planar_cnf

December 1, 2023

1 Continuous Normalizing Flow implementation

30/11/23, Hugo Gangloff

This notebook implements a Normalizing Flow with planar layers and a Continuous Normalizing Flow with planar layers. The code relies on: - JAX as a differentiable and optimized numpy wrapper - equinox as a multifunction deep learning library - flowjax for the normalizing flows - diffrax for the differentiable ODE solver

Our goal is to estimate the distribution of a toy dataset *two moons*. This reimplements the experiments of Fig. 4 and Fig. 5 of Neural ODE paper

1.1 A Normalizing Flow with planar layers

[1]:
```
import jax
import jax.numpy as jnp
from flowjax.flows import PlanarFlow, CouplingFlow
from flowjax.distributions import Affine
from flowjax.distributions import Normal, StandardNormal
from flowjax.train import fit_to_data
import optax
import matplotlib.pyplot as plt
import numpy as np
from flowjax.tasks import two_moons
```

[2]:
```
key = jax.random.PRNGKey(0)
key, subkey = jax.random.split(key, 2)
```

We define the planar flow with flowjax. This is a composition of K planar layers. `invert=True` means that we in fact learn T^{-1} as $z_0 = T^{-1}(z_K)$, this is the way we need to evaluate the log density in the maximum likelihood approach.

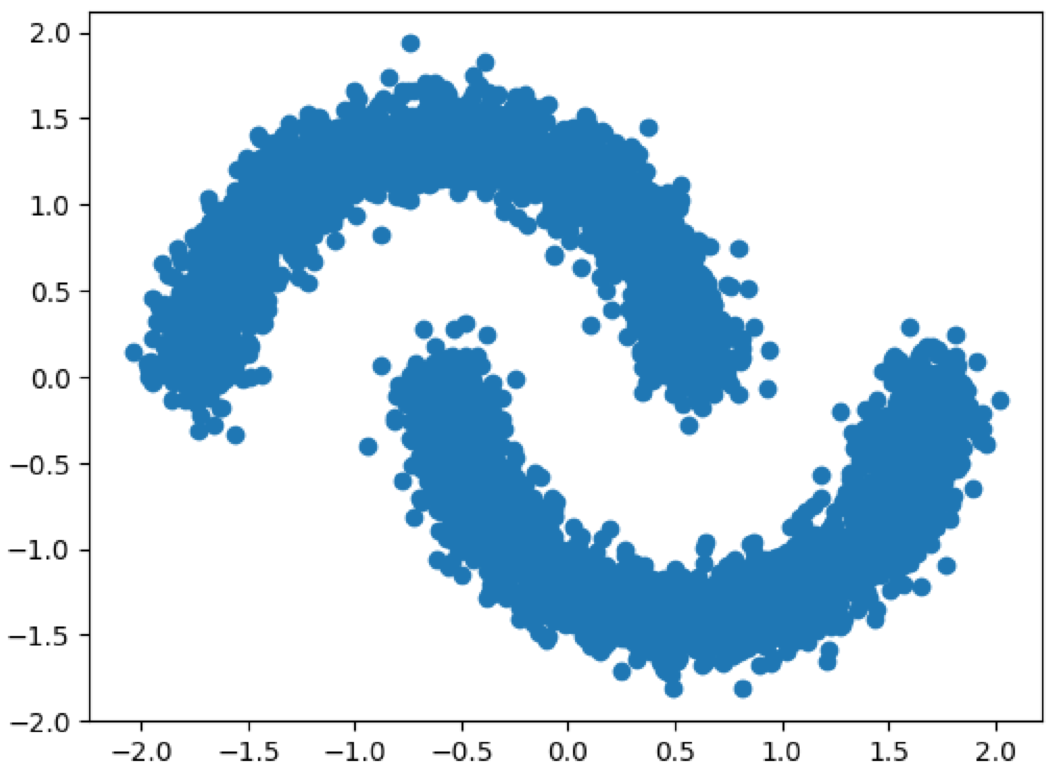
[3]:
```
prior_z = StandardNormal(shape=(2,))

K = 20

nf = PlanarFlow(
    key=subkey,
    base_dist=prior_z,
```

```
    flow_layers=K,
    invert=True
)
```

[4]:
```
batch_size = 5000
points = two_moons(subkey, batch_size)
points = (points - points.mean(axis=0)) / points.std(axis=0)  # Standardize
```

