Density Estimation with Neural Models

Density Estimation vs Autoencoders

Anomaly detection can be formulated as density estimation

- This is probably the cleanest formulation for the problem
- ...And usually leads to good results

KDE as an estimation technique

- ...Works reasonably well for low-dimensional data
- ...Becomes slower and more data hungry for higher-dimensional data

Autoencoders overcome some of these limitations

- They are faster and less data hungry for high-dimensional data
- They can provide additional insight in the anomalies
- ...But they tend to be worse than D.E. in terms of pure detection power

Let's try to understand why this may be the case...

Density Estimation vs Autoencoders

Anomaly Detection based on D.E. checks whether:

$$-\log f(\mathbf{x}, \lambda) \geq \theta$$

- Where $\mathbf{x}$ is the input vector, f the density estimator, and λ its parameter vector
- θ is the anomaly detection threshold

Anomaly Detection based on autoencoders usually relies on:

$$\|g(\mathbf{x}, \lambda) - \mathbf{x}\|_2^2 \geq \theta'$$

- Where g is the autoencoder, with parameter vector λ
- θ' is again a suitably-chosen detection threshold

Density Estimation vs Autoencoders

The detection condition for autoencoders admits a probabilistic interpretation

Like we did for linear regression, we can rewrite:

$$\|g(\mathbf{x}, \lambda) - \mathbf{x}\|_2^2 \longrightarrow \sum_{j=1}^{m} (g_j(\mathbf{x}, \lambda) - x_j)^2 \longrightarrow \log \prod_{j=1}^{m} \exp\left((g_j(\mathbf{x}, \lambda) - x_j)^2\right)$$

From which, with an affine transformation, for some fixed $\boldsymbol{\sigma}$ we get:

$$\log \frac{1}{\sigma\sqrt{2\pi}} + \frac{1}{\sigma^2} \log \prod_{j=1}^{m} \exp\left((g_j(\mathbf{x}, \lambda) - x_j)^2\right) \quad \longrightarrow$$

$$\longrightarrow \quad \log \prod_{j=1}^{m} \frac{1}{\sigma\sqrt{2\pi}} \exp\left(\left(\frac{g_j(\mathbf{x}, \lambda) - x_j}{\sigma}\right)^2\right)$$

Density Estimation vs Autoencoders

Therefore, optimizing the MSE is equivalent to optimizing

$$-\log \prod_{j=1}^{m} \varphi(x_j \mid g_j(\mathbf{x}, \lambda), \sigma)$$

- I.e. the log likelihood (estimated conditional probability of the data)...
- ...Assuming that the prediction for each x_i is independent and normally distributed
- ...with mean equal to the predictions $g_j(\mathbf{x}, \lambda)$ and fixed standard deviation σ

This is similar to what we observed for Linear Regression

- In LR, we assume normality, independence and fixed variance on the samples
- Here, we do it also on the features

Density Estimation vs Autoencoders

The bottomline

- Even with autoencoders, at training time we solve a density estimation problem
- ...But we do it with some limiting assumptions

This is why D.E.-based anomaly detection tends to work better

So we have

- Either a density estimator with issues on high-dimensional data (KDE)
- ...Or a worse D.E. with good support for high-dimensional data (autoencoders)

Can we get the best of both worlds?

Flow Models

Ideally, we wish a neural approach for density estimation

There are only a handful of approaches, often referred to as flow models:

- Normalizing Flows
- Real Non-Volume Preserving transformations (Real NVP)
- Generative Flow with 1x1 convolutions (Glow)

These are all (fairly) advanced and recent approaches

- Main idea: transforming a simple (and known) probability distribution...
- ...Into a complex (and unknown) distribution that matches that of the available data

As many ML models, they are trained for maximum likelihood

- I.e. to maximize the estimated probability of the available data

Flow Models

All flow models rely on the change of variable formula

- Let x be a random variable representing the source of our data
- Let $p_x(x)$ be its (unknown) density function
- Let z be a random latent variable with known distribution p_z
- Let f be a bijective (i.e. invertible) transformation

Then, the change of variable formula states that:

$$p_x(x) = p_z(f(x)) \left| \det \left(\frac{\partial f(x)}{\partial x^T} \right) \right|$$

