Normalising Flow Principle

Background

We start form 1D dimension example , with base probability density function Z~(0,1)

Currently, we are not using neural network to train the model

Our Work

In this code, we first define our prior distribution as standard normal distribution N(0,1), and we define our x to be of nolinear equation:

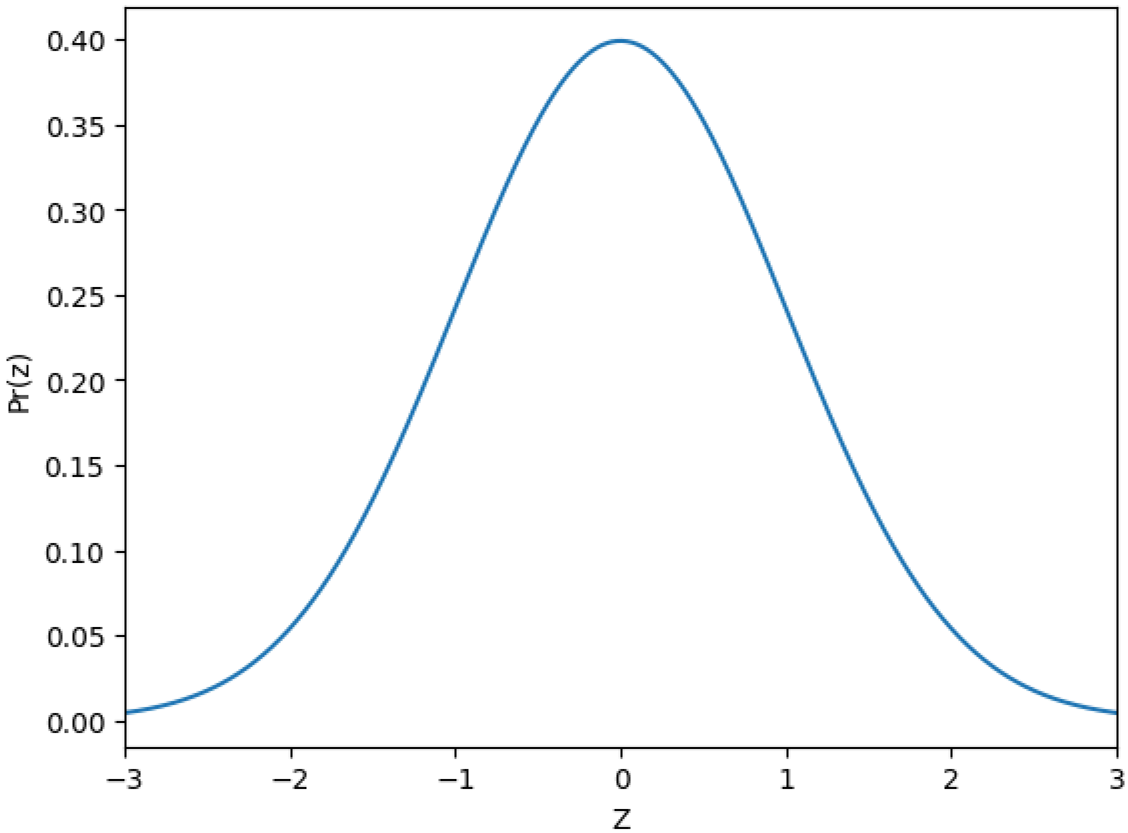
$$
\begin{aligned}
f(z) &= (1-p)\cdot x_1 + p\cdot x_2, \quad \text{where} \\
x_1 &= \frac{6}{1+\exp[-1.5(z-0.25)]} - 3 \\
x_2 &= z \\
p &= \frac{z^2}{9}
\end{aligned}
$$

Then, recall from the book "Deep learning" , we have the distribution equation as:

$$
Pr(\mathbf{x} \mid \boldsymbol{\phi}) = \left| \frac{\partial \mathbf{f}[\mathbf{z}, \boldsymbol{\phi}]}{\partial \mathbf{z}} \right|^{-1} \cdot Pr(\mathbf{z}), \qquad (16.3)
$$

First we define our standard normal, we directily use gauss_pdf to represent our standard normal. Indeed, when we plot the gauss_pdf, it is exaclty a normal

In [16]:
```
import numpy as np
import matplotlib.pyplot as plt
# Define the base pdf
def gauss_pdf(z, mu, sigma):
  pr_z = np.exp( -0.5 * (z-mu) * (z-mu) / (sigma * sigma))/(np.sqrt(2*3.1413
  return pr_z

z = np.arange(-3,3,0.01)
pr_z = gauss_pdf(z, 0, 1)

fig,ax = plt.subplots()
ax.plot(z, pr_z)
ax.set_xlim([-3,3])
ax.set_xlabel('Z')
ax.set_ylabel('Pr(z)')
plt.show();
```



Then define our f(z) and the differentiation. Since it is 1 dimensional, we use central difference to express it :)

In [5]:
```
# Define a function that maps from the base pdf over z to the observed space
def f(z):
    x1 = 6/(1+np.exp(-(z-0.25)*1.5))-3
    x2 = z
    p = z * z/9
    x = (1-p) * x1 + p * x2
    return x

# Compute gradient of that function using finite differences
def df_dz(z):
    return (f(z+0.0001)-f(z-0.0001))/0.0002
```

Check whether the integral is equal to 1 , because we require the integral of any pdf to be 1.