Have a look at thre original points

[5]:
```
plt.scatter(points[:, 0], points[:, 1])
plt.show()
```

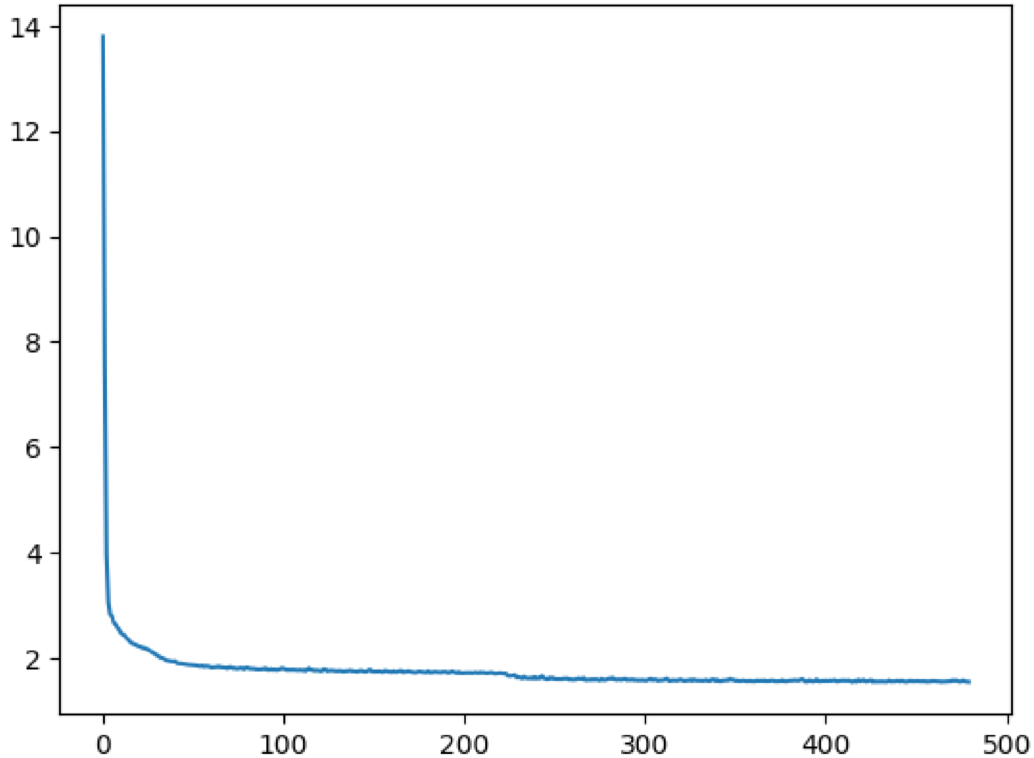


We use the function provided by flowjax for maximum likelihood learning.

[6]:
```
key, subkey = jax.random.split(key)
nf, losses = fit_to_data(subkey, nf, points, learning_rate=1e-2,␣
↪max_epochs=1000, max_patience=100)
```

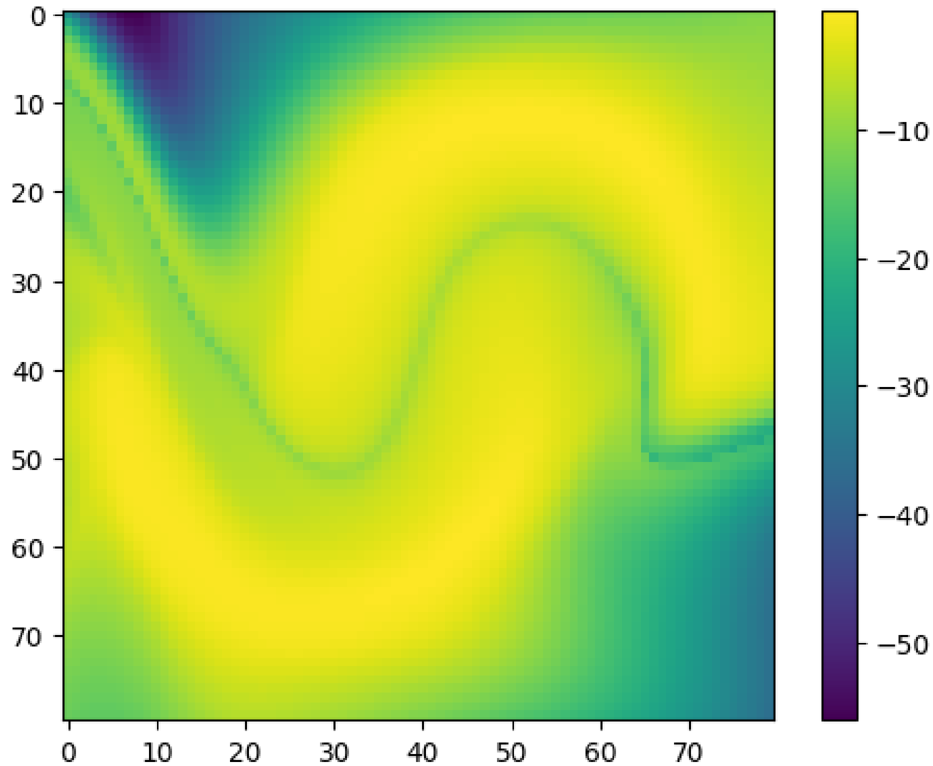
```
48%| | 479/1000 [02:10<02:21,  3.67it/s, train=1.5511765, val=1.6230481 (Max pa
```