- Where $\det$ is the determinant and $\partial f / \partial x^T$ is the Jacobian of f

The formula links the two distributions via the flow model f

Flow Models

Let's consider how we can use the formula

$$p_x(x) = p_z(f(x)) \left| \det \left(\frac{\partial f(x)}{\partial x^T} \right) \right|$$

- Given an example x (e.g. from our dataset)
- We compute the mapping $f(x)$, i.e. the corresponding value for the latent variable z
- ...Plus the determinant of the Jacobian $\partial f / \partial x^T$ in x
- Then we can use the formula to compute the probability of the example

The challenge is defining the transformation f (i.e. the mapping)

- It must be invertible (for the formula to hold)
- It must be non-linear (to handle any distribution)
- It should allow for an easy computation of the determinant

Real NVP

We will use Real Non-Volume Preserving transformations as an example

Real NVPs are a type of neural network

- Input: a vector $\boldsymbol{x}$ representing an example
- Output: a vector $\boldsymbol{z}$ of values for the latent variable
- Key property: $\boldsymbol{z}$ should have a chosen probability distribution
- ...Typically: standard Normal distribution for each $\boldsymbol{z}_i$:

$$z \sim \mathcal{N}(\mathbf{0}, I)$$

In other words

- $\boldsymbol{z}$ follows a multivariate distribution
- ...But the covariance matrix is diagonal, i.e. each component is independent

Real NVP

A Real NVP architecture consists of a stack of affine coupling layers

Each layer treats its input x as split into two components, i.e. $x = (x^1, x^2)$

- One component is passed forward as it is
- The second is processed via an affine transformation

$$
\begin{aligned}
y^1 &= x^1 \\
y^2 &= e^{s(x^1)} \odot x^2 + t(x^1)
\end{aligned}
$$

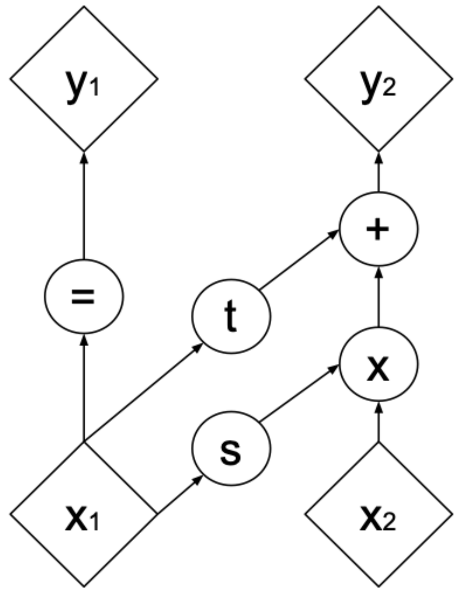
The affine transformation is parameterized with two functions:

- x^2 is scaled using $e^{s(x^1)}$, x^2 is translated using $t(x^1)$
- $\odot$ is the element-wise product (Hadamard product)

Since we have functions rather than fixed vectors, the transformation is non-linear

Real NVP - Affine Coupling Layers

Visually, each layer has the following compute graph:



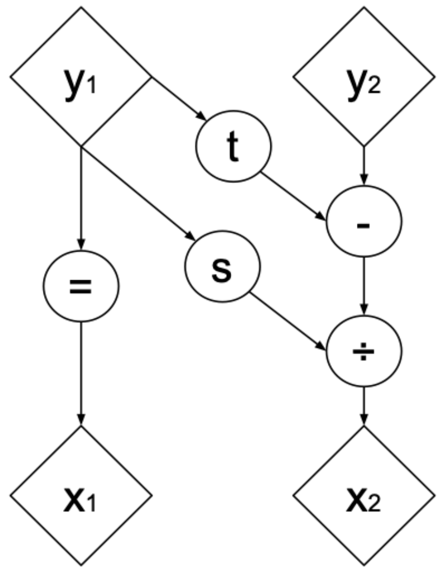
- We are using part of the input (i.e. x^1)...
- ...To transform the remaining part (i.e. x^2)