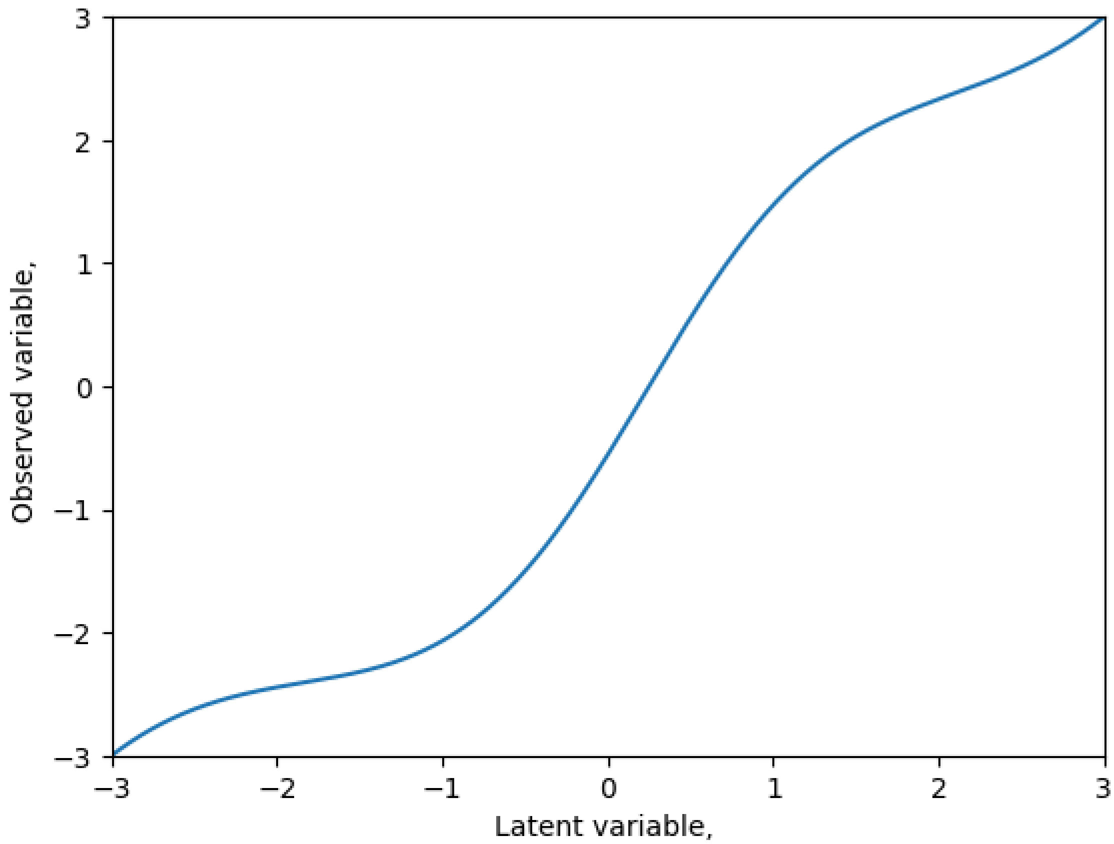
In [7]:
```
area = np.sum(pr_z) * 0.01  # 0.01 is dx
print("Integral of Pr(z):", area)
```

```
Integral of Pr(z): 0.9973464370981742
```

Now plot f(z)

In [8]:
```
x = f(z)
fig, ax = plt.subplots()
ax.plot(z,x)
```

```
ax.set_xlim(-3,3)
ax.set_ylim(-3,3)
ax.set_xlabel('Latent variable, ')
ax.set_ylabel('Observed variable, ')
plt.show()
```



Calculate the distributon $P_r(x)$

In [9]:
```
# TODO -- plot the density in the observed space
# Replace these line
z = np.arange(-3,3,0.01)
x = f(z)
pr_x = pr_z*1/np.abs(df_dz(z))
```

Check the integral = 1?

