In [1]:
```
%load_ext autoreload
%autoreload 2
import numpy as np
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.distributions.multivariate_normal import MultivariateNormal
from sklearn.datasets import make_moons

from nf_utils import *
```

In [2]:
```
device = 'cuda'
#device = 'cpu'  # uncomment this line to run the model on the GPU
```

Project 1, part 1: Normalizing Flows (90 pt)

In this notebook we implement 2 Normalizing Flows (NF), stack tranformations and train them with maximum likelihood on 3 datasets.

Your task

Complete the missing code. Make sure that all the functions follow the provided specification, i.e. the output of the function exactly matches the description in the docstring.

Do not add or modify any code outside of the following comment blocks

```
##########################################################
# YOUR CODE HERE
.....
##########################################################
```

After you fill in all the missing code, restart the kernel and re-run all the cells in the notebook.

The following things are **NOT** allowed:

- Using additional `import` statements
- Using `torch.autograd.functional.jacobian`
- Using `torch.det`
- Using `torch.distributions`
- Copying / reusing code from other sources (e.g. code by other students)

If you plagiarise even for a single project task, you won't be eligible for the bonus this semester.

1. Normalizing Flow

We provide a base class for normalizing flows called Flow with functions to implement when creating a new Flow:

- Forward pass: (Slide 31)
- Reverse pass (if it exists in closed form): (Slide 27)

Additonally, the class InverseFlow inverse the parametrization fo a Flow i.e. it uses the forward pass for the reverse parametrization and the reverse pass for forward parametrization.

In this section, we aim at implementing two NF transformations:

- Affine transformation
- Radial transformation

Affine tranformation

An affine tranformation is able to scale each dimension independently and shift a given distribution. The tranformation is descibed as follows:

$$f(\mathbf{z}) = \exp(\mathbf{a}) \odot \mathbf{z} + \mathbf{b}$$

where parameters $\mathbf{a} \in \mathbb{R}^D$ and $\mathbf{b} \in \mathbb{R}^D$. We apply $\exp$ elementwise to $\mathbf{a}$ to obtain positive scales for each dimension.

Note that this transformation is invertible.

Task 1: Affine - forward (10 pt)

Implement the forward method in the class Affine

Task 2: Affine - inverse (10 pt)

Implement the inverse method in the class inverse

In [3]:
```
class Affine(Flow):
    """Affine transformation y = a * x + b."""

    def __init__(self, dim=2):
        """Create and init an affine transformation.

        Args:
            dim: dimension of input/output data. int
        """
        super().__init__()
        self.dim = dim
        self.log_scale = nn.Parameter(torch.zeros(self.dim))  # a
        self.shift = nn.Parameter(torch.zeros(self.dim))  # b

    def forward(self, x):
        """Compute the forward transformation given an input x.

        Args:
            x: input sample. shape [batch_size, dim]

        Returns:
            y: sample after forward tranformation. shape [batch_size, dim]
            log_det_jac: log determinant of the jacobian of the forward tran
formation, shape [batch_size]
        """
        ##########################################################
        # YOUR CODE HERE
        batch_size = x.size()[0]
        y = torch.exp(self.log_scale) * x + self.shift

        log_det_jac = torch.zeros(batch_size,).to(device)
        log_det_jac += self.log_scale.sum()

        ##########################################################
        return y, log_det_jac

    def inverse(self, y):
        """Compute the inverse transformation given an input y.

        Args:
            y: input sample. shape [batch_size, dim]

        Returns:
            x: sample after inverse tranformation. shape [batch_size, dim]
            inv_log_det_jac: log determinant of the jacobian of the inverse
tranformation, shape [batch_size]
        """
        ##########################################################
        # YOUR CODE HERE
        batch_size = y.size()[0]

        x = (y - self.shift)/torch.exp(self.log_scale)


        inv_log_det_jac = torch.zeros(batch_size,).to(device)
        inv_log_det_jac += -self.log_scale.sum()


        ##########################################################
        return x, inv_log_det_jac
```

Radial tranformation

A radial flow is a simple but expressive transformation. It has been introduced in this paper (https://arxiv.org/pdf/1505.05770.pdf) from Rezende and Mohamed. The transformation can be described as follows:

$$f(\mathbf{z}) = \mathbf{z} + \beta h(\alpha, r)(\mathbf{z} - \mathbf{z_0})$$

where $r = \mathbf{z} - \mathbf{z_0}$, $h(\alpha, r) = \frac{1}{\alpha + r}$ and parameters $\mathbf{z_0} \in \mathbb{R}^D$, $\alpha \in \mathbb{R}_+$ and $\beta \in \mathbb{R}$. To be an invertible tranformation, the parameters should satisfy $\beta \geq -\alpha$. Implement the radial flow and ensure that the tranformation is valid with a relevant parametrization. Hints:

- You can for example consider the exponential or softplus functions to tranform a parameter $\alpha \in \mathbb{R}$ to softplus$(\alpha) \in \mathbb{R}_+$.
- To enforce the inequality constraint on β, you can use an additional variable e.g. $\beta = -\alpha + \text{softplus}(\tilde{\beta}) > -\alpha$.

Task 3: Radial - forward (20 pt)

Implement the forward method in the class Radial

In [4]:
```
class Radial(Flow):
    """Radial transformation."""

    def __init__(self, dim=2):
        """Create and init an affine transformation.

        Args:
            dim: dimension of input/output data. int
        """
        super().__init__()
        self.dim = 2
        self.x0 = nn.Parameter(torch.Tensor(self.dim,)) # Vector used to par
ametrize z_0
        self.pre_alpha = nn.Parameter(torch.Tensor(1,)) # Scalar used to ind
irectly parametrized \alpha
        self.pre_beta = nn.Parameter(torch.Tensor(1,)) # Scaler used to indi
reclty parametrized \beta

        stdv = 1. / np.sqrt(self.dim)
        self.pre_alpha.data.uniform_(-stdv, stdv)
        self.pre_beta.data.uniform_(-stdv, stdv)
        self.x0.data.uniform_(-stdv, stdv)

    def forward(self, x):
        """Compute the forward transformation given an input x.

        Args:
            x: input sample. shape [batch_size, dim]

        Returns:
            y: sample after forward tranformation. shape [batch_size, dim]
            log_det_jac: log determinant of the jacobian of the forward tran
formation, shape [batch_size]
        """
        ##########################################################
        # YOUR CODE HERE
        batch_size = x.size()[0]
        alpha = F.softplus(self.pre_alpha)
        beta = -alpha + F.softplus(self.pre_beta)

        #get a tensor with size [100]
        r = (x-self.x0).norm(dim=1)

        h_alpha_r = 1/(alpha + r)
        h_alpha_r_copy = torch.zeros(batch_size,self.dim).to(device)

        h_alpha_r_copy[:,0] += h_alpha_r
        h_alpha_r_copy[:,1] += h_alpha_r


        h_alpha_r_dev = -1/torch.pow(alpha + r,2)
        y = x + beta * h_alpha_r_copy * (x-self.x0)

        log_det_jac = (torch.pow(1 + beta * h_alpha_r,self.dim-1)*\
                      ((1 + beta * h_alpha_r)+(beta * h_alpha_r_dev * r))).l
og()


        ##########################################################
        return y, log_det_jac

    def inverse(self, y):
        """Compute the inverse transformation given an input y.

        Args:
            y: input sample. shape [batch_size, dim]

        Returns:
```

Stacking tranformations

We want to stack normalizing flow transformations. From the stacked tranformations, we want to be able to evaluate densities (slide 27: Reverse transformation) and/or sample points (slide 31: Forward transformation).