[7]:
```
plt.plot(losses["train"])
plt.show()
```



[8]:
```
x = y = jnp.arange(-2, 2, 0.05)
Y, X = jnp.meshgrid(y, x)
xy = jnp.stack([jnp.ravel(Y), jnp.ravel(X)], axis=-1)
density_map = nf.log_prob(xy).reshape(X.shape)
```

[9]:
```
plt.imshow(density_map)
plt.colorbar()
plt.show()
```



Unfortunately we cannot get an explicit expression for the invert of the planar layer to get $z_K = T(z_0)$. Therefore we cannot draw samples from the trained model.

1.2 A Continuous Normalizing Flow with planar layer

This example is based on diverse piece of codes, notably this tutorial on CNF in diffrax and the CNF code accompanying the original Neural ODE article.

We define our planar CNF class built over equinox and flowjax.

[10]:
```
from flowjax.bijections import Planar
from flowjax.bijections import Bijection
from flowjax.distributions import Distribution
import diffrax
import equinox as eqx
from jax import Array
from typing import List, Callable
from functools import partial

class PlanarCNF(eqx.Module):
    prior_z0: Distribution
    t0: float
    t1: float
```

```
    dt0: float
    key: Array
    dim: int
    planars: List
    hidden_unit: int

    """
    dim is D in the paper
    hidden_dim is M
    """
    def __init__(
        self,
        key,
        dim,
        hidden_unit=1,
        **kwargs
    ):
        super().__init__(**kwargs)
        key, subkey = jax.random.split(key)
        keys_init_planar = jax.random.split(subkey, hidden_unit * 2)
        self.planars = []
        for i in range(hidden_unit):
            self.planars.append(
                (
                    eqx.nn.MLP(
                        in_size=1,
                        out_size=1,
                        width_size=40,
                        depth=3,
                        key=keys_init_planar[2 * i]
                    ),
                    Planar(
                        key=keys_init_planar[2 * i + 1],
                        dim=dim
                    )
                )
            )
        self.hidden_unit = hidden_unit
        self.prior_z0 = StandardNormal(shape=(dim,))
        self.t0 = 0.0
        self.t1 = 1.0

        self.dt0 = 0.05
        self.key = key
        self.dim = dim

    def gating_mechanism(self, t, z, i):
```

```
        return self.planars[i][0](t[None]) * self.planars[i][1].transform(z)

    # custom transform function
    def transform(self, t, z):
        return jnp.sum(jnp.stack([self.gating_mechanism(t, z, i) for i in␣
↪range(self.hidden_unit)], axis=-1), axis=-1)

    def transform_density(self, t, z, args):
        # Use approx formula
        return jnp.sum(jnp.stack([self.trace_Jac_g_approx(lambda z:partial(self.
↪gating_mechanism, t=t, i=i)(z=z), z, args) for i in range(self.
↪hidden_unit)]))
        # use the exact formula
        #return jnp.sum(jnp.stack([self.trace_Jac_g_exact(lambda z:partial(self.
↪gating_mechanism, t=t, i=i)(z=z), z, args) for i in range(self.
↪hidden_unit)]))

    def trace_Jac_g_approx(self, g, z, args):
        (eps,) = args
        # forward mode of Hutchison's trace estimator
        _, dg_dz = jax.jvp(g, (z,), (eps,))
        return jnp.sum(eps * dg_dz)
        # backward mode of Hutchinson's trace estimator
        # _, vjp_fn = jax.vjp(g, z)
        # (eps_dgdy,) = vjp_fn(eps)
        # return jnp.sum(eps_dgdy * eps)

    def trace_Jac_g_exact(self, g, z, args):
        assert self.dim == 2
        _, dg1_dz1 = jax.jvp(lambda z: g(z)[0], (z,), (jnp.array([1.0, 0.0]),))
        _, dg2_dz2 = jax.jvp(lambda z: g(z)[1], (z,), (jnp.array([0.0, 1.0]),))
        return dg1_dz1 + dg2_dz2
        # a naive way to compute the trace would be to compute the full␣
↪Jacobian matrix
        # return jnp.trace(jax.jacrev(g)(z))

    def rhs_term_augmented(self, t, z, args):
        z, _ = z # we have two states in z
        return (self.transform(t, z),
                self.transform_density(t, z, args)
               )

    # Backward in time to train the CNF
    def eval_log_density(self, x, key):
        term = diffrax.ODETerm(self.rhs_term_augmented)
        solver = diffrax.Tsit5()
```

```
        eps = jax.random.normal(key, x.shape) # used in Hutchinson trace␣
↪approximation
        diff_log_likelihood_t1 = 0.0
        sol = diffrax.diffeqsolve(
            term, solver, self.t1, self.t0, -self.dt0, (x,␣
↪diff_log_likelihood_t1), (eps,)
        )
        (z_t0,), (diff_log_likelihood_t0,) = sol.ys # Note that we only get the␣
↪last value but could get some other (see diffrax.Solution)
        log_likelihood_x = self.prior_z0.log_prob(z_t0) + diff_log_likelihood_t0
        return log_likelihood_x

    # Forward in time to sample from the CNF
    def sample(self, key):
        z_t0 = jax.random.normal(key, (2,))

        # The RHS term is just the function g now that is has been learnt
        term = diffrax.ODETerm(lambda t, z, args: self.transform(t, z))
        solver = diffrax.Tsit5()
        sol = diffrax.diffeqsolve(term, solver, self.t0, self.t1, self.dt0,␣
↪z_t0, ())
        (x,) = sol.ys # x = z_t1
        return x
```