Both s and t are usually implemented as Multilayer Perceptrons

- I.e. pretend there are a few fully connected layers when you see s and t

Real NVP - Affine Coupling Layers

Each affine coupling layer is easy to invert



Since part of the input (i.e. x^1) has been passed forward unchanged, we have that:

$$x^1 = y^1$$
$$x^2 = (y^2 - t(y^1)) \oslash e^{s(y^1)}$$

- $\oslash$ is the element-wise division

Real NVP - Affine Coupling Layers

The determinant of each layer is easy to compute

The Jacobian of the transformation is:

$$\frac{\partial y}{\partial x^T} = \begin{pmatrix} I & 0 \\ \frac{\partial t(x^1)}{\partial x^T} & \mathrm{diag}(e^{s(x^1)}) \end{pmatrix}$$

The most (only, actually) important thing is that the matrix is triangular:

- ...Hence, its determinant is the product of the terms on the main diagonal:

$$\det\left(\frac{\partial y}{\partial x^T}\right) = \prod_{j \in I_{x_1}} e^{s(x_i^1)} = \exp\left(\sum_{j \in I_{x_1}} s(x_i^1)\right)$$

Real NVP - Considerations

Overall, we have a transformation that:

- ...Is non-linear, and can be made arbitrarily deep
- ...Is Invertible (so as to allow application of the change of variable formula)
- ...Is well suited for determinant computation

Depth and non-linearity are very important:

- The whole approach works only if we can construct a mapping between x and

z...

- ...I.e. if we can transform one probability distribution into the other

A poor mapping will lead to poor estimates

Real NVP - Considerations

At training time we maximize the log likelihood...

...Hence we care about log probabilities:

$$\log p_x(x) = \log p_z(f(x)) + \log \left|\det\left(\frac{\partial f(x)}{\partial x^T}\right)\right|$$

- If we choose a Normal distribution for z, the log cancels all exponentials in the formula
- I.e. the one in the Normal PDF and the one in the determinant computation

In general, we want to make sure that all variables are transformed

- We need to be careful to define the x^1, x^2 components on different layers...
- ...So that no variable is passed forward unchanged along the whole network

A simple approach: alternate the roles (i.e. swap the role of x^1, x^2 at every layer)

Real NVP as Generative Models

Since Real NVPs are invertible, they can be used as generative models

Formally, they can sample from the distribution they have learned

- We just need to sample from p_z, i.e. on the latent space
  - ...And this is easy since the distribution is simple an known
- Then we go through the whole architecture backwards
  - ...Using the inverted version of the affine coupling layers

In fact, generating data is often their primary purpose

They can (or could) be used for:

- Super resolution
- Procedural content generation
- Data augmentation (relevant in an industrial context)

Recent versions allow for data generation with controlled attributes

Implementing Real NVPs

Implementing Real NVPs

We will now see how to implement Real NVPs

The basis from our code comes from the official keras documentation

- It will rely partially on low-level APIs of keras

We start by importing several packages:

```
import tensorflow as tf
import tensorflow_probability as tfp
from tensorflow.keras.layers import Dense
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.regularizers import l2
from sklearn.datasets import make_moons
```

- `tensorflow_probability` is a tensorflow extension for probabilistic computations
- ...And allows for easy manipulation of probability distributions

Affine Coupling Layer

Then we define a function to build each affine coupling layer:

```python
def coupling(input_shape, nunits=64, nhidden=2, reg=0.01):
    assert(nhidden >= 0)
    x = keras.layers.Input(shape=input_shape)
    # Build the layers for the t transformation (translation)
    t = x
    for i in range(nhidden):
        t = Dense(nunits, activation="relu", kernel_regularizer=l2(reg))(t)
    t = Dense(input_shape, activation="linear", kernel_regularizer=l2(reg))(t)
    # Build the layers for the s transformation (scale)
    s = x
    for i in range(nhidden):
        s = Dense(nunits, activation="relu", kernel_regularizer=l2(reg))(s)
    s = Dense(input_shape, activation="tanh", kernel_regularizer=l2(reg))(s)
    # Return the layers, wrapped in a keras Model object
    return keras.Model(inputs=x, outputs=[s, t])
```

Affine Coupling Layer

This part of the code builds the translation (i.e. t) function:

```
def coupling(input_shape, nunits=64, nhidden=2, reg=0.01):
    ...
    x = keras.layers.Input(shape=input_shape)
    t = x
    for i in range(nhidden):
        t = Dense(nunits, activation="relu", kernel_regularizer=l2(reg))(t)
    t = Dense(input_shape, activation="linear", kernel_regularizer=l2(reg))(t)
    ...
```