Task 4: stacking tranformations - log_prob (20 pt)

Implement the method log_prob in class StackedFlows. This method should compute tthe log density for each sample.

Task 5: stacking tranformations - rsample (20 pt)

Implement the method rsample in class StackedFlows. This method should sample from the transforamed distribution and compute its log density.

In [5]:
```
class StackedFlows(nn.Module):
    def __init__(self, transforms, dim=2, base_dist='Normal'):
        """Stack a list of tranformations with a given based distribtuion.

        Args:
            tranforms: list fo stacked tranformations. list of Flows
            dim: dimension of input/output data. int
            base_dist: name of the base distribution. options: ['Normal']
        """
        super().__init__()
        if isinstance(transforms, Flow):
            self.transforms = nn.ModuleList([transforms, ])
        elif isinstance(transforms, list):
            if not all(isinstance(t, Flow) for t in transforms):
                raise ValueError("transforms must be a Flow or a list of Flo
ws")
            self.transforms = nn.ModuleList(transforms)
        else:
            raise ValueError(f"transforms must a Flow or a list, but was {ty
pe(transforms)}")
        self.dim = dim
        if base_dist == "Normal":
            self.base_dist = MultivariateNormal(torch.zeros(self.dim).to(dev
ice), torch.eye(self.dim).to(device))
        else:
            raise NotImplementedError

    def log_prob(self, x):
        """Compute log probability of a batch of data (slide 27).

        Args:
            x: input sample. shape [batch_size, dim]

        Returns:
            log_prob: Log probability of the data, shape [batch_size]
        """
        ##########################################################
        # YOUR CODE HERE
        batch_size = x.size()[0]
        log_prob = torch.zeros(batch_size).to(device)

        for index,transform in enumerate (self.transforms):
            x,inv_log_det_jac = self.transforms[- index - 1].inverse(x)
            log_prob += inv_log_det_jac
        log_prob += self.base_dist.log_prob(x)

        ##########################################################
        return log_prob

    def rsample(self, batch_size):
        """Sample from the transformed distribution (slide 31).

        Returns:
            x: sample after forward tranformation, shape [batch_size, dim]
            log_prob: Log probability of x, shape [batch_size]
        """
        ##########################################################
        # YOUR CODE HERE

        log_prob = torch.zeros(batch_size).to(device)
        x = self.base_dist.sample((batch_size,))
        x0_prob = self.base_dist.log_prob(x)
        for index,transform in enumerate (self.transforms):
            x,log_det_jac = self.transforms[index].forward(x)
            log_prob -= log_det_jac
        log_prob += x0_prob
```

2. Maximum-likelihood training

We train normalizing flows by maximizing the likelihood (Slide 28) of the data $x^{(i)}$ w.r.t. the flow parameters ϕ i.e.:

$$\max_{\phi} \frac{1}{|\mathcal{D}|} \sum_{x^{(i)} \in \mathcal{D}} logp(x^{(i)})$$

Task 6: training - max-likelihood (10 pt)

Complete the functions train such that it trains the model with maximum-likelihood.

The variable loss should be a scalar equal to the the mean loss for the data in the current batch. Note that here we expect to minimize the negaitve log-likelihood instead of maximizing the log-likelihood.

In [6]:
```
def train(model, dataset, batch_size=100, max_epochs=1000, frequency=250):
    """Train a normalizing flow model with maximum likelihood.

    Args:
        model: normalizing flow model. Flow or StackedFlows
        dataset: dataset containing data to fit. Dataset
        batch_size: number of samples per batch. int
        max_epochs: number of training epochs. int
        frequency: frequency for plotting density visualization. int

    Return:
        model: trained model. Flow or StackedFlows
        losses: loss evolution during training. list of floats
    """
    # Load dataset
    train_loader = torch.utils.data.DataLoader(dataset, batch_size=batch_siz
e)

    # Train model
    losses = []
    optimizer = torch.optim.Adam(model.parameters(), lr=1e-3, weight_decay=1
e-6)
    for epoch in range(max_epochs + 1):
        total_loss = 0
        for batch_index, (X_train) in enumerate(train_loader):
            ###########################################################
            # YOUR CODE HERE
            #X_train.size()=torch.Size([batch_size, 2]) 2 is the dimension o
f the data
            X_train = X_train.to(device)
            loss = -model.log_prob(X_train).sum()/batch_size


            ###########################################################
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            total_loss += loss
        total_loss /= len(train_loader)
        losses.append(total_loss)

        if epoch % frequency == 0:
            print(f"Epoch {epoch} -> loss: {total_loss.item():.2f}")
            plot_density(model, train_loader, device=device)

    return model, losses
```

3. Results

In this section we use three 2D datasets:

- A single Gaussian with non-zero mean
- Three gaussians
- Two moons

For each dataset, we train an affine and a radial transformation with a Gaussian base distribution. The affine tranformation should only be able to scale and shift the base distribution. The radial tranformation is capable of more complex transformations.

Plots show:

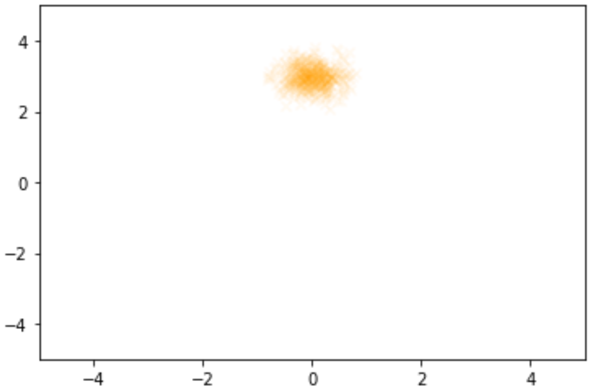
- Evolution of density estimation during training.
- The loss curve during training.
- The density learned by the model after training.
- Samples from the model after training (if possible).

If it learns corretly, the density estimation significantly changes after 100 epochs already for each toy dataset.

Dataset 1: shifted Gaussian

The first dataset composed of one Gaussian with a non zero mean. All flows should manage to fit this density.