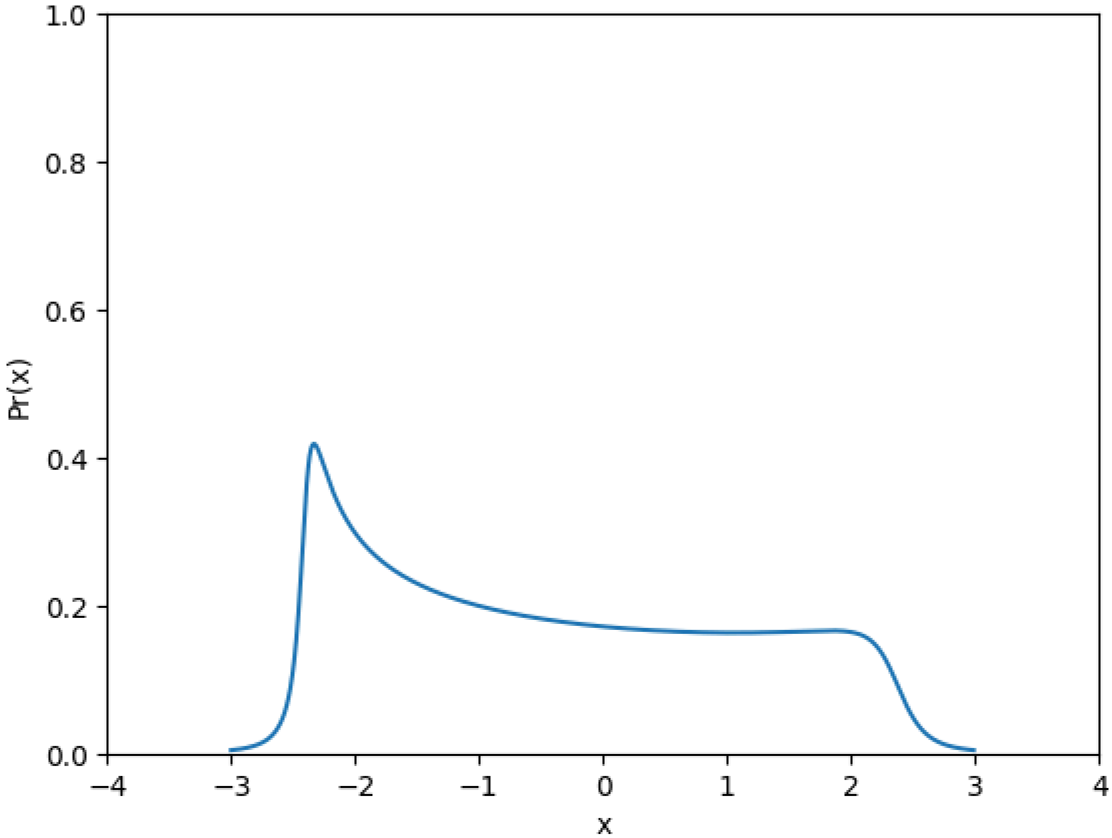
In [10]:
```
area = np.sum(pr_x) * 0.01  # 0.01 is dx
print("Integral of Pr(x):", area)
```

```
Integral of Pr(x): 1.0181088268106
```

From the output, we know that the integral is equal to 1, so the transformation is reasonable.

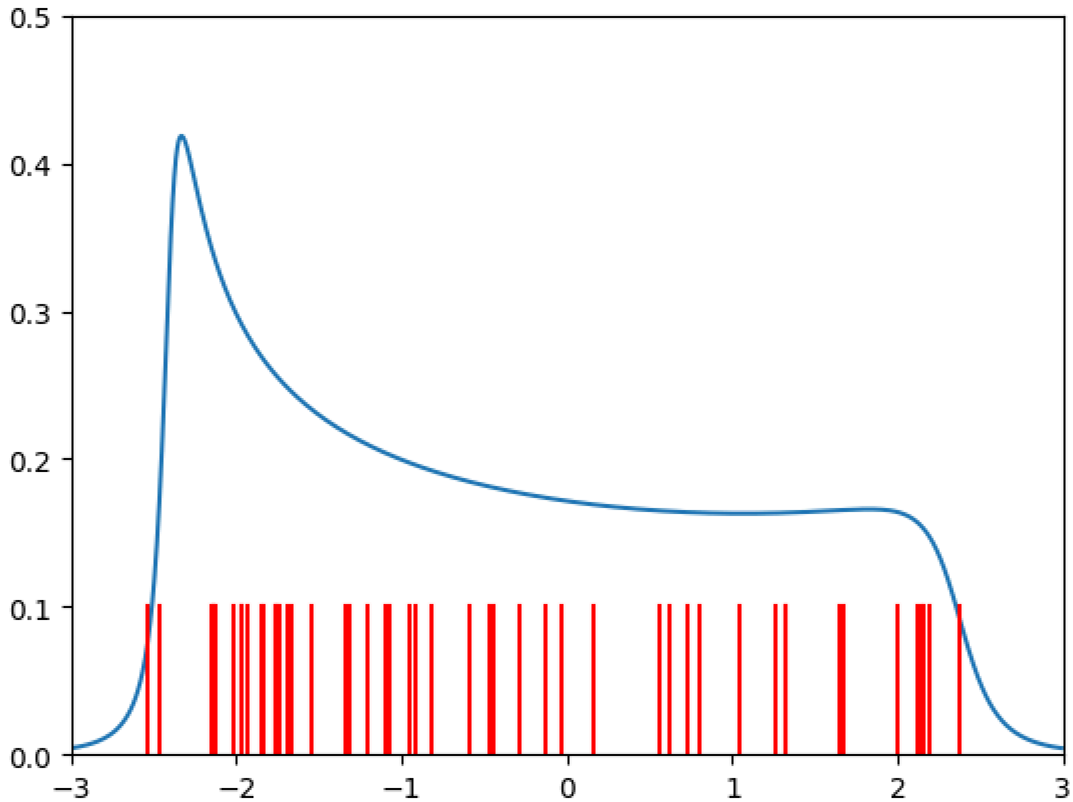
In [11]:
```
# Plot the density in the observed space

fig,ax = plt.subplots()
ax.plot(x, pr_x)
```

```
ax.set_xlim([-4,4])
ax.set_ylim([0, 1])
ax.set_xlabel('x')
ax.set_ylabel('Pr(x)')
plt.show();
```



Now we try to sample from our new distribution - notice that we need to sample from our original distribution - Normal Distribution first then by transformation we have our new samples from x