- It's just a Multi-Layer Perceptron built using the functional API
- The output represents an offset, hence the "linear" activation function in the last layer

Affine Coupling Layer

This part of the code builds the translation (i.e. t) function:

```
def coupling(input_shape, nunits=64, nhidden=2, reg=0.01):
    ...
    x = keras.layers.Input(shape=input_shape)
    t = x
    for i in range(nhidden):
        t = Dense(nunits, activation="relu", kernel_regularizer=l2(reg))(t)
    t = Dense(input_shape, activation="linear", kernel_regularizer=l2(reg))(t)
    ...
```

- The output and input have the same shape, but x^1 and x^2 may have different size
- This will be resolved by masking some of the output of the affine layer
- ...The masked portions will have no effect, with effectively the same result
- The main drawback is higher memory consumption (and computational cost)

Affine Coupling Layer

This part of the code builds the scaling (i.e. s) function:

```
def coupling(input_shape, nunits=64, nhidden=2, reg=0.01):
    ...
    x = keras.layers.Input(shape=input_shape)
    ...
    s = x
    for i in range(nhidden):
        s = Dense(nunits, activation="relu", kernel_regularizer=l2(reg))(s)
    s = Dense(input_shape, activation="tanh", kernel_regularizer=l2(reg))(s)
    ...
```

- Another MLP, with a bipolar sigmoid ("tanh") activation function in the output layer
- Using "tanh" limits the amount of scaling per affine coupling layer
- ...Which in turn makes training more numerically stable
- For the same reason, we use L2 regularizers on the MPL weights

RNVP Model

Then, we define a Real NVP architecture by subclassing keras.model

```python
class RealNVP(keras.Model):
    def __init__(self, input_shape, num_coupling, units_coupling=32, depth_coupling=0,
            reg_coupling=0.01): ...
    @property
    def metrics(self): ...
    def call(self, x, training=True): ...
    def log_loss(self, x): ...
    def score_samples(self, x): ...
    def train_step(self, data): ...
    def test_step(self, data): ...
```

- We will now discuss the most important methods
- Sometimes with a few simplifications (for sake of clarity)

RNVP Model

The `__init__` method (constructor) initializes the internal fields

```
    def __init__(self, input_shape, num_coupling, units_coupling=32, depth_coupling=0,
            reg_coupling=0.01):
        super(RealNVP, self).__init__()
        self.distribution = tfp.distributions.MultivariateNormalDiag(
            loc=np.zeros(input_shape, dtype=np.float32),
            scale_diag=np.ones(input_shape, dtype=np.float32)
        )
        half_n = int(np.ceil(input_shape/2))
        m1 = ([0, 1] * half_n)[:input_shape]
        m2 = ([1, 0] * half_n)[:input_shape]
        self.masks = np.array([m1, m2] * (num_coupling // 2), dtype=np.float32)
        self.loss_tracker = keras.metrics.Mean(name="loss")
        self.layers_list = [coupling(input_shape, units_coupling, depth_coupling, reg_cou
ing)
                            for i in range(num_coupling)]
```

RNVP Model

The `__init__` method (constructor) initializes the internal fields

```
    def __init__(self, input_shape, num_coupling, units_coupling=32, depth_coupling=0,
            reg_coupling=0.01):
        ...
        self.distribution = tfp.distributions.MultivariateNormalDiag(
            loc=np.zeros(input_shape, dtype=np.float32),
            scale_diag=np.ones(input_shape, dtype=np.float32)
        )
        ...
```

Here we build a `tfp` object to handle the known distribution

- As it is customary, we chosen a Multivariate Normal distribution
- ...With independent components, zero mean, and unary standard deviation

RNVP Model

The `__init__` method (constructor) initializes the internal fields

```
def __init__(self, input_shape, num_coupling, units_coupling=32, depth_coupling=0,
        reg_coupling=0.01):
    ...
    half_n = int(np.ceil(input_shape/2))
    m1 = ([0, 1] * half_n)[:input_shape]
    m2 = ([1, 0] * half_n)[:input_shape]
    self.masks = np.array([m1, m2] * (num_coupling // 2), dtype=np.float32)
    ...
```