In [7]:
```
dataset_1 = CircleGaussiansDataset(n_gaussians=1, n_samples=500)
plt.scatter(dataset_1.X[:,0], dataset_1.X[:,1], alpha=.05, marker='x', c='or
ange')
plt.xlim(-5, 5)
plt.ylim(-5, 5)
plt.show()
```

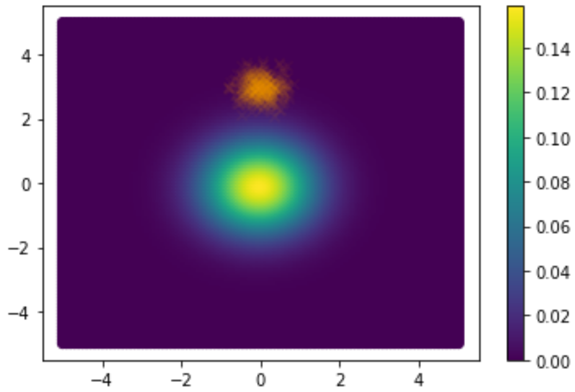


Affine flow

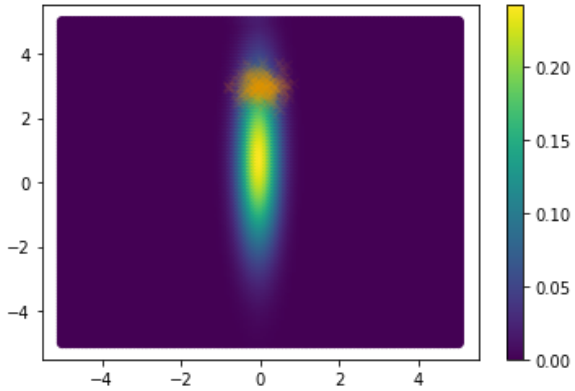
In [8]:
```
transforms = [Affine().to(device)]
model = StackedFlows(transforms, base_dist='Normal').to(device)
model, losses = train(model, dataset_1, max_epochs=1500)

# Plots
plt.plot(losses)
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.show()
plot_density(model, [], device=device)
plot_samples(model)
```