[11]:
```
def loss(params, x, key):
    """ Train with maximum likelihood """
    planar_cnf = eqx.combine(params, static)

    v_train = jax.vmap(planar_cnf.eval_log_density, (0, 0))

    log_likelihood_x = jnp.mean(v_train(x, key))
    # return the minus because we want to maximize

    return -log_likelihood_x
```

[12]:
```
key, subkey = jax.random.split(key)
dim = 2
hidden_unit = 20
planar_cnf = PlanarCNF(subkey, dim, hidden_unit)
```

1.2.1 Train a planar CNF

[13]:
```
import optax
from jax_tqdm import scan_tqdm

n_iter = 2000
```

```
batch_size = 500
learning_rate = 1e-2

optimizer = optax.adam(learning_rate)
init_params, static = eqx.partition(planar_cnf, eqx.is_inexact_array)
opt_state = optimizer.init(init_params)
```

[14]:
```
params = init_params
```

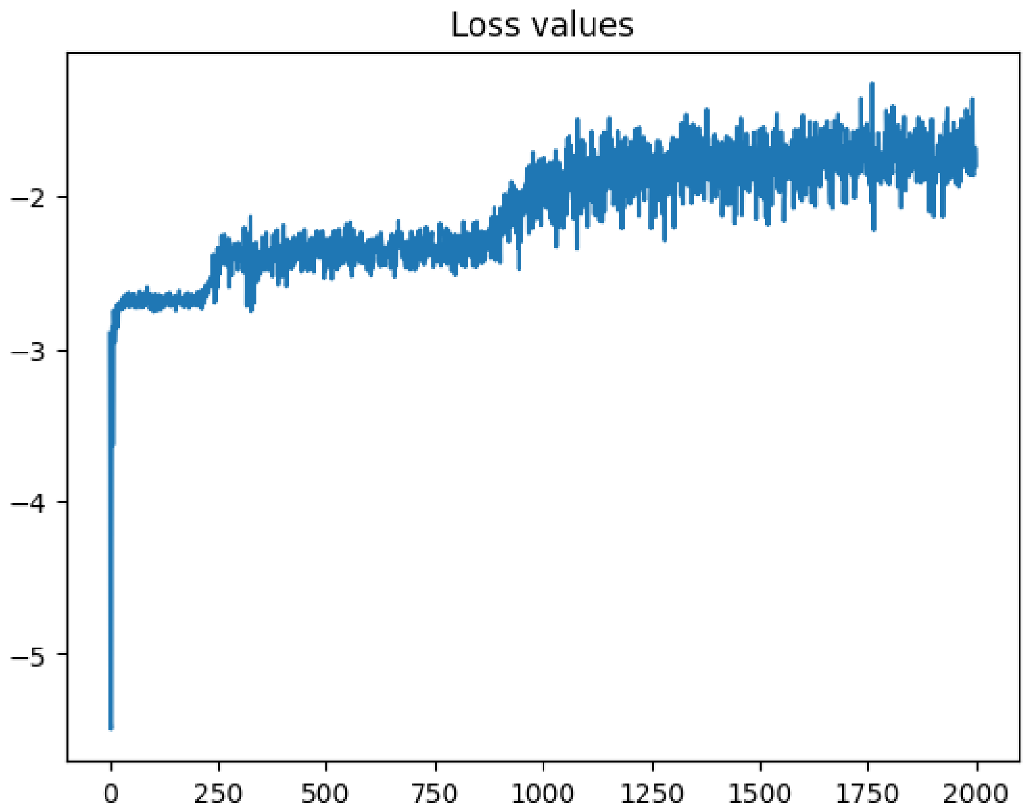
[15]:
```
@scan_tqdm(n_iter)