Here we build the masks to discriminate the x_1 and x_2 components at each layer

- As in the original RNVP paper, we use an alternating checkboard pattern
    - I.e. we take even indexes at one layer, and odd indexes at the next layer
- ...So that all variables are transformed, if we have at least 2 affine coupling layers

RNVP Model

The `__init__` method (constructor) initializes the internal fields

```
    def __init__(self, input_shape, num_coupling, units_coupling=32, depth_coupling=0,
            reg_coupling=0.01):
        ...
        self.layers_list = [coupling(input_shape, units_coupling, depth_coupling, reg_cou
ing)
                              for i in range(num_coupling)]
```

Finally, here we build the model layers

- Each one consists in an affine coupling
- ...And contains in turn two Multi Layer Perceptrons
- Recall that we need at least 2 affine couplings to transform all variables

RNVP Model

The `call` method handles the transformation, in both directions

```
def call(self, x, training=True):
    log_det_inv, direction = 0, 1
    if training: direction = -1
    for i in range(self.num_coupling)[::direction]:
        x_masked = x * self.masks[i]
        reversed_mask = 1 - self.masks[i]
        s, t = self.layers_list[i](x_masked)
        s, t = s*reversed_mask, t*reversed_mask
        gate = (direction - 1) / 2
        x = reversed_mask * (x * tf.exp(direction * s) + direction * t * tf.exp(gate * s)
 \
            + x_masked
        log_det_inv += gate * tf.reduce_sum(s, axis=1)
    return x, log_det_inv
```

RNVP Model

The `call` method handles the transformation, in both directions

```
def call(self, x, training=True):
    log_det_inv, direction = 0, 1
    if training: direction = -1
    for i in range(self.num_coupling)[::direction]:
    ...
```

The `direction` variable controls the direction of the transformation

- By default, this implementation transforms z into x
    - I.e. it works backwards, compared to our theoretical discussion
- This is the case since RNVP are often mainly used as generative models
- At training time, we always want to transform x into z
- ...And this is why `direction = -1` when `training` is `True`

RNVP Model

The `call` method handles the transformation, in both directions

```
def call(self, x, training=True):
    for i in range(self.num_coupling)[::direction]:
        x_masked = x * self.masks[i]
        reversed_mask = 1 - self.masks[i]
        s, t = self.layers_list[i](x_masked)
        s, t = s*reversed_mask, t*reversed_mask
        ...
```

- Here we mask x, i.e. filter the x_1 subset of variables
- ...We compute the value of the s and t function
- Then we filter such values using a the reversed (i.e. negated) mask
- I.e. prepare s and t for their application to the x_2 subset

RNVP Model

The `call` method handles the transformation, in both directions

```
def call(self, x, training=True):
        ...
        gate = (direction - 1) / 2
        x = reversed_mask * (x * tf.exp(direction * s) + direction * t * tf.exp(gate * s)
 \
            + x_masked
        ...
```

Here we compute the main transformation (backwards, as mentioned):

- If `training = True`, we have `direction = -1` and we compute:

$$x^1 = y^1$$
$$x^2 = (y^2 - t(y^1)) \oslash e^{s(y^1)}$$

RNVP Model

The `call` method handles the transformation, in both directions

```
def call(self, x, training=True):
        ...
        gate = (direction - 1) / 2
        x = reversed_mask * (x * tf.exp(direction * s) + direction * t * tf.exp(gate * s)
 \
            + x_masked
        ...
```

Here we compute the main transformation (backwards, as mentioned):

- If `training = False`, we have `direction = 1` and we compute:

$$
\begin{aligned}
y^1 &= x^1 \\
y^2 &= e^{s(x^1)} \odot x^2 + t(x^1)
\end{aligned}
$$

RNVP Model

The `call` method handles the transformation, in both directions

```
def call(self, x, training=True):
    ...
    for i in range(self.num_coupling)[::direction]:
        ...
        log_det_inv += gate * tf.reduce_sum(s, axis=1)
    return x, log_det_inv
```

At each layer, we also compute the $\log \det$ of the Jacobian

- ...Which is simply the sum of the s function values
- Determinants of different layers should be multiplied (due to the chain rule)...
- ...Which means that their $\log$ is simply summed