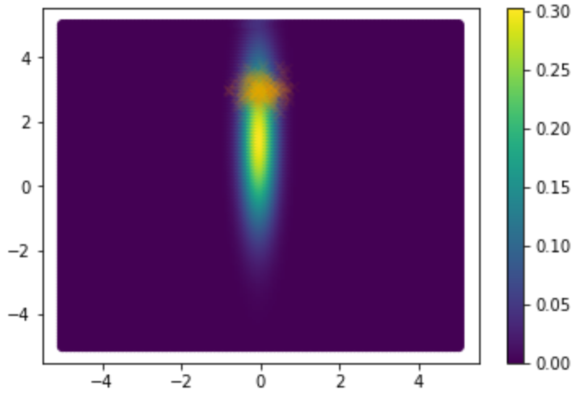
```
Epoch 0 -> loss: 6.34
```



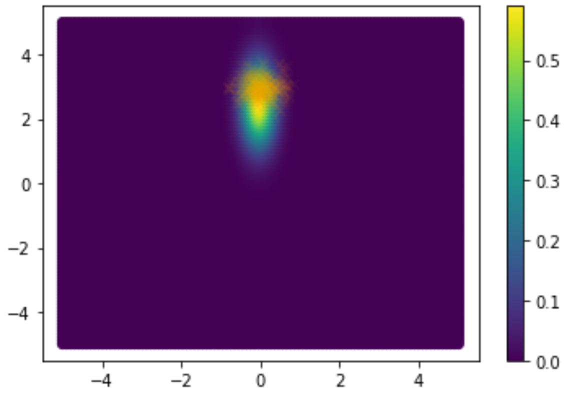

```
Epoch 250 -> loss: 2.46
```



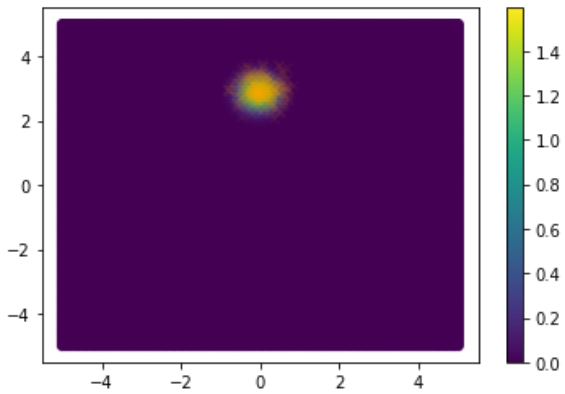

```
Epoch 500 -> loss: 2.04
```



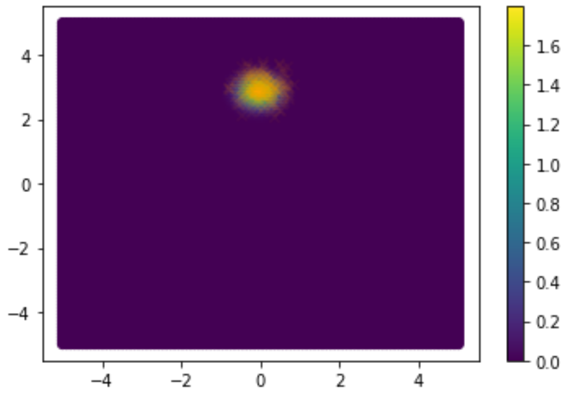

```
Epoch 750 -> loss: 1.18
```



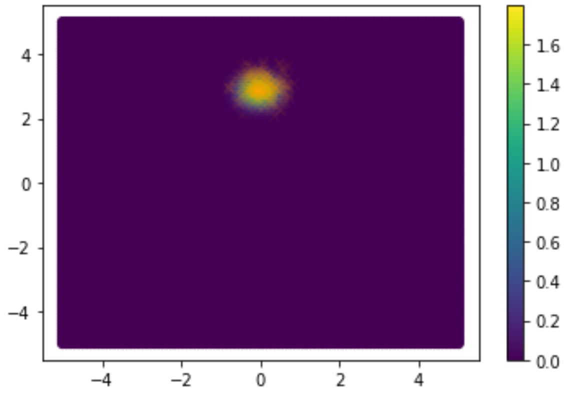
```
Epoch 1000 -> loss: 0.42
```

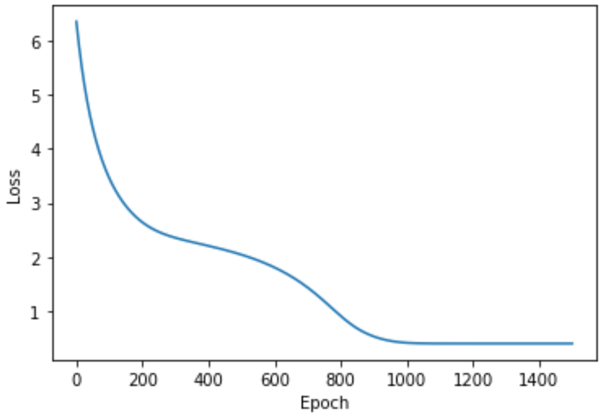


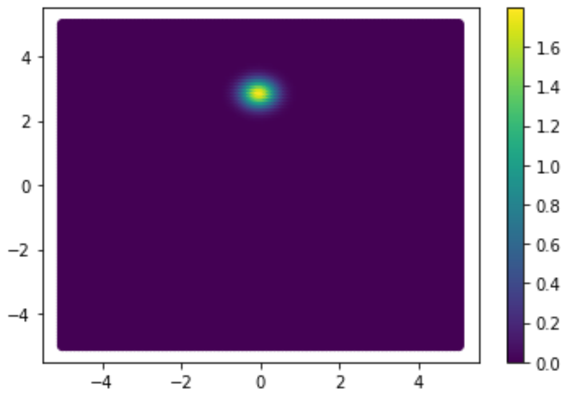
```
Epoch 1250 -> loss: 0.40
```

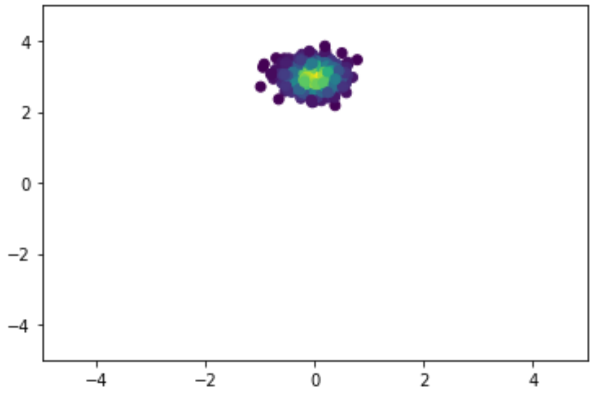


```
Epoch 1500 -> loss: 0.40
```







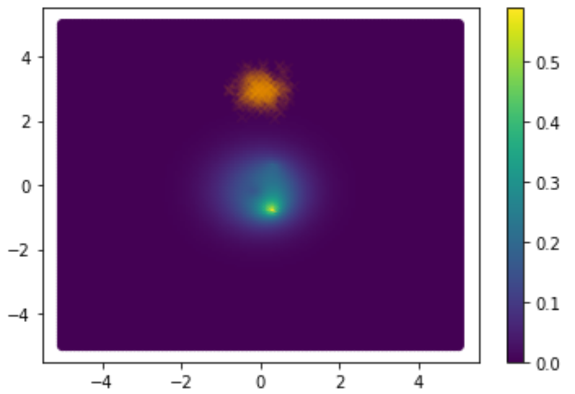


Radial flow (4 layers)

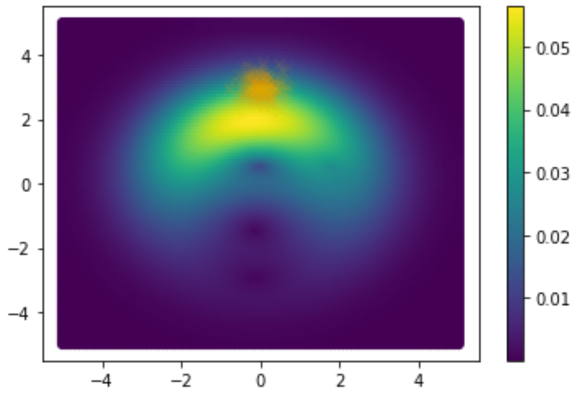
In [9]:
```
transforms = [InverseFlow(Radial()).to(device) for _ in range(4)]
model = StackedFlows(transforms, base_dist='Normal').to(device)
model, losses = train(model, dataset_1, max_epochs=1500)

# Plots
plt.plot(losses, marker='*')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.show()
plot_density(model, [], device=device)
```