def scan_fun(carry, i):
    key, params, opt_state = carry
    key, subkey = jax.random.split(key)
    x = two_moons(subkey, batch_size) # x = z_t1
    x = (x - x.mean(axis=0)) / x.std(axis=0)

    key, subkey = jax.random.split(key)
    loss_val, grads = jax.value_and_grad(loss)(params, x, jax.random.
 ↪split(subkey, x.shape[0]))
    #jax.debug.print("{x}",x=(loss))

    updates, opt_state = optimizer.update(grads, opt_state, params)
    params = optax.apply_updates(params, updates)

    return (key, params, opt_state), loss_val
```

[16]:
```
carry, loss_values = jax.lax.scan(
    scan_fun,
    (key, params, opt_state),
    jnp.arange(n_iter)
)
```

```
  0%|          | 0/2000 [00:00<?, ?it/s]
```

[17]:
```
params = carry[1]
```

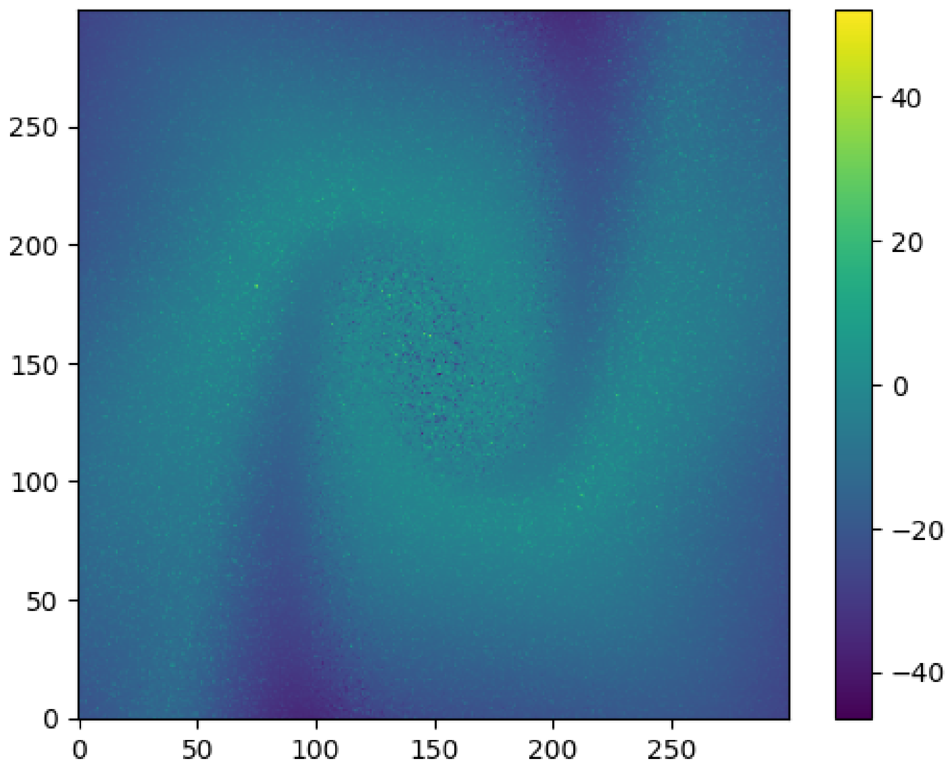
[18]:
```
plt.plot(-loss_values)
plt.title("Loss values")
plt.show()
```



