Introduction

In this notebook we'll see what are Normalizing Flows exactly and play a bit with a standard implementation. Let's import what we need. We need to have `pytorch` and `nflows`

```
!pip install -q nflows
```

```
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 45.8/45.8 kB 1.1 MB/s eta 0:00:00
  Preparing metadata (setup.py) ... done
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 363.4/363.4 MB 4.6 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 13.8/13.8 MB 39.8 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 24.6/24.6 MB 33.7 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 883.7/883.7 kB 27.6 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 664.8/664.8 MB 2.8 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 211.5/211.5 MB 6.2 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 56.3/56.3 MB 13.4 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 127.9/127.9 MB 8.6 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 207.5/207.5 MB 6.8 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 21.1/21.1 MB 42.4 MB/s eta 0:00:00
  Building wheel for nflows (setup.py) ... done
```

```
# standard python stuff
import os
import sys
import numpy as np
import scipy.stats as st
import matplotlib.pyplot as plt

# stuff for torch+nflows
import torch
from torch import nn
from tqdm import tqdm
from torch.nn.modules import Module

# optimizer from torch
from torch import optim

# base Flow to construct model
from nflows.flows.base import Flow

# base distribution to use
from nflows.distributions.normal import StandardNormal

# the MADE coupling layer
from nflows.transforms.autoregressive import MaskedAffineAutoregressiveTransform

# this adds a RandomPermutation to add variance between layers
from nflows.transforms.permutations import RandomPermutation
```

```
# this will combine modules
from nflows.transforms.base import CompositeTransform
```

Transformation rules of probability density functions

The transformation rule of pdfs for a change of variable $x = f(z)$ is

$p_X(x) = p_Z\left(f^{-1}(x)\right) \left|\det \nabla_x f^{-1}(x)\right|$

The determinant of the Jacobian can be rewritten in terms of f for ease of computation as

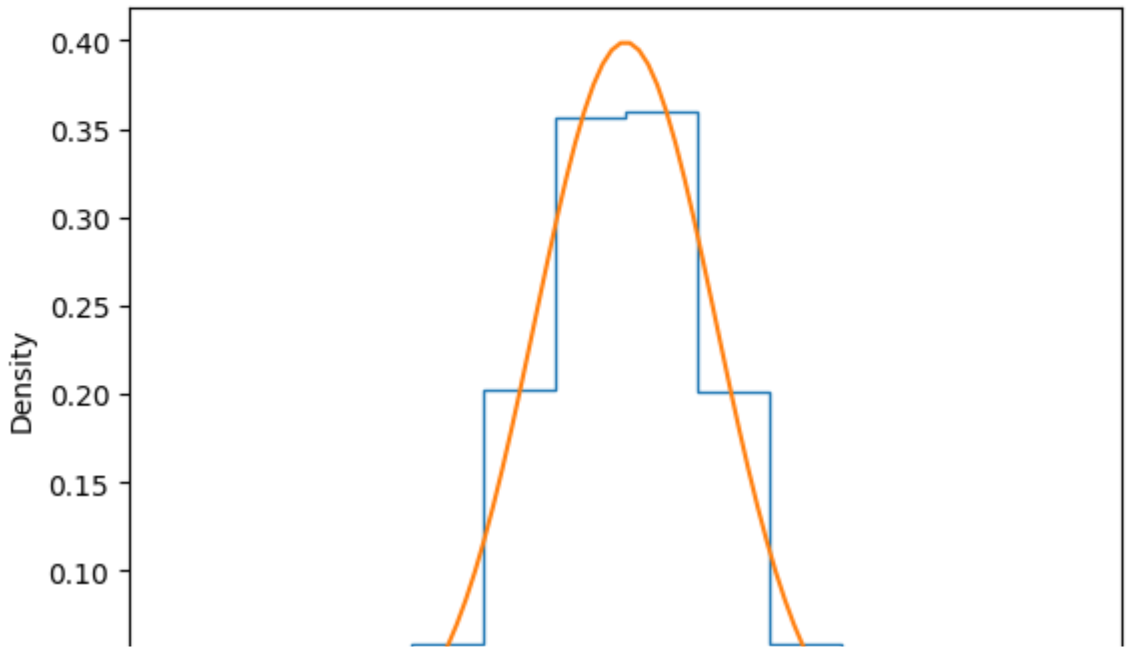
$p_X(x) = p_Z\left(f^{-1}(x)\right) \left|\det \nabla_x f^{-1}(x)\right| = p_Z\left(f^{-1}(x)\right) \left|\det \nabla_z f\left(f^{-1}(x)\right)\right|^{-1}$

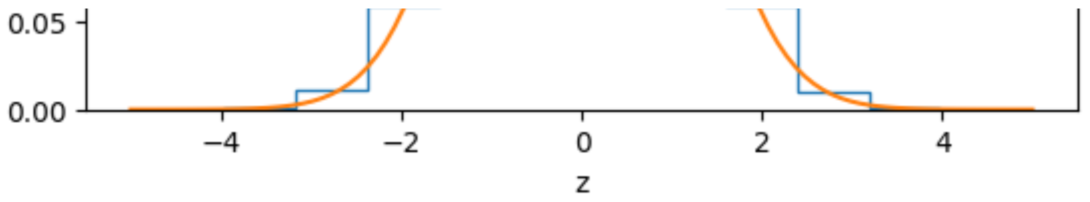
Example:

Show how you can transform a normally distributed $x \sim \mathcal{N}(0,1)$ to a normally distributed variable $x \sim \mathcal{N}(\mu, \sigma)$ through the change of variables $x = \mu + \sigma z$ by completing this code

```
### an example of this
N = 10000
pdf_z = st.norm(loc=0, scale=1)
Z = pdf_z.rvs(N)
plt.hist(Z, histtype="step", density=True)
ztoplot = np.linspace(-5, 5, 100)  # dummy variables for plotting the pdf
pdf_vals_z = pdf_z.pdf(ztoplot)  # evalute pdf for plotting
plt.plot(ztoplot, pdf_vals_z)
plt.xlabel("z")
plt.ylabel("Density")
```

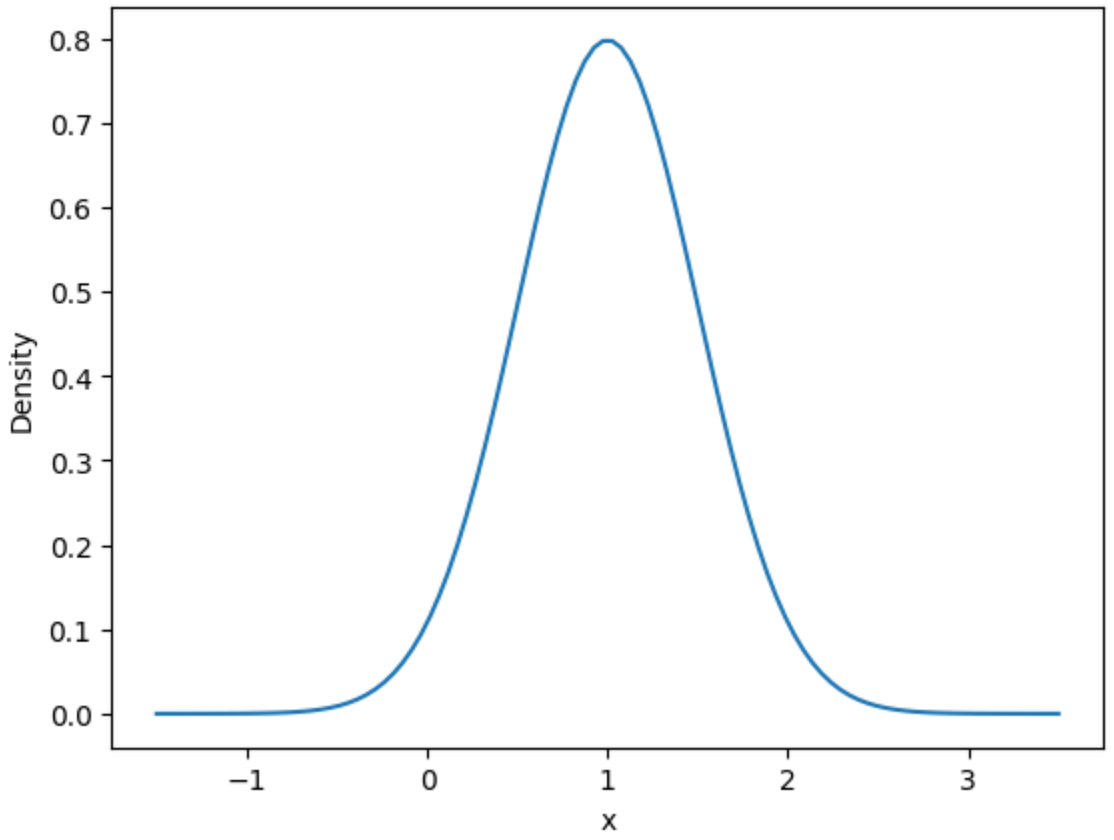
```
Text(0, 0.5, 'Density')
```





```
mu = 1.0  # arbitrary values
sigma = 0.5  # arbitrary values
X = mu + sigma * Z
xtoplot = mu + sigma * ztoplot  # dummy variables for plotting the pdf
gradient_xtoplot_over_z = 1 / sigma * np.ones(len(xtoplot))  # compute the gradient
pdf_vals_x = (
    pdf_z.pdf(ztoplot) * gradient_xtoplot_over_z
)  # evaluate the new pdf using the old pdf + jacobian
plt.plot(xtoplot, pdf_vals_x)
plt.xlabel("x")
plt.ylabel("Density")
```

```
Text(0, 0.5, 'Density')
```



Normalizing Flows

At its essence, Normalizing Flows are bijective functions that map a sample space to a new space where data is distributed however we chose it to. That is, if we have data $x \sim p_X$, we want to learn an invertible function $x = f(z, \theta)$ such that z follows an base distribution easy to sample from and to evaluate. The most common choice is a normal distribution $z \sim p_Z \equiv \mathcal{N}(0, 1)$.

f will be a learnable neural network with parameters θ and an easy to compute gradient. The loss function which θ needs to minimize is nothing more than the negative Log Likelihood obtained using the transformation rule of pdfs

$$\mathcal{L} = -\sum_{x\in\mathcal{D}} \text{Ln}\, p_X(x) = -\sum_{x\in\mathcal{D}} \text{Ln}\left[p_Z(f^{-1}(x,\theta))|\det\nabla_z f|^{-1}\right]$$

$$\mathcal{L} = \sum_{x\in\mathcal{D}} \left(-\text{Ln}\left[p_Z(f^{-1}(x,\theta))\right] + \text{Ln}\left[|\det\nabla_z f|\right]\right)$$