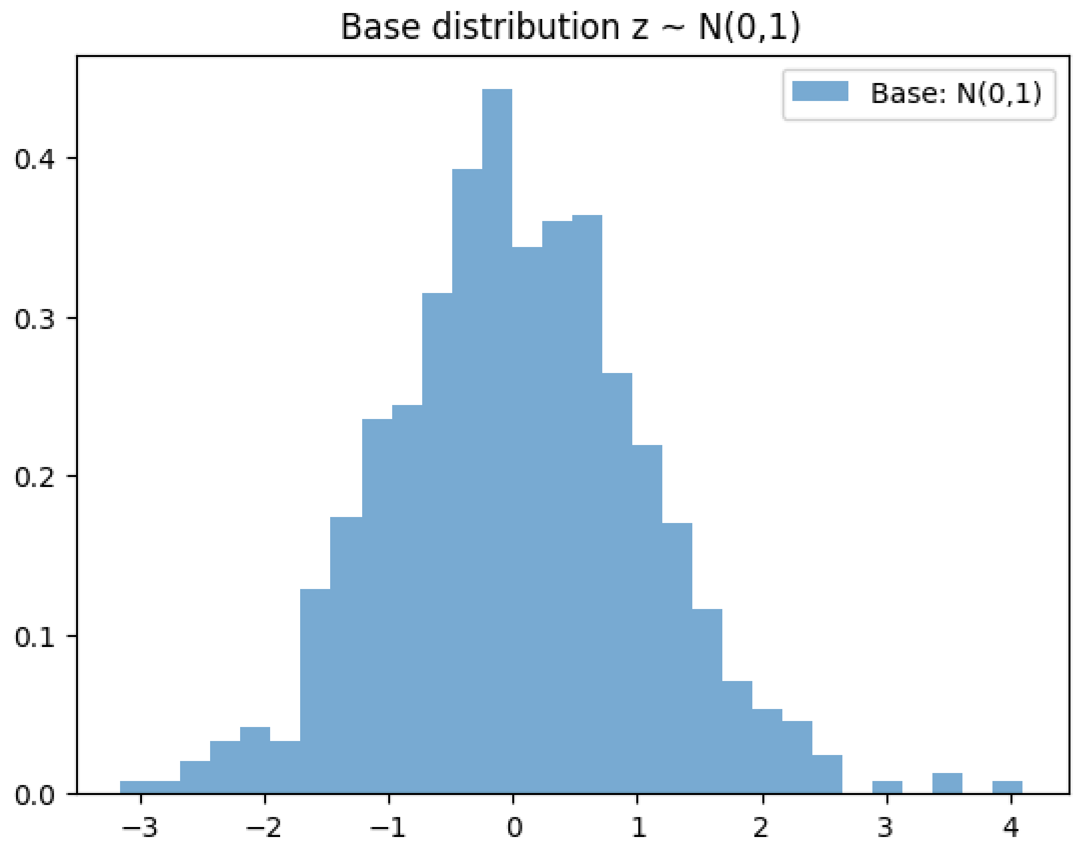
In [12]:
```
np.random.seed(1)
n_sample = 50
z_samples = np.random.normal(0,1,n_sample)
x_samples = f(z_samples)
# Draw the samples
fig,ax = plt.subplots()
ax.plot(x, pr_x)
for x_sample in x_samples:
  ax.plot([x_sample, x_sample], [0,0.1], 'r-')

ax.set_xlim([-3,3])
ax.set_ylim([0, 0.5])
ax.set_xlabel('')
ax.set_ylabel('')
plt.show();
```