Let's plot the learnt density

[19]:
```
planar_cnf = eqx.combine(params, static)

x = y = jnp.arange(-3, 3, 0.02)
Y, X = jnp.meshgrid(y, x)
xy = jnp.stack([jnp.ravel(Y), jnp.ravel(X)], axis=-1)
keys = jax.random.split(key, xy.shape[0] + 1)
key = keys[0]
v_density = jax.vmap(planar_cnf.eval_log_density, (0, 0))
density_map = v_density(xy, keys[1:]).reshape(X.shape)
```

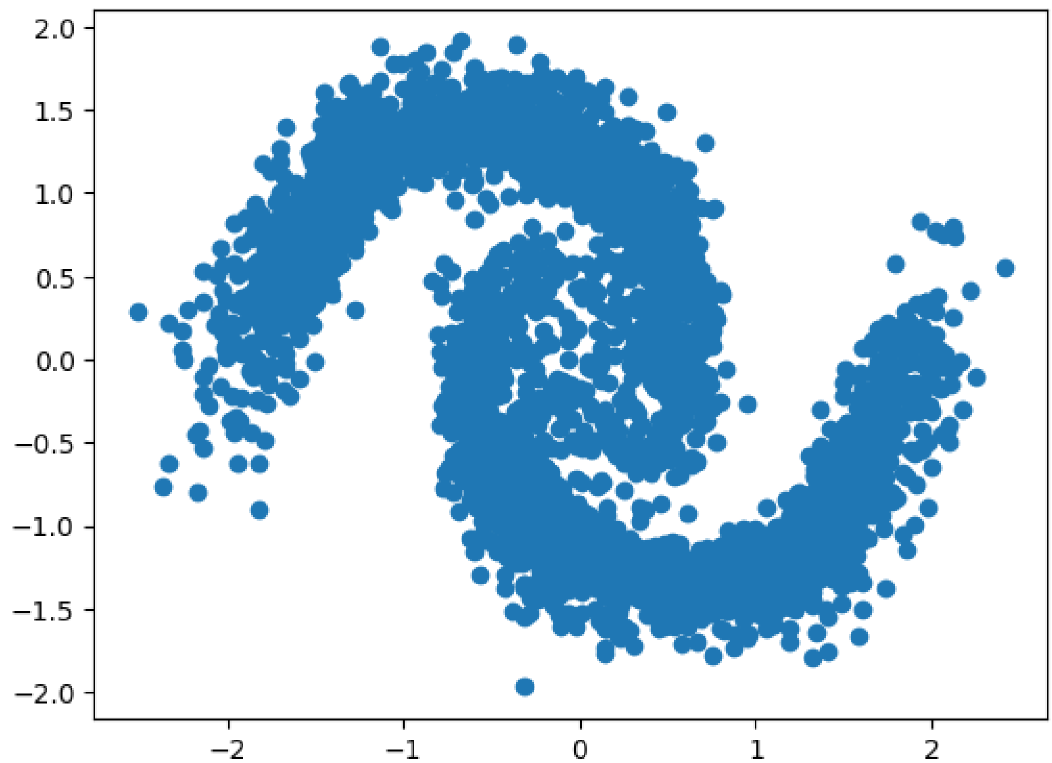
[20]:
```
plt.imshow(density_map, origin='lower')
plt.colorbar()
plt.show()
```



Let's sample from the trained model

[21]:
```
planar_cnf = eqx.combine(params, static)
num_samples = 5000
key, subkey = jax.random.split(key)
sample_key = jax.random.split(key, num_samples)
samples = jax.vmap(planar_cnf.sample, (0))(sample_key)
plt.scatter(samples[:, 0], samples[:, 1])
```

[21]:
```
<matplotlib.collections.PathCollection at 0x7fc42e4f2890>
```


2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2
−1
0
1
2