And assuming a standard normal distribution

$$\mathcal{L} = \sum_{x\in\mathcal{D}} \left(-\text{Ln}\,\mathcal{N}\left(f^{-1}(x,\theta);0,1\right) + \text{Ln}\left[|\det\nabla_z f|\right]\right)$$

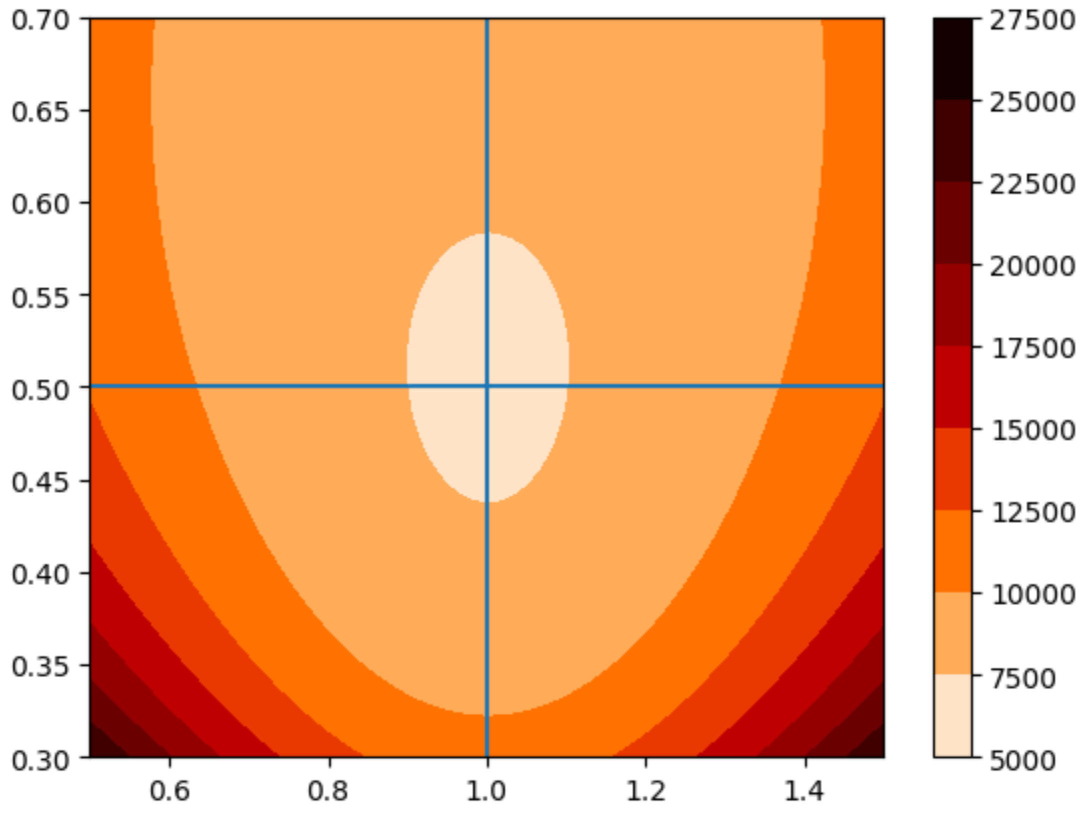
The trick is how to chose a learnable f with easy gradient (which is not a problem using the gradient chain rule with standard NNs + backpropagation) but also easily invertable to go back and forth from x to z.

Example

In the previous, very simplified example, we know that a good choice of $f(z,\theta)$ is simply $f(z,\theta) = \theta_0 + \theta_1 z$ with inverse $f^{-1}(x,\theta) = (x-\theta_0)/\theta_1$ and jacobian $|\det\nabla_z f| = |\theta_1|$ (which does not depend on the evaluation on $z = (x-\theta_0)/\theta_1$. We can thus simply write the loss function and do a very naive grid minimization

```
def loss_function(X, theta0, theta1):
    return np.sum(
        -st.norm(loc=0, scale=1).logpdf((X - theta0) / theta1) + np.log(theta1)
    )


theta0vals = np.linspace(
    0.5, 1.5, 100
)  # substitute adequate range if you changed mu, sigma before
theta1vals = np.linspace(
    0.3, 0.7, 100
)  # substitute adequate range if you changed mu, sigma before
theta0vals_plot, theta1vals_plot = np.meshgrid(theta0vals, theta1vals)
# print(theta0vals_plot.shape,theta1vals_plot.shape)
loss_function_vals = np.zeros(theta0vals_plot.shape)
for ntheta1val, theta1val in enumerate(theta1vals):
    for ntheta0val, theta0val in enumerate(theta0vals):
        loss_function_vals[ntheta1val, ntheta0val] = loss_function(
            X, theta0val, theta1val
        )
plt.contourf(theta0vals, theta1vals, loss_function_vals, cmap="gist_heat_r")
plt.axhline(sigma)
plt.axvline(mu)
plt.colorbar()
```

```
<matplotlib.colorbar.Colorbar at 0x793b0734efd0>
```



```
theta0min, theta1min = (
    theta0vals_plot.flatten()[np.argmin(loss_function_vals)],
    theta1vals_plot.flatten()[np.argmin(loss_function_vals)],
)
print(theta0min, theta1min)
```