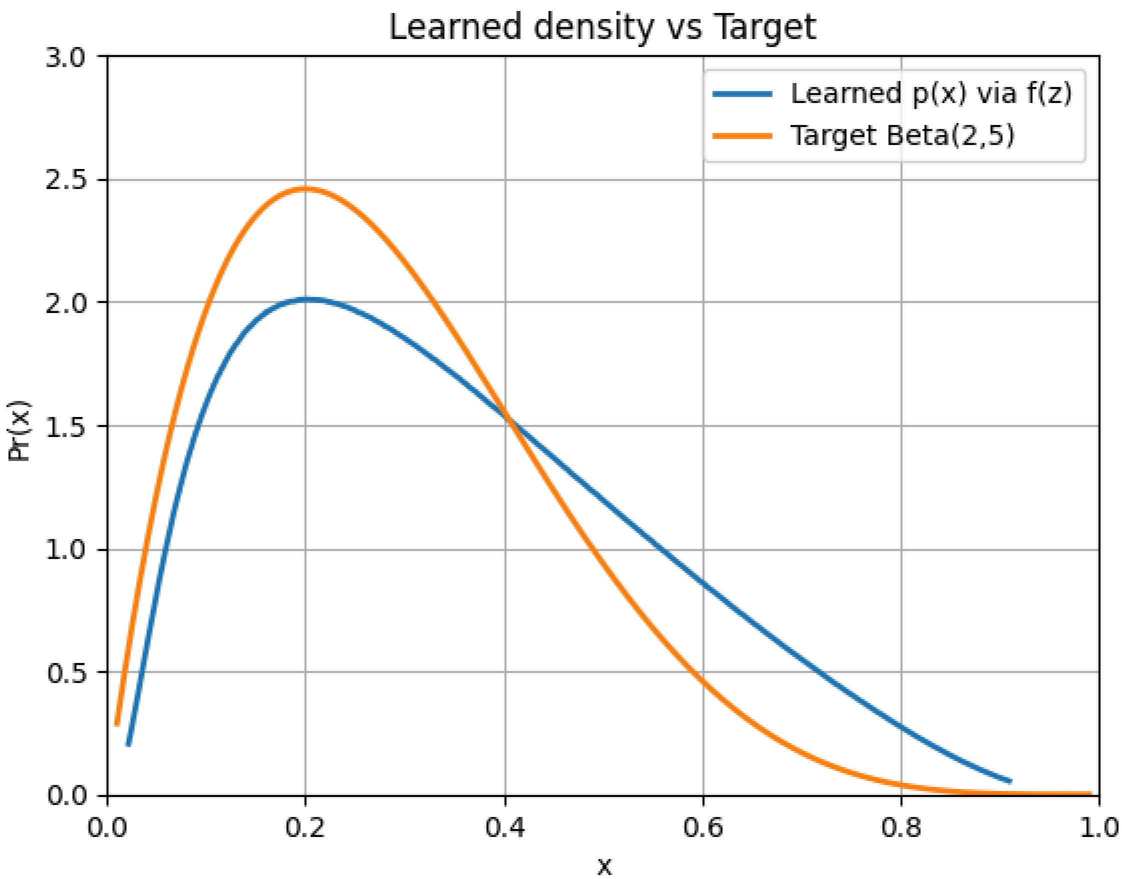
In [2]:
```
import torch
import matplotlib.pyplot as plt
from torch.distributions import Normal
import numpy as np
# Set random seed for reproducibility
torch.manual_seed(0)

# Define the base distribution: standard normal N(0, 1)
base_dist = Normal(loc=0.0, scale=1.0)

# Sample 500 points from the base distribution
z = base_dist.sample((1000,))

# Plot the histogram of the base samples
plt.hist(z.numpy(), bins=30, density=True, alpha=0.6, label='Base: N(0,1)')
plt.title("Base distribution z ~ N(0,1)")
plt.legend()
plt.show()


from torch.distributions import Beta
# Your observed data x ~ Beta(2, 5)
target_dist = Beta(2.0, 5.0)
x_data = target_dist.sample((512,))  # Use x as training data

import torch.nn as nn
import torch

class InvertibleSigmoidAffineTransform(nn.Module):
    def __init__(self):
```

```
        super().__init__()
        self.a = nn.Parameter(torch.tensor(1.0))
        self.b = nn.Parameter(torch.tensor(0.0))

    def forward(self, z):
        return torch.sigmoid(self.a * z + self.b)

    def inverse(self, x):
        # Clamp to avoid log(0) or log(∞)
        x = x.clamp(1e-4, 1 - 1e-4)
        t = torch.log(x / (1 - x))  # inverse sigmoid
        return (t - self.b) / self.a

    def log_abs_det_jacobian(self, x):
        # Use inverse pass
        z = self.inverse(x)
        t = self.a * z + self.b
        s = torch.sigmoid(t)
        return torch.log(torch.abs(self.a) * s * (1 - s) + 1e-8)



def compute_nll(x, transform, base_dist):
    # Invert x to get z
    z = transform.inverse(x)

    # Base log-prob in z space
    log_p_z = base_dist.log_prob(z)

    # Log determinant of Jacobian
    log_det = transform.log_abs_det_jacobian(x)

    # Change-of-variable formula: log p(x) = log p(z) - log|df/dz|
    return -(log_p_z - log_det).sum()



# Transform
transform = InvertibleSigmoidAffineTransform()
optimizer = torch.optim.Adam(transform.parameters(), lr=1e-3)

# Train using negative log-likelihood
for epoch in range(1000):
    x_batch = x_data[torch.randperm(len(x_data))[:256]]
    loss = compute_nll(x_batch, transform, base_dist)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if epoch % 50 == 0:
        print(f"Epoch {epoch}, NLL: {loss.item():.4f}")

def f(z):
    z_tensor = torch.tensor(z, dtype=torch.float32)
    with torch.no_grad():
```

```
        return transform.forward(z_tensor).numpy()

# Compute gradient of that function using finite differences
def df_dz(z):
    eps = 1e-4
    return (f(z + eps) - f(z - eps)) / (2 * eps)


from scipy.stats import norm
z = np.arange(-3,3,0.01)
pr_z = norm.pdf(z, loc=0, scale=1)
x = f(z)
pr_x = pr_z*1/np.abs(df_dz(z))


fig, ax = plt.subplots()

# Plot learned density via change of variables
ax.plot(x, pr_x, label="Learned p(x) via f(z)", linewidth=2)

# Plot target Beta(2,5) PDF
x_curve = torch.linspace(0.01, 0.99, 500)
y_curve = target_dist.log_prob(x_curve).exp().numpy()
ax.plot(x_curve.numpy(), y_curve, label="Target Beta(2,5)", linewidth=2)

# Formatting
ax.set_xlim([0, 1])
ax.set_ylim([0, 3])
ax.set_xlabel('x')
ax.set_ylabel('Pr(x)')
ax.set_title("Learned density vs Target")
ax.grid(True)
ax.legend()

plt.show()
```




```
Epoch 0, NLL: 21.1782
Epoch 50, NLL: 3.5319
Epoch 100, NLL: -6.0916
Epoch 150, NLL: -25.7403
Epoch 200, NLL: -28.7222
Epoch 250, NLL: -35.1980
Epoch 300, NLL: -47.5691
Epoch 350, NLL: -53.3015
Epoch 400, NLL: -57.4336
Epoch 450, NLL: -59.7225
Epoch 500, NLL: -62.7016
Epoch 550, NLL: -66.2620
Epoch 600, NLL: -76.6304
Epoch 650, NLL: -72.8317
Epoch 700, NLL: -84.5013
Epoch 750, NLL: -84.8313
Epoch 800, NLL: -87.4595
Epoch 850, NLL: -95.7812
Epoch 900, NLL: -95.6020
Epoch 950, NLL: -109.9214
```



