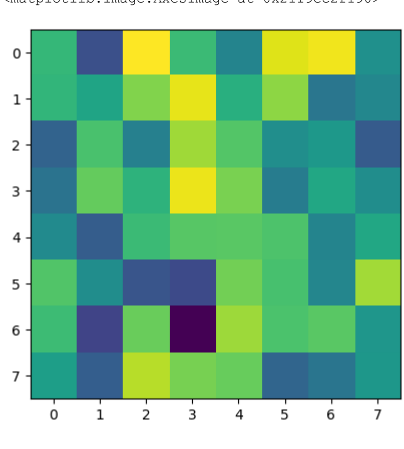
0.01 seems to be a good value for l.

In [36]:

```
bn_inn = bottleneck_realNVP(64, 512, 8, 8)
train_bn_inn(bn_inn, epochs=1000, l=0.01)

sample = bn_fives_inn.sample(1)
plt.imshow(sample.reshape((8,8)).detach().numpy())
```

```
Training INN on all digits.
Epoch [200/1000], Loss: 116.8179
Epoch [400/1000], Loss: 87.2068
Epoch [600/1000], Loss: 76.5333
Epoch [800/1000], Loss: 65.6031
Epoch [1000/1000], Loss: 60.1841
```

Out[36]: <matplotlib.image.AxesImage at 0x2119ec2f190>



Unfortunately, the bottleneck network was not able to learn the digits dataset.

4 Higher-dimensional data with a conditional INN

Now we try to learn the digits dataset with a conditional INN. We will evaluate the generated samples by plotting some of them and checking, if a random forest classifier is able to classify them.

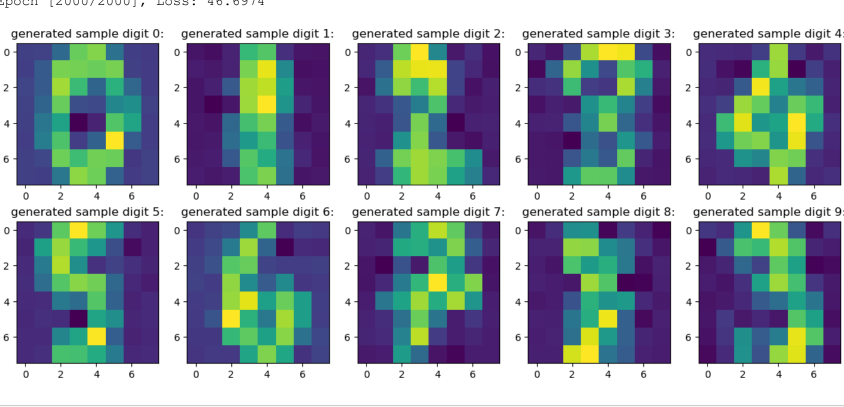
In [37]:

```
def train_digits_cinn(model, batchsize=1000, epochs=100, lr=0.001):
    optimizer = torch.optim.Adam(params=model.parameters(), lr=lr)
    for epoch in range(epochs):
        optimizer.zero_grad()

        digits_dataset = load_digits()
        images = torch.Tensor(digits_dataset["data"])
        labels = torch.Tensor(digits_dataset["target"])

        z, ljd = model(images, labels)

        loss = torch.sum(0.5*torch.sum(z**2, -1)-ljd) / batchsize
        loss.backward()

        optimizer.step()

        if (epoch+1) % (epochs//5) == 0:
            print(f"Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}")
```

In [38]:

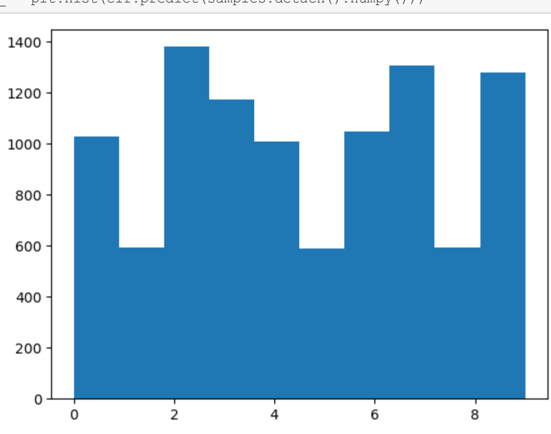
```
digits_cinn = conditional_realNVP(64, 512, 8, 10)
train_digits_cinn(digits_cinn, epochs=2000)
sample = digits_cinn.sample(1)

figs, axs = plt.subplots(2, 5, figsize=(15, 6))
axs = axs.flatten()

for i in range(10):
    axs[i].imshow(sample[i].reshape((8,8)).detach().numpy())
    axs[i].set_title(f"generated sample digit {i}:")
```

```
Epoch [400/2000], Loss: 101.4222
Epoch [800/2000], Loss: 74.3553
Epoch [1200/2000], Loss: 54.1391
Epoch [1600/2000], Loss: 84.1828
Epoch [2000/2000], Loss: 46.6974
```



In [39]:

```
from sklearn.ensemble import RandomForestClassifier
clf = RandomForestClassifier(max_depth=2, random_state=0)

digits_dataset = load_digits()
images = digits_dataset["data"]
labels = digits_dataset["target"]
clf.fit(images, labels)

samples = digits_cinn.sample(1000)
_ = plt.hist(clf.predict(samples.detach().numpy()))
```



The samples are not evenly distributed, but one can see that the network is in principle able to learn the digits dataset.