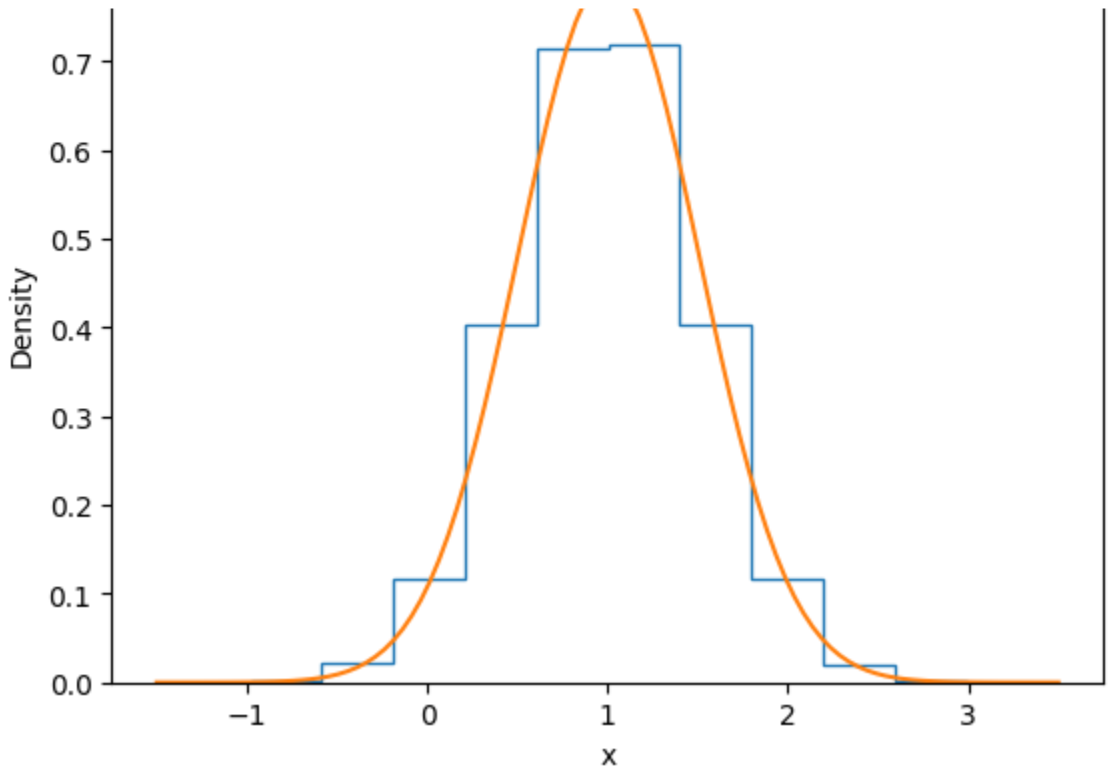
```
1.0050505050505052 0.502020202020202
```

```
loss_function(X, mu, sigma), loss_function(
    X, theta0min, theta1min
)  # why? likely overfitting
```

```
(np.float64(7291.13836573048), np.float64(7291.197514193597))
```

```
plt.hist(X, histtype="step", density=True)
# plt.plot(xtoplot,pdf_vals_x)
# now we use the min parameters explicitly with xtoplot
pdf_vals_x_bis = (
    st.norm(loc=0, scale=1).pdf((xtoplot - theta0min) / theta1min) / theta1min
)
plt.plot(xtoplot, pdf_vals_x_bis)
plt.xlabel("x")
plt.ylabel("Density")
```

```
Text(0, 0.5, 'Density')
```



Note that we **haven't seen the true Z during training**. The technique is aimed at learning $p_X(x)$. We did cheat by knowing that the simple parameterization was good enough.

Choice of f

There are many ways to do this, but the usual trick consists of concatenating several individual, simpler modules. That is

$z_1 = f_1(z)$

$z_i = f_i(z_{i-1})$ with $i = 2, \ldots, n-1$

$x = f_n(z_{n-1})$

and having each individual f_i module as a simple, invertible function whose parameters are Neural Networks. The choice of module aims to be easily invertible while allowing for as much expressivity as possible. That is, we want to be able to train while also capturing Jacobians as general as possible.

Let's talk about one possible and very popular choice, **Autoregressive flows**. Other strategies can be found in the referred paper. As always, each choice has advantages and disadvantages.

Masked Autoregressive flows

For inputs with dimension K, we parameterise $z_{i,k}$ as a function of the previous dimensions in z_{i-1}:

$z_{i,k} = f_i(z_{i-1,k}, g_{i,k}(z_{i-1,1:k-1}))$

That is, a function of the same input feature at the previous step $z_{i-1,k}$ and a **conditioner** g that takes the previous input and combines all the previous features $z_{i-1,1:k-1}$. When choosing a parametric, invertible form for f, g_k is a map from $z_{i-1,1:k-1}$. In the previous example, because there are no previous feature dimensions as the problem was 1D, g_1 was simply $g_1 = [\theta_0, \theta_1]$.

What is the advantage of this parameterization? That the Jacobian is triangular! One can show that

$\det \nabla f_i = \prod_{k=1}^{K} \frac{\partial z_{i,k}}{\partial z_{i-1,k}}$

The forward flow ($z_0 = z \rightarrow z_n = x$) itself can be obtained in one sweep by "masking" the features appropriately (that's why it's called Masked). "Masking" for a Neural Network means setting some features to zero before feeding the input to said Neural Network. In this case, masking allows for the conditioner to be a single set of Neural Networks, one for each parameter of f, each of them a function $\mathbb{R}^K \rightarrow \mathbb{R}$ that is evaluated for $x_{i,1:k-1}$ simply by masking or setting to zero the $k : K$ remaining entries.