Normal - Beta Example

Setup

Set the random seed to 0. Define:

- **Base distribution:** $z \sim \mathcal{N}(0, 1)$
- **Target distribution:** $\theta \sim \mathrm{Beta}(2.0,\ 5.0)$
- **Training set:** samples x_i drawn from the Beta target.

Flow Transformation

We model a simple normalizing flow using a sigmoid-affine layer:

$$f(z) = \sigma\bigl(a\,z + b\bigr),$$

where:

- $\sigma(u) = \frac{1}{1+e^{-u}}$ is the sigmoid function,
- a, b are learnable scalars.

The inverse mapping is

$$f^{-1}(x) = \frac{1}{a}\left(\log\frac{x}{1-x} - b\right).$$

We also track the log-absolute-determinant of the Jacobian:

$$\log\left|f'(z)\right| = \log\Bigl(|a|\,\sigma(az+b)\,(1-\sigma(az+b))\Bigr).$$

Training Objective

We train by maximum likelihood. For a sample $x_i = f(z_i)$, the log-density is

$$\log p(x_i) = \log p(z_i) \;-\; \log\left|f'(z_i)\right|.$$

The negative log-likelihood (NLL) loss over all samples is

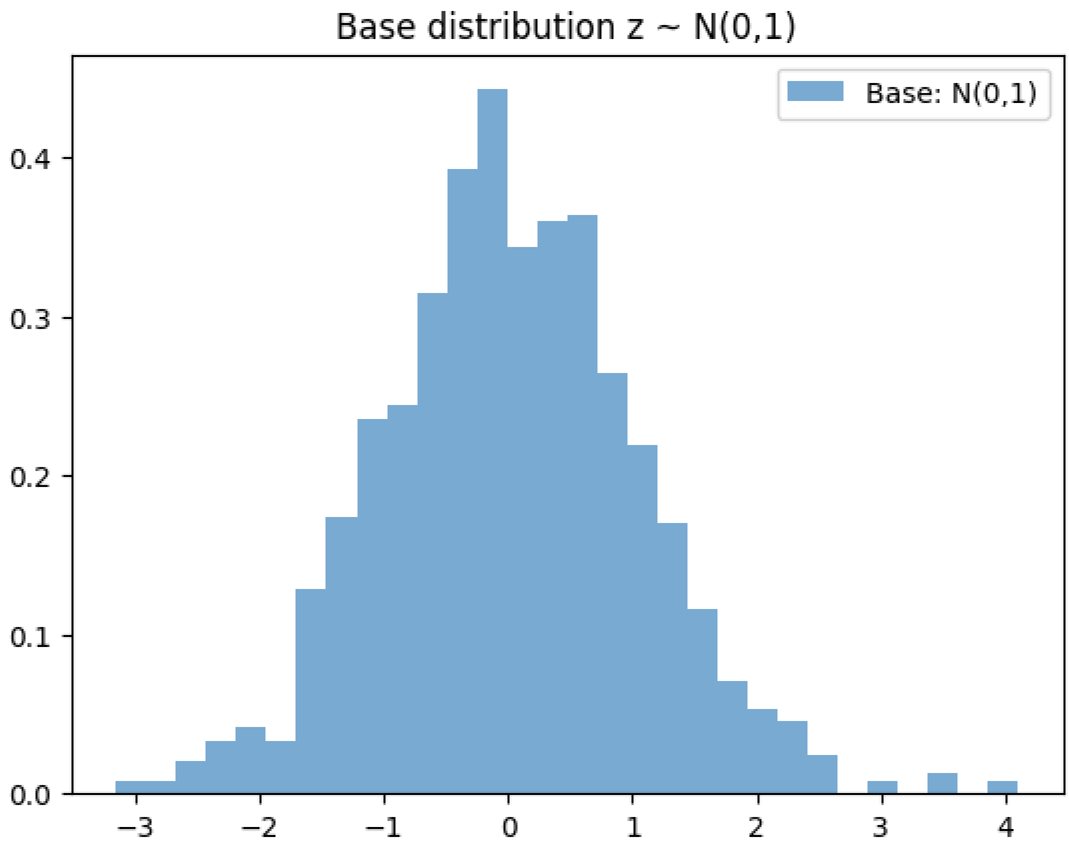
$$\mathcal{L} = -\sum_i \Bigl[\log p(z_i) \;-\; \log\left|f'(z_i)\right|\Bigr].$$

Later, we will experiment with minimizing the KL divergence, which only requires the forward mapping.