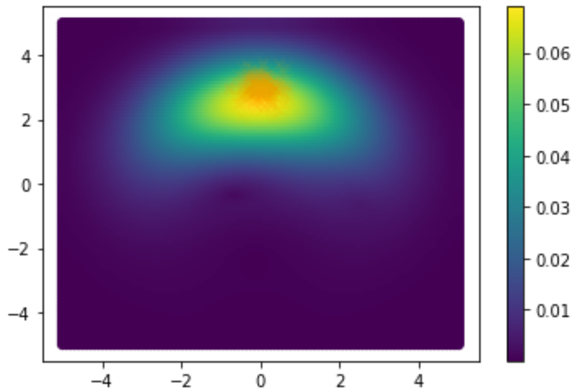
```
Epoch 0 -> loss: 7.76
```



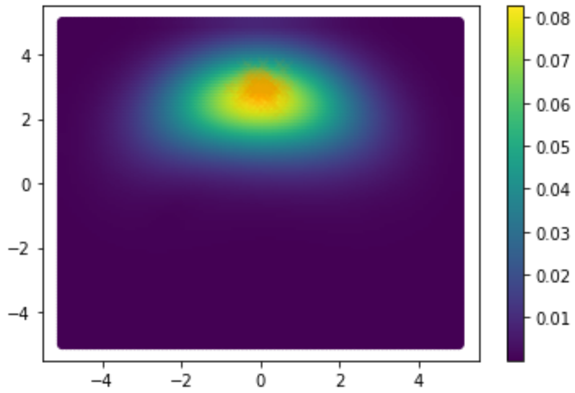

```
Epoch 250 -> loss: 3.31
```



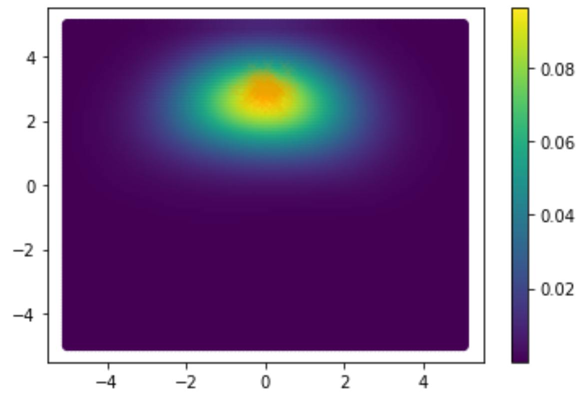

```
Epoch 500 -> loss: 2.78
```



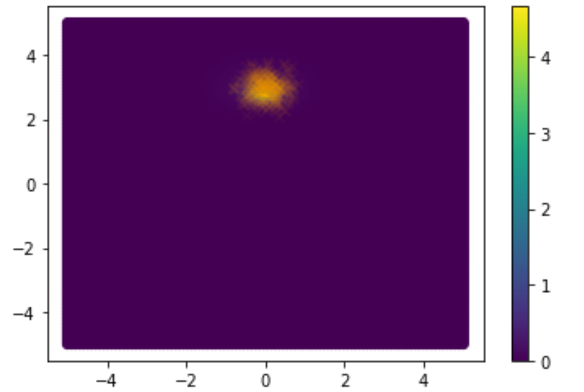

```
Epoch 750 -> loss: 2.57
```



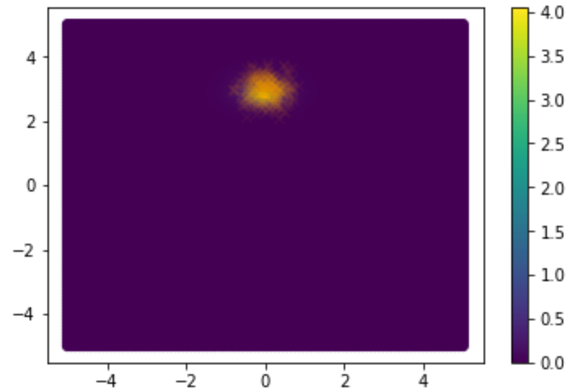
```
Epoch 1000 -> loss: 2.42
```

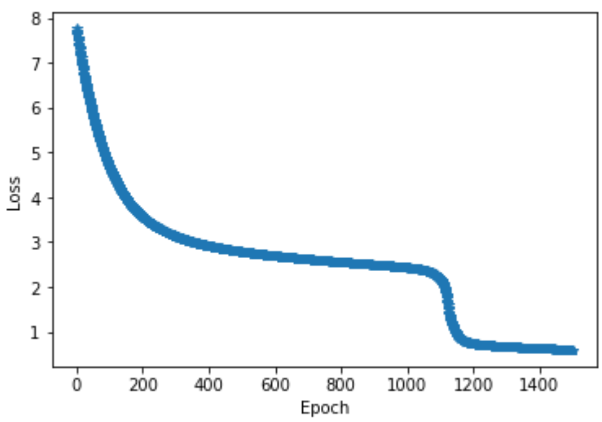


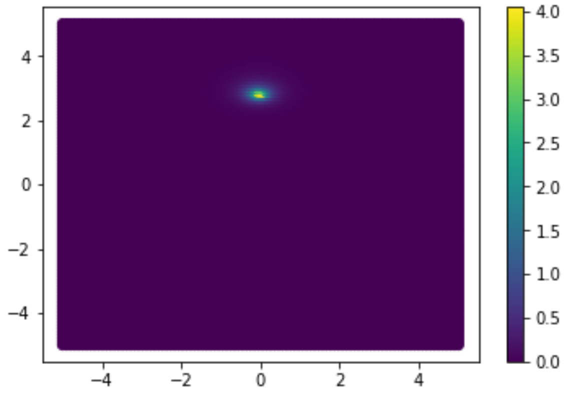

```
Epoch 1250 -> loss: 0.68
```




```
Epoch 1500 -> loss: 0.59
```



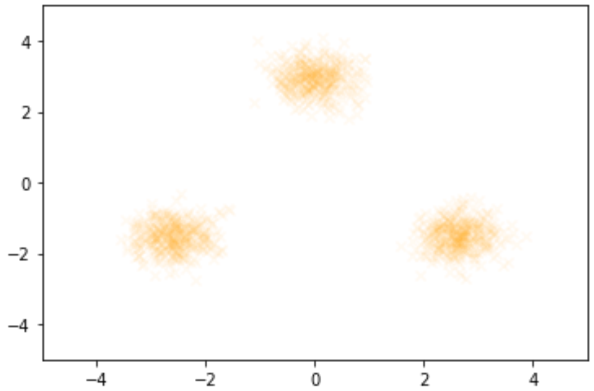




Dataset 2: 3 Gaussians

The second dataset is composed of 3 gaussians with means on a circle.

In [10]:
```
dataset_2 = CircleGaussiansDataset(n_gaussians=3, n_samples=400, variance=.
4)
plt.scatter(dataset_2.X[:,0], dataset_2.X[:,1], alpha=.05, marker='x', c='or
ange')
plt.xlim(-5, 5)
plt.ylim(-5, 5)
plt.show()
```

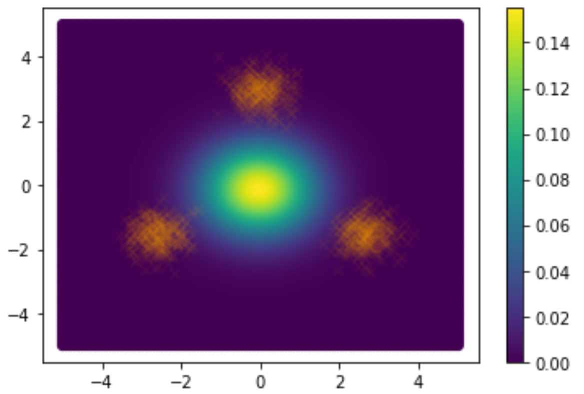


Affine flow

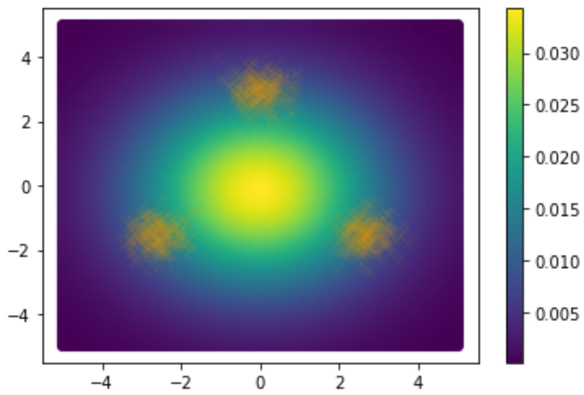
In [11]:
```
transforms = [Affine().to(device)]
model = StackedFlows(transforms, base_dist='Normal').to(device)
model, losses = train(model, dataset_2, max_epochs=500)

# Plots
plt.plot(losses, marker='*')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.show()
plot_density(model, [], device=device)
plot_samples(model)
```