The inverse function and jacobian computation ($z_0 = z \leftarrow z_n = x$) is more challenging computationally speaking because we need to use recursion and compute each entry one step at a time with $z_{i-1,1} = f_i^{-1}(z_{i,1})$ and $z_{i-1,k} = f_i^{-1}(z_{i,1}, g_{i,k}(z_{i-1,1:k-1}))$.

If we are more interested in the inverse (for problems such as a Variational Inference), we can use **Inverse Autoregressive Flows** (IAF) where the transformation is instead:

$z_{i,k} = f_i(z_{i-1,k}, g_{i,k}(z_{i,1:k-1}))$

(note the different index in the conditioner) and thus the recursion needs to be applied in the forward direction $z_0 = z \rightarrow z_n = x$.

The rule of thumb is: **MAF** for fast density estimation and **IAF** for fast sampling. The difference in speed is not always an issue, and as usual with ML rules of thumb can be disregarded for simple enough problems...

One can also use **Coupling flows** instead of **Autoregressive** ones. Coupling flows are equally fast in the forward and backward directions, so there is no difference between density estimation and sampling. However, they may be less flexible although very good performances can be obtained with Neural Spline Flows or Non Volume Preserving transformations. In the end, one should know what to play with and decide.

Coupling functions

In any case, once we have defined we are using MAFs or IAFs, we need to define the coupling function f_i. This defines the parameters obtained using the conditioners $g_{i,k}$ which are Neural Networks.

A very common couplign function is MADE where the update is using a Gaussian kernel and the conditioner models the mean and variance of the Gaussian:

$$z_{i,k} = z_{i-1,k} \exp \alpha_i(z_{i-1,1:k-1}) + \mu_i(z_{i-1,1:k-1})$$

α_i and μ_i are Neural Networks, which are evaluated on the first $k-1$ features by masking the remaining $K-(k-1)$ features.

Example

We'll use `nflows` to implement a MAF with MADE. There are many packages, with different implementations of different flows, so I recommend you always chose based on your problem. `nflows` is not perfect but it will suffice for the examples here.

```
# Reshape things for flows
X = X.reshape(-1, 1)
print(X.shape)
```

```
    (10000, 1)
```

```
# Define a base distribution.
base_distribution = StandardNormal(shape=[X.shape[1]])

# Define an invertible transformation.
num_layers = 5

transforms = []
for _ in range(num_layers):
    transforms.append(
        MaskedAffineAutoregressiveTransform(
            features=X.shape[1], hidden_features=4, num_blocks=2
        )
    )

    transforms.append(RandomPermutation(features=1))  # useless for 1 feature

transform = CompositeTransform(transforms)

# Combine into a flow.

flow = Flow(transform, base_distribution)
optimizer = optim.Adam(flow.parameters())
```

Before continuing, take some time to think about the relevant hyperparameters of the model. That is, what choices have I made. An "obvious" one is that for each initialized `MaskedAffineAutoregressiveTransform` there are two Neural Networks with `hidden_features` units per layer and `num_blocks` layers. You can look for more hyperparameters by going over the source code of the `nflows` package.

```
# transform inputs to torch tensors
X_torch = torch.tensor(X, dtype=torch.float32)
xtoplot_torch = torch.tensor(xtoplot.reshape(-1, 1), dtype=torch.float32)
```

The `nflows` package allows for straightforward evaluation of the likelihood with

```
flow.log_prob(inputs=X_torch)
```

```
    tensor([-2.4053, -2.4171, -2.4058,  ..., -2.4228, -2.4188, -2.4109],
           grad_fn=<AddBackward0>)
```

Let's see how things look before training the Flow. We can evaluate the likelihood using

```
pdf_vals_x = np.exp(flow.log_prob(xtoplot_torch).detach().numpy())
```

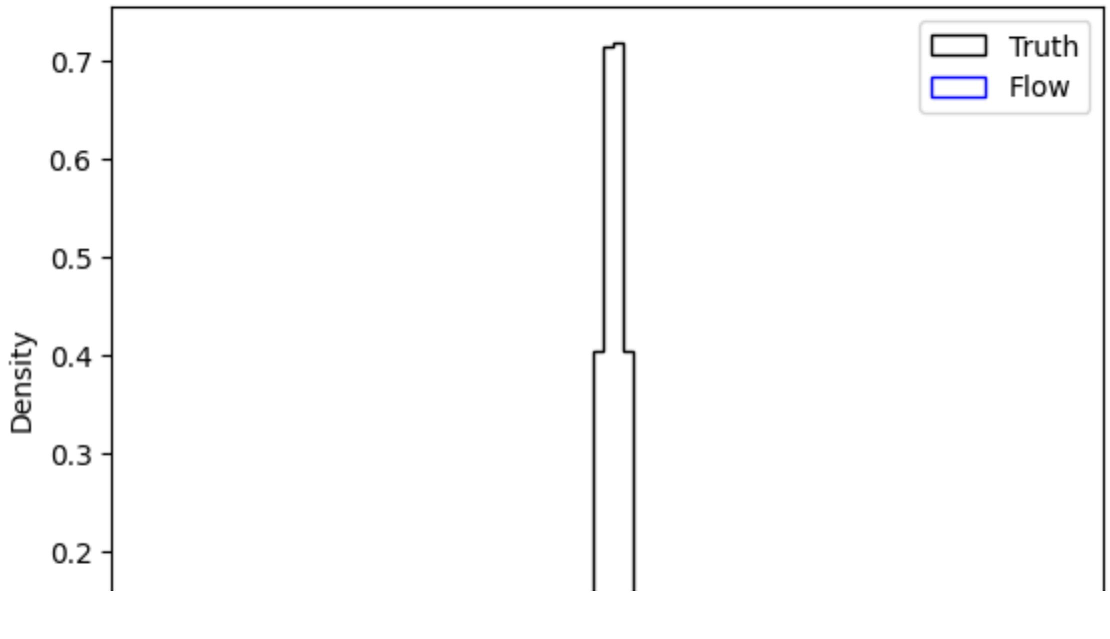
And we can sample using the learned likelihood using