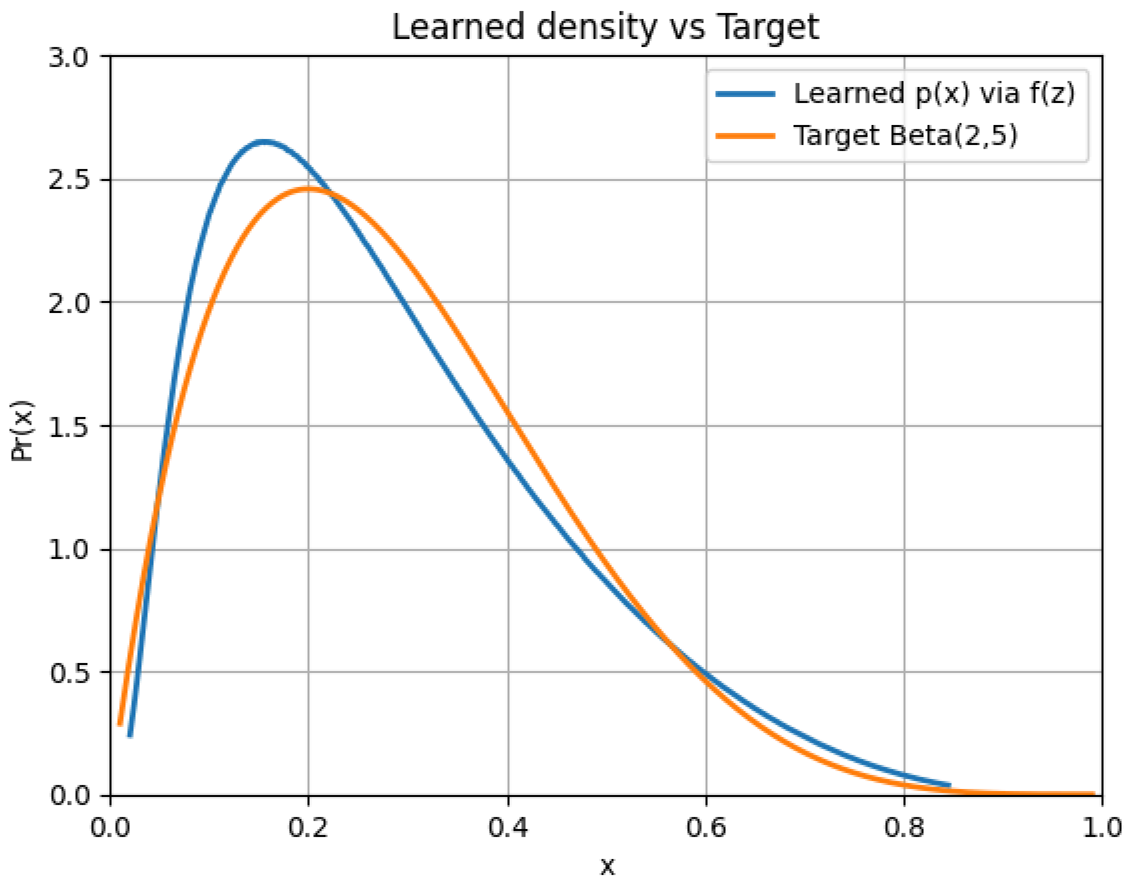
In [40]:
```
import torch
import matplotlib.pyplot as plt
from torch.distributions import Normal
import numpy as np
# Set random seed for reproducibility
torch.manual_seed(0)

# Define the base distribution: standard normal N(0, 1)
base_dist = Normal(loc=0.0, scale=1.0)

# Sample 500 points from the base distribution
z = base_dist.sample((1000,))

# Plot the histogram of the base samples
plt.hist(z.numpy(), bins=30, density=True, alpha=0.6, label='Base: N(0,1)')
plt.title("Base distribution z ~ N(0,1)")
plt.legend()
plt.show()

from torch.distributions import Beta
# Your observed data x ~ Beta(2, 5)
target_dist = Beta(2.0, 5.0)
x_data = target_dist.sample((2000,))  # Use x as training data

import torch.nn as nn
import torch

class InvertibleSigmoidAffineTransform(nn.Module):
    def __init__(self):
        super().__init__()
        self.a = nn.Parameter(torch.tensor(1.0))
        self.b = nn.Parameter(torch.tensor(0.0))
```