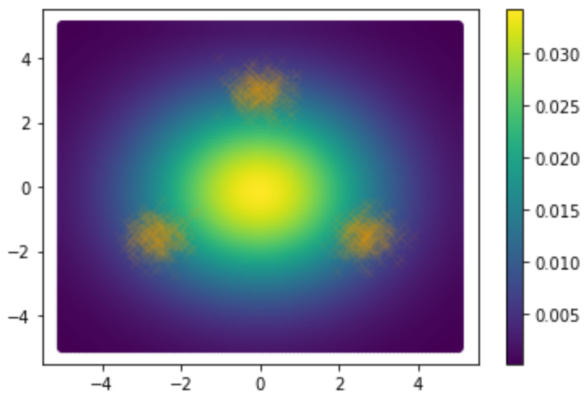
```
Epoch 0 -> loss: 6.45
```

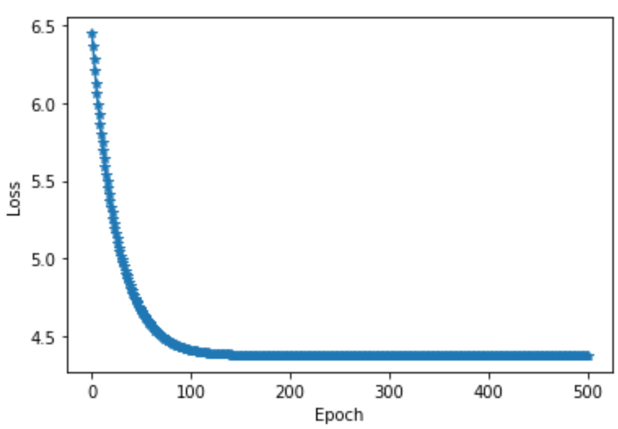


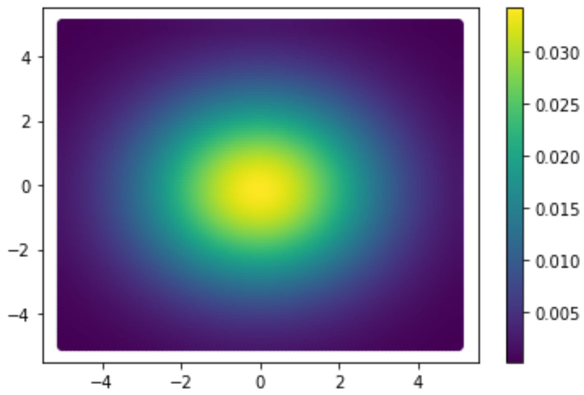
```
Epoch 250 -> loss: 4.37
```

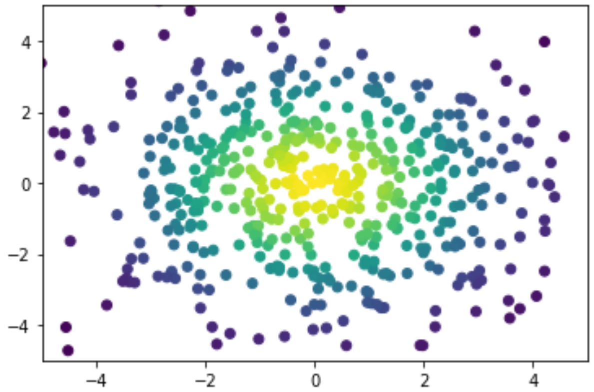


```
Epoch 500 -> loss: 4.37
```









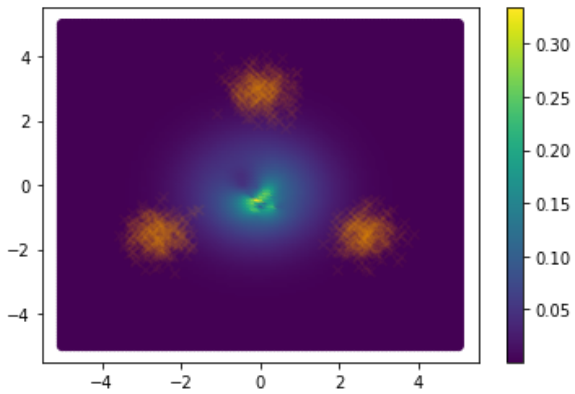
Affine flow should fail here.

Radial flow (32 layers)

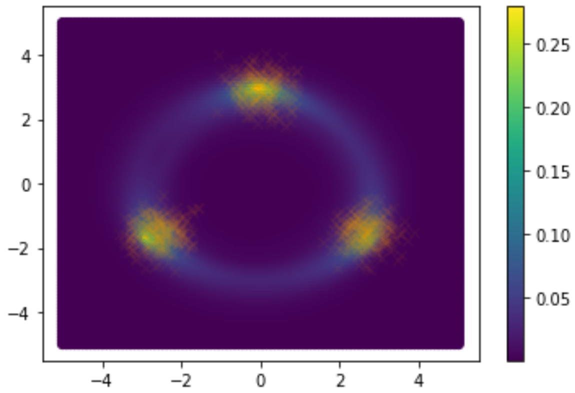
In [12]:
```
transforms = [InverseFlow(Radial()).to(device) for _ in range(32)]
model = StackedFlows(transforms, base_dist='Normal').to(device)
model, losses = train(model, dataset_2, max_epochs=500, frequency=100)

# Plots
plt.plot(losses, marker='*')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.show()
plot_density(model, [], device=device)
```

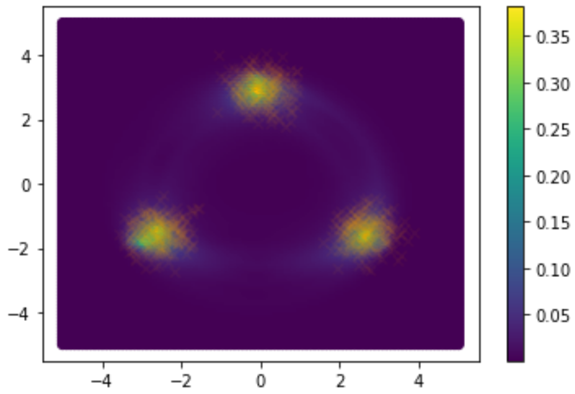
```
Epoch 0 -> loss: 6.08
```



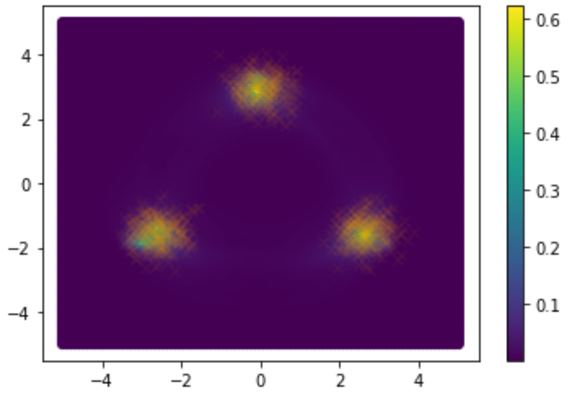
```
Epoch 100 -> loss: 2.83
```



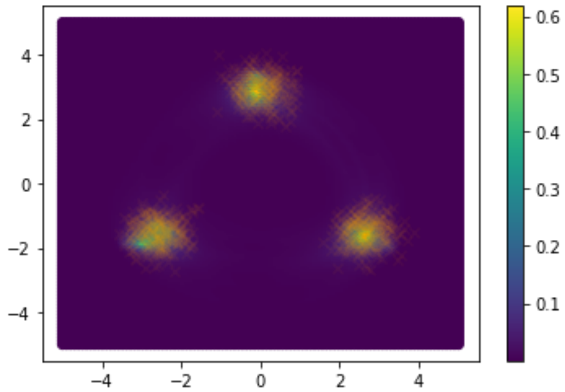
```
Epoch 200 -> loss: 2.46
```



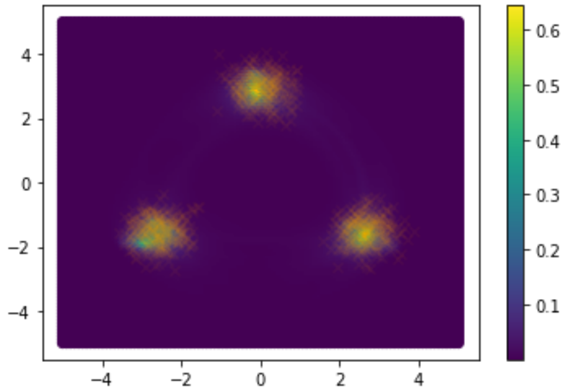
```
Epoch 300 -> loss: 2.34
```

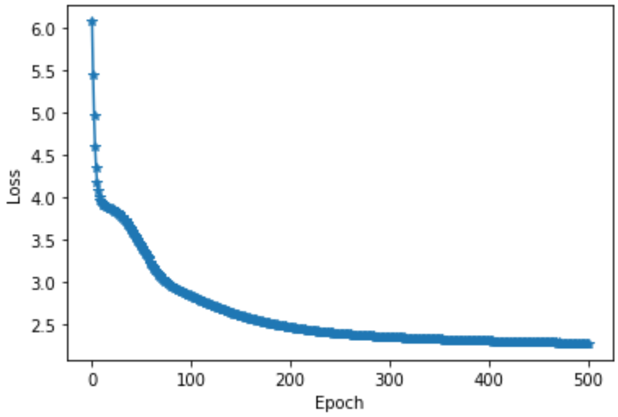


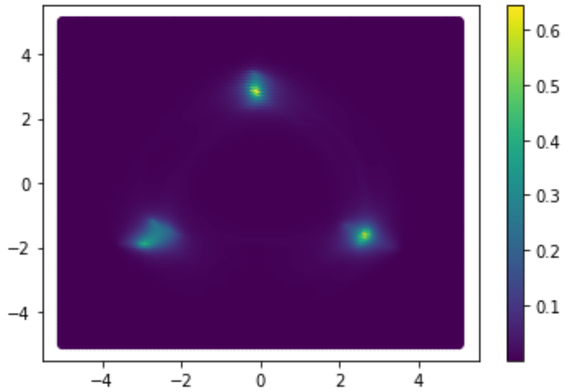
```
Epoch 400 -> loss: 2.30
```




```
Epoch 500 -> loss: 2.27
```