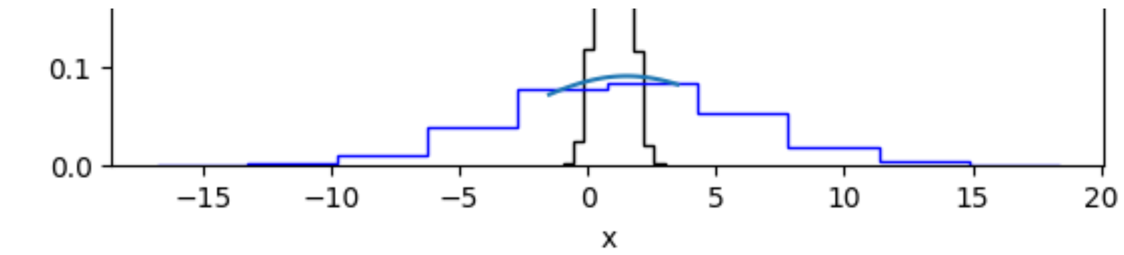
```
x_sample = flow.sample(len(X)).detach().numpy()
```

We can see how initially everything is random and thus a poor model

```
a, b, c = plt.hist(X, histtype="step", density=True, label="Truth", color="black")
plt.hist(x_sample, histtype="step", density=True, label="Flow", color="blue")
plt.legend(loc="upper right")
plt.plot(xtoplot, pdf_vals_x)
plt.xlabel("x")
plt.ylabel("Density")
```

```
    Text(0, 0.5, 'Density')
```





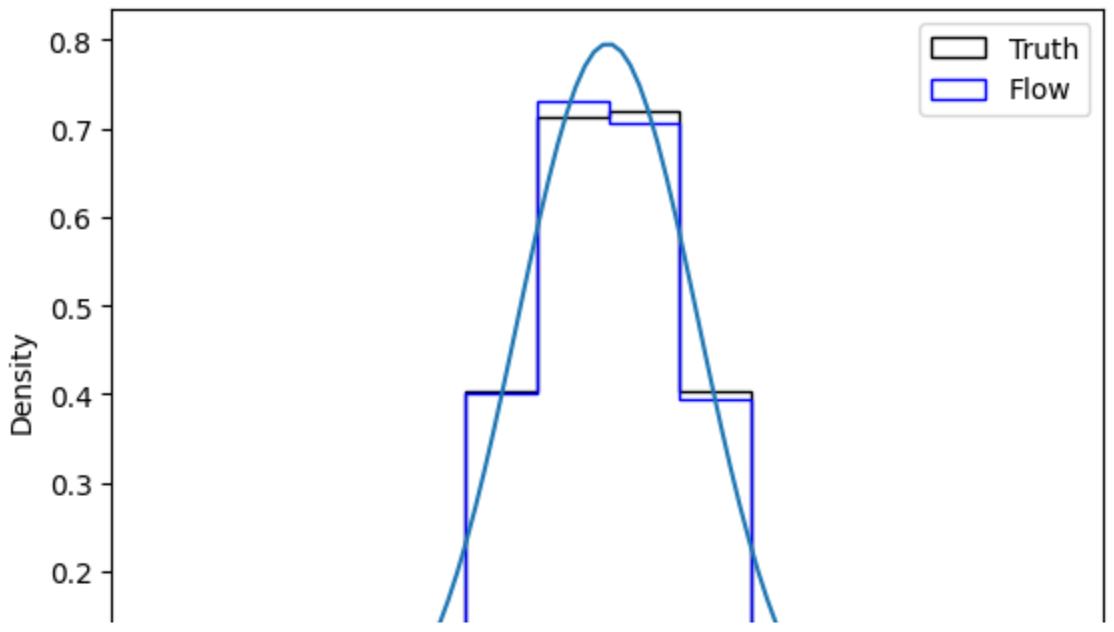
Now, let's train

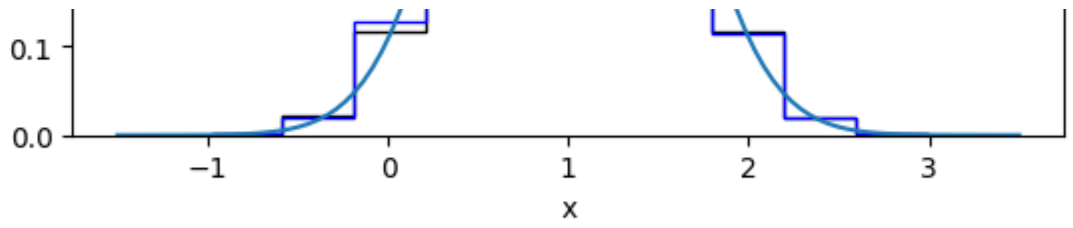
```
# number of epochs
num_iter = 1000

for i in range(num_iter):
    optimizer.zero_grad()
    # the loss is simply - E[Log Prob] !
    loss = -flow.log_prob(inputs=X_torch).mean()
    loss.backward()
    optimizer.step()
```