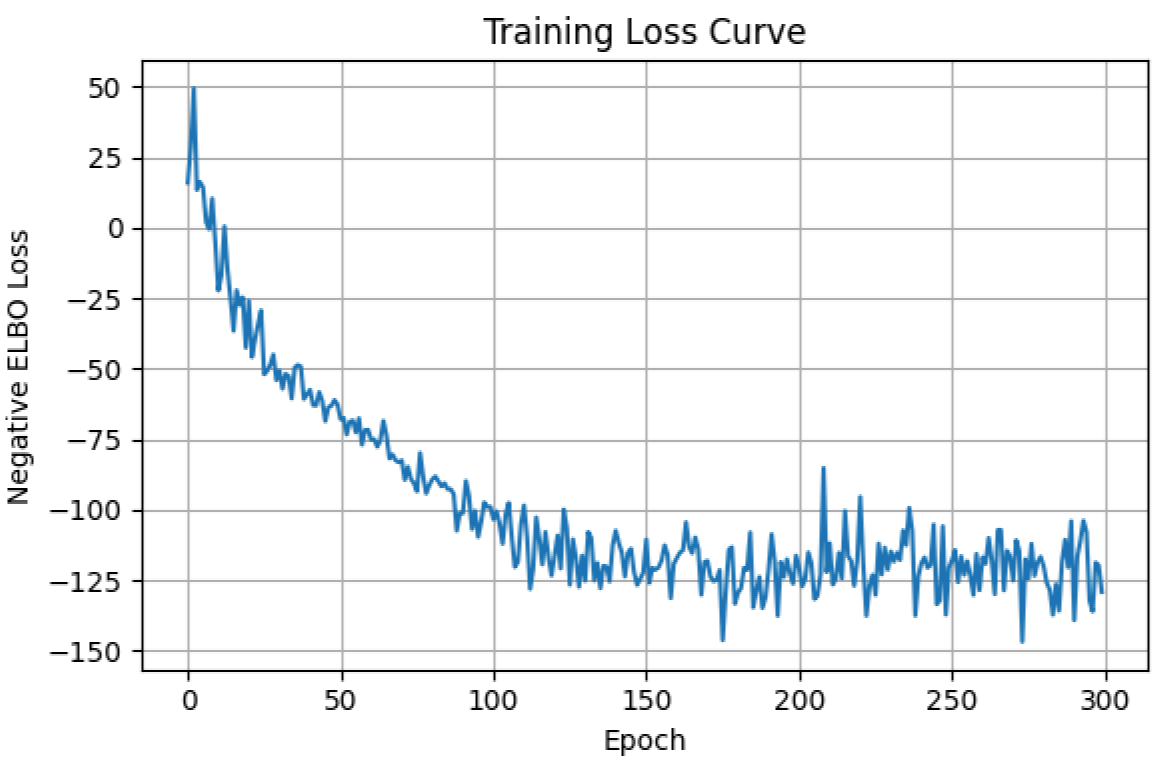
```
    def forward(self, z):
        return torch.sigmoid(self.a * z + self.b)

    def inverse(self, x):
        # Clamp to avoid log(0) or log(∞)
        x = x.clamp(1e-4, 1 - 1e-4)
        t = torch.log(x / (1 - x))  # inverse sigmoid
        return (t - self.b) / self.a

    def log_abs_det_jacobian(self, x):
        # Use inverse pass
        z = self.inverse(x)
        t = self.a * z + self.b
        s = torch.sigmoid(t)
        return torch.log(torch.abs(self.a) * s * (1 - s) + 1e-8)



def compute_nll(x, transform, base_dist):
    # Invert x to get z
    z = transform.inverse(x)

    # Base log-prob in z space
    log_p_z = base_dist.log_prob(z)

    # Log determinant of Jacobian
    log_det = transform.log_abs_det_jacobian(x)

    # Change-of-variable formula: log p(x) = log p(z) - log|df/dz|
    return -(log_p_z - log_det).sum()



# Transform
transform = InvertibleSigmoidAffineTransform()
optimizer = torch.optim.Adam(transform.parameters(), lr=1e-2)

# Train using negative log-likelihood
losses= []
for epoch in range(300):
    x_batch = x_data[torch.randperm(len(x_data))[:256]]
    loss = compute_nll(x_batch, transform, base_dist)
    losses.append(loss.item())
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if epoch % 50 == 0:
        print(f"Epoch {epoch}, NLL: {loss.item():.4f}")




def f(z):
    z_tensor = torch.tensor(z, dtype=torch.float32)
```

```
    with torch.no_grad():
        return transform.forward(z_tensor).numpy()

# Compute gradient of that function using finite differences
def df_dz(z):
    eps = 1e-4
    return (f(z + eps) - f(z - eps)) / (2 * eps)


from scipy.stats import norm
z = np.arange(-3,3,0.01)
pr_z = norm.pdf(z, loc=0, scale=1)
x = f(z)
pr_x = pr_z*1/np.abs(df_dz(z))


fig, ax = plt.subplots()

# Plot learned density via change of variables
ax.plot(x, pr_x, label="Learned p(x) via f(z)", linewidth=2)

# Plot target Beta(2,5) PDF
x_curve = torch.linspace(0.01, 0.99, 500)
y_curve = target_dist.log_prob(x_curve).exp().numpy()
ax.plot(x_curve.numpy(), y_curve, label="Target Beta(2,5)", linewidth=2)

# Formatting
ax.set_xlim([0, 1])
ax.set_ylim([0, 3])
ax.set_xlabel('x')
ax.set_ylabel('Pr(x)')
ax.set_title("Learned density vs Target")
ax.grid(True)
ax.legend()
plt.show()
# Plot the losses
plt.figure(figsize=(6, 4))
plt.plot(losses)
plt.xlabel("Epoch")
plt.ylabel("Negative ELBO Loss")
plt.title("Training Loss Curve")
plt.grid(True)
plt.tight_layout()
plt.show()
```




```
Epoch 0, NLL: 15.9777
Epoch 50, NLL: -67.9393
Epoch 100, NLL: -103.4452
Epoch 150, NLL: -110.5881
Epoch 200, NLL: -120.5078
Epoch 250, NLL: -117.9322
```





Highlights

- **Learning rate (lr):**
  Optimal performance at $\mathrm{lr} = 10^{-2}$. Values both above and below this degrade the

fit.

- **Number of epochs (E):**
  Convergence is robust for $E \geq 500$; training beyond this yields diminishing returns.