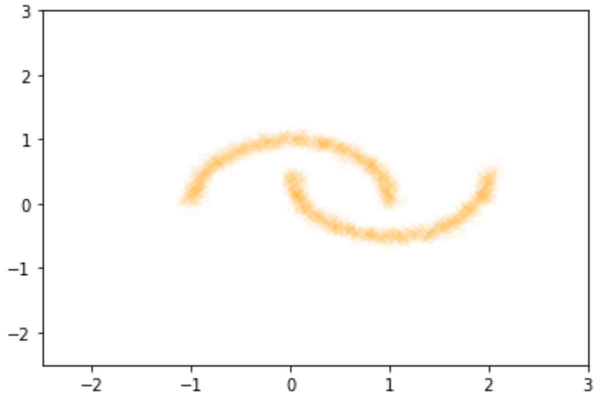


Using 32 layers of radial flow should lead to a good reasonable fit of the data after 500 epochs. Traning with more layers and for more epochs would improve the density estimation but would take more time. You might have to run the training multiple times to learn the three Gaussians (it learns sometimes only two Gaussians).

Dataset 3: 2 Moons

The third dataset is composed of 2 moons. Affine flow should fail again. With more layers, Radial flow should work.

In [13]:
```
dataset_3 = MoonsDataset()
plt.scatter(dataset_3.X[:,0], dataset_3.X[:,1], alpha=.05, marker='x', c='or
ange')
plt.xlim(-2.5, 3)
plt.ylim(-2.5, 3)
plt.show()
```

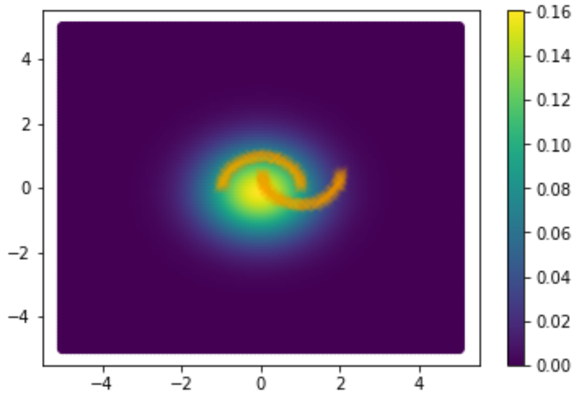


Affine flow

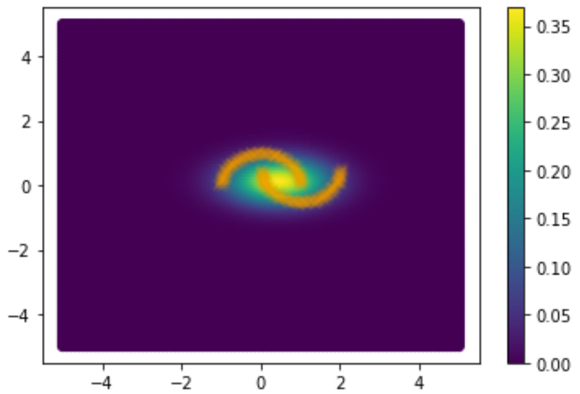
In [14]:
```
transforms = [Affine().to(device)]
model = StackedFlows(transforms, base_dist='Normal').to(device)
model, losses = train(model, dataset_3, max_epochs=500)

# Plots
plt.plot(losses, marker='*')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.show()
plot_density(model, [], device=device)
plot_samples(model)
```

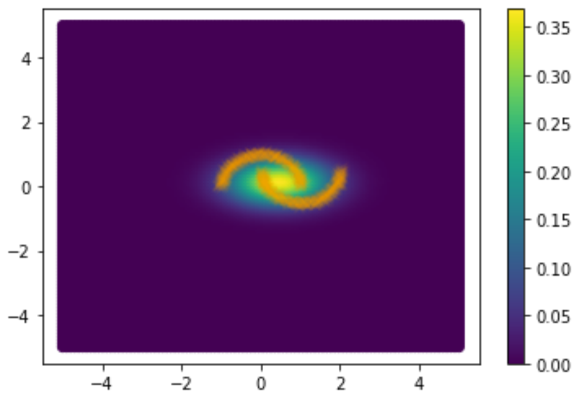
```
Epoch 0 -> loss: 2.49
```

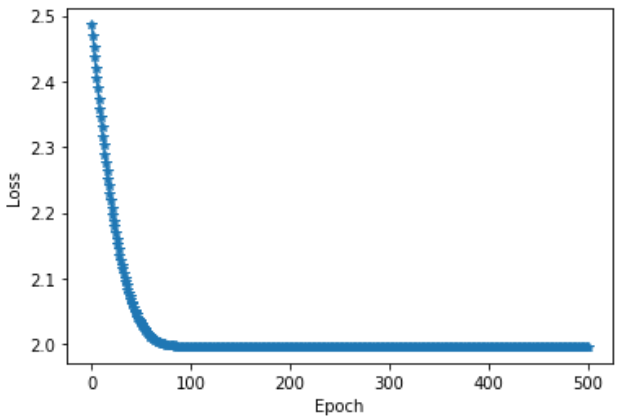


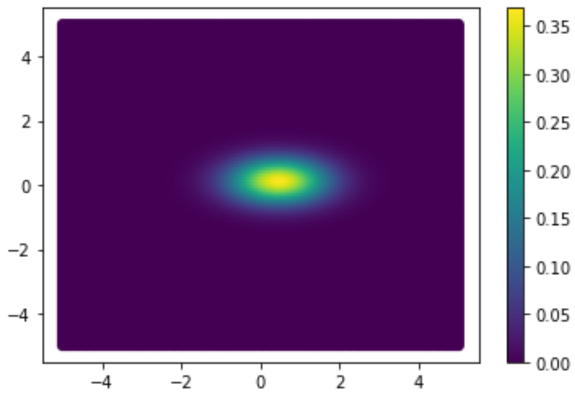
```
Epoch 250 -> loss: 2.00
```

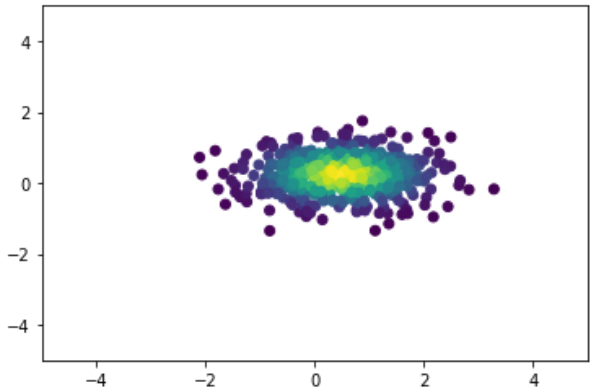


```
Epoch 500 -> loss: 2.00
```









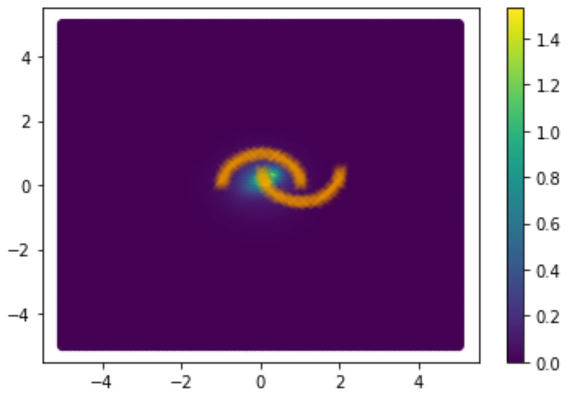
Affine flow should fail here.