And now we can re-evaluate

```
pdf_vals_x = np.exp(flow.log_prob(xtoplot_torch).detach().numpy())
x_sample = flow.sample(len(X)).detach().numpy()


a, b, c = plt.hist(X, histtype="step", density=True, label="Truth", color="black")
plt.hist(x_sample, histtype="step", bins=b, density=True, label="Flow", color="blue")
plt.legend(loc="upper right")
plt.plot(xtoplot, pdf_vals_x)
plt.xlabel("x")
plt.ylabel("Density")
```

```
Text(0, 0.5, 'Density')
```





Much better!

You can now play with dimensions or devise some tests to quantify the agreement (like two sample tests, because this simple example is one-dimensional) and define interesting problems. As for us, we'll go to one possible application, Anomaly detection.

Anomaly Detection with flows

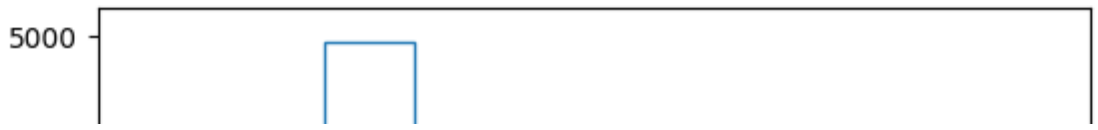
Anomaly detection is in some sense self-explanatory: given a dataset X we want to find a subset X' which is "anomalous" or different. This could be to detect malicious outliers: a spam filter, a banking fraud detector, simply badly measured samples, etc. Or it could be to detect gold: high-reward stocks or options, good fits for a sporting team, etc.

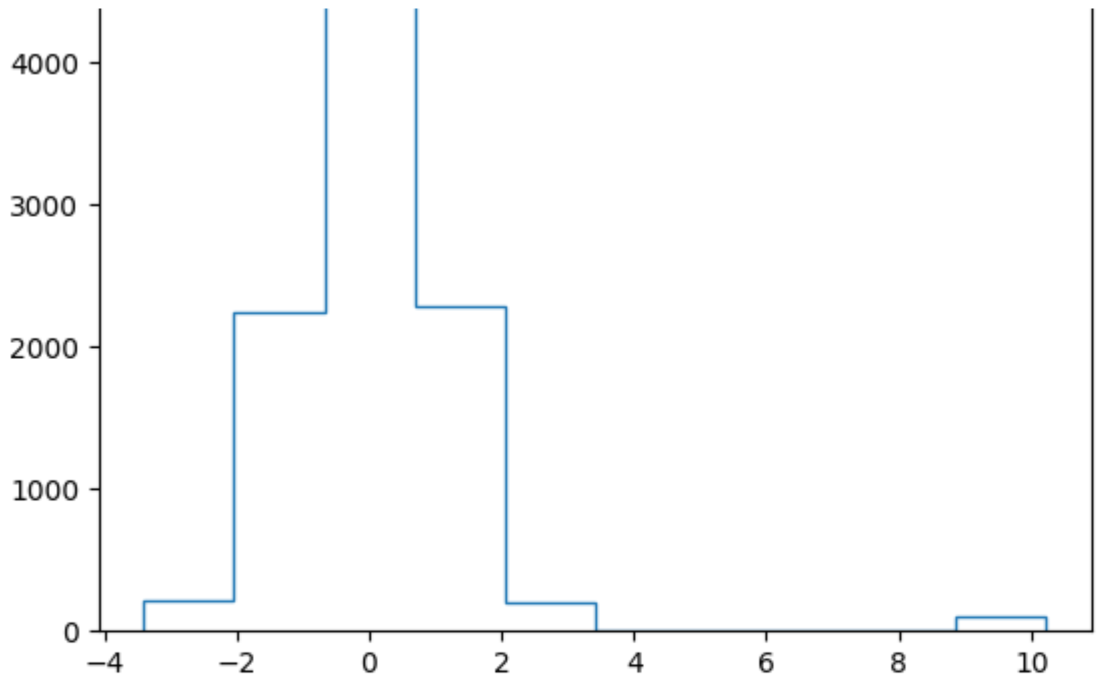
The task will define is the anomaly is good or bad. Additionally, it may define the type of anomaly we seek. There are roughly two types of anomalies: out-of-density and over-densities. Usually, anomaly detection is an **unsupervised** task, where we do not have labels to train our models and simply try to understand the data and find anomalies within. Some methods are **semi-supervised** because they use some noisy labels to get a better sense of what an anomaly is.

In out-of-density cases, we are really looking for some outliers, things that are far away from most of the data. In 1D, this is very easy to visualize

```
N = 10000
X1 = st.norm().rvs(int(0.99 * N))
X2 = st.norm(loc=10, scale=0.1).rvs(int(0.01 * N))
X = np.hstack([X1, X2])
plt.hist(X, histtype="step")
# plt.yscale('log')
```

```
(array([2.160e+02, 2.244e+03, 4.952e+03, 2.285e+03, 2.020e+02, 1.000e+00,
        0.000e+00, 0.000e+00, 0.000e+00, 1.000e+02]),
 array([-3.4029182 , -2.03992303, -0.67692786,  0.68606731,  2.04906248,
         3.41205765,  4.77505281,  6.13804798,  7.50104315,  8.86403832,
        10.22703349]),
 [<matplotlib.patches.Polygon at 0x793afac01b10>])
```





You can see how there is a small subset of data which is far away from the rest. This would be our anomaly. Again, we do not have labels here. You might say that this is very easy: just look at the data and that's it. However, 1D is very misleading. As you increase the number of dimensions, you not only lose visualization but *every* point is in some sense far away from the rest. This is **the curse of dimensionality**.

However, we can use Normalizing Flows to our advantage here. We can simply define the outliers as the points with lowest probability. Thus, our **anomaly score** is simply Log $p(X)$. We can use our anomaly score to select events in the usual way:

Anomalous events (α) = {x | Log $p(x) \leq \alpha$ }

Where α is the parametric choice that selects how anomalous do we want our anomalous events to be. You can think that α defines events whose probability are lower or equal to e^{α}.

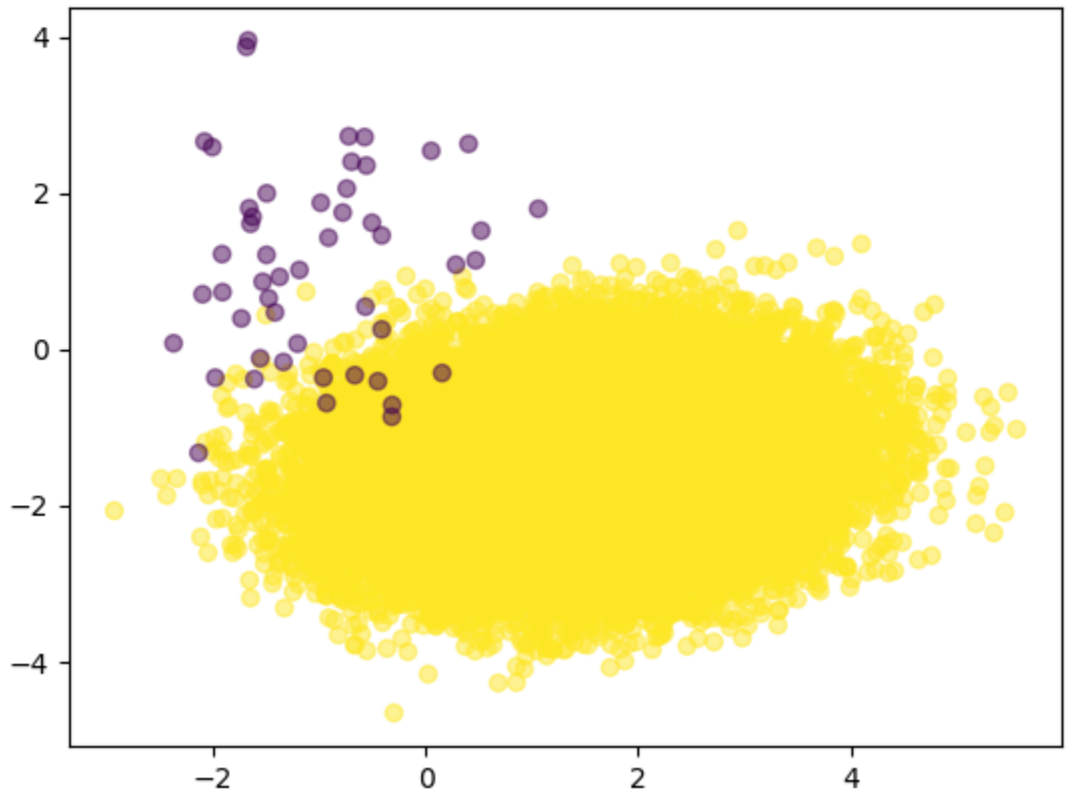
Exercise:

Given the 2D dataset

```
N = 50000
X1 = st.multivariate_normal(mean=[1.5, -1.5], cov=[[1, 0.1], [0.1, 0.5]]).rvs(
    int(0.999 * N)
)
X2 = st.multivariate_normal(mean=[-1.0, 1.0], cov=[[0.7, -0.2], [-0.2, 1.2]]).rvs(
    int(0.001 * N)
)
X = np.vstack([X1, X2])
Y = np.hstack([np.ones(len(X1)), -np.ones(len(X2))])
```

```
print(X.shape)
plt.scatter(X[:, 0], X[:, 1], c=Y, alpha=0.5)
```

```
(50000, 2)
<matplotlib.collections.PathCollection at 0x793afaa5f3d0>
```



Train a Normalizing Flow: this includes implementing a batch-size to deal with the higher number of events (not done in the previous notebook, but can be done simply by sampling a subset of X at each iteration) and more importantly **evaluating** whether the flow has trained succesfully or not. An interesting question here is whether we need to flow to match the exact dataset or just the bulk of the dataset so we get anomalous events. The degree of precision depends on the application.

Use the anomaly score to select "intereting" events. Produce summary plots. Some suggestions: A useful metric (which would not be available in real data) is the fraction of selected events of X2 as a function of a α. Another is a scatter plot as above but where the color is the anomaly score.