Radial flow (32 layers)

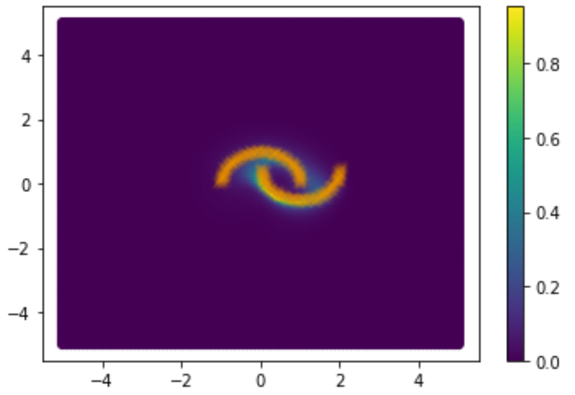
In [15]:
```
transforms = [InverseFlow(Radial()).to(device) for _ in range(32)]
model = StackedFlows(transforms, base_dist='Normal').to(device)
model, losses = train(model, dataset_3, max_epochs=500, frequency=100)

# Plots
plt.plot(losses, marker='*')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.show()
plot_density(model, [], mesh_size=3, device=device)
```

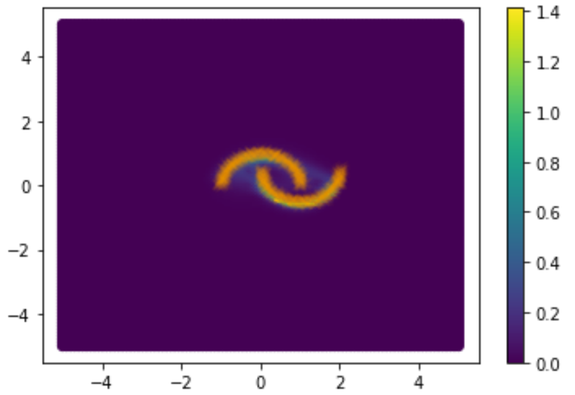
```
Epoch 0 -> loss: 2.82
```



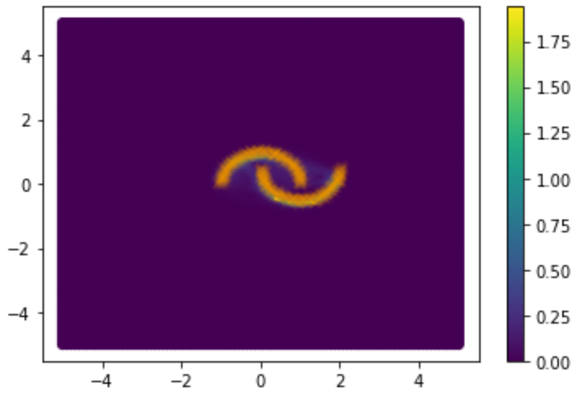
```
Epoch 100 -> loss: 1.57
```



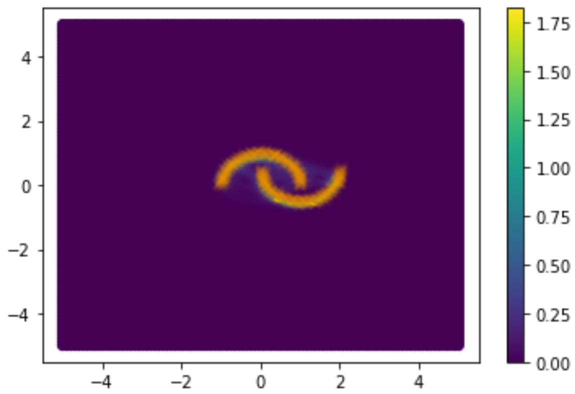
```
Epoch 200 -> loss: 1.15
```



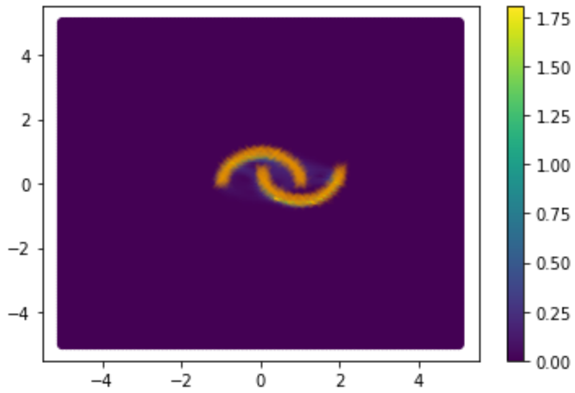
```
Epoch 300 -> loss: 0.97
```

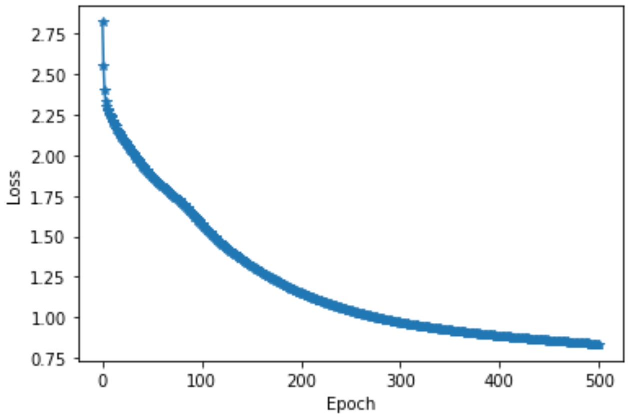


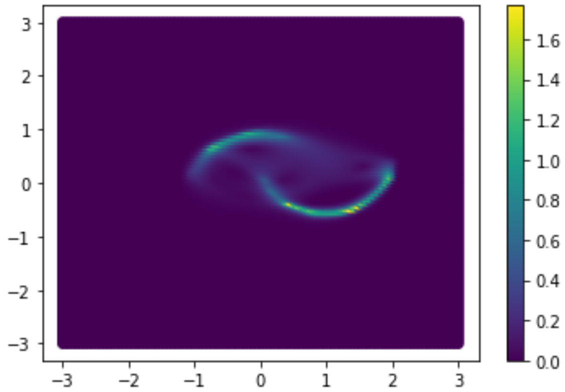
```
Epoch 400 -> loss: 0.88
```



```
Epoch 500 -> loss: 0.83
```







Using 32 layers of radial flow should lead to a good reasonable fit of the data after 500 epochs. Traning with more layers and for more epochs would improve the density estimation but would take more time.