At then end of the process, the determinant has been computed

RNVP Model

The `score_samples` method performs density estimation

```
def score_samples(self, x):
    y, logdet = self(x)
    log_probs = self.distribution.log_prob(y) + logdet
    return log_probs
```

The process relies on the change of variable formula:

- First, it triggers the `call` method with `training=True`
  - I.e. transforms data points x into their latent representation z
- Then, it computes the (log) density of z
  - Using `tensorflow_probability` comes in handy at this point
- ...And then sums the log determinant

RNVP Model

The `log_loss` method computes the loss function

```
def log_loss(self, x):
    log_densities = self.score_samples(x)
    return -tf.reduce_mean(log_densities)
```

This is done by:

- Obtaining the estimated densities via `score_samples`
- ...Summing up (in log scale, i.e. a product in the original scale)
- ...And finally swapping the sign of the resut
  - ...Since we want to maximize the likelihood

RNVP Model

The `train_step` method is called by the keras `fit` method

```
def train_step(self, data):
    with tf.GradientTape() as tape:
        loss = self.log_loss(data)
    g = tape.gradient(loss, self.trainable_variables)
    self.optimizer.apply_gradients(zip(g, self.trainable_variables))
    self.loss_tracker.update_state(loss)
    return {"loss": self.loss_tracker.result()}
```

The `GradientTape` is how tensorflow handles differentiation

- All tensor operations made in the scope of a `GradientTape` are tracked
- ...So that a gradient can then be extracted
- Then we apply the gradient to the model weights (using the optimizer)
- ...And finally we track the loss

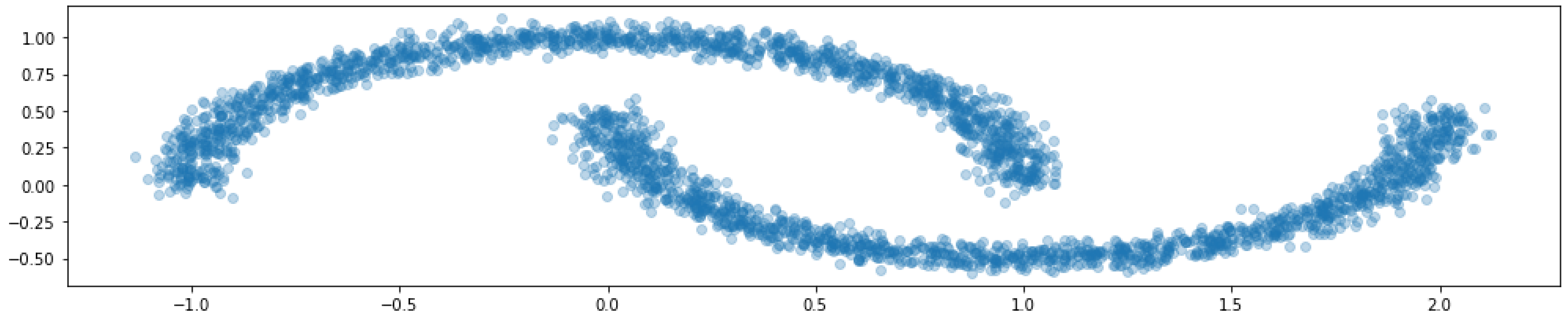
Using Real NVPs

Using Real NVP

We are ready to test our model

We will use a classical benchmark for density estimation (shaped like two half moons)

```
In [6]: from sklearn.datasets import make_moons
        data = make_moons(3000, noise=0.05)[0].astype(np.float32)
        nn.plot_distribution_2D(samples=data, figsize=figsize)
```



- We use `float32` numbers for easier interplay with tensorflow

Training

Now, we need to train a Real NVP model

- We will use the whole dataset (this is just a simple test)
- ...But first, we need to standardize it

```
In [7]: data_s = (data - data.mean(axis=0)) / data.std(axis=0)
```

Standardization is very important when using Real NVPs

- This is true for Neural Networks in general, for the usual reasons
- But even more in this case, since the distribution for z is standardized
  - Standardizing the data makes it easier to learn a mapping

Training

Next we can perform training, as usual in keras

In [8]:
```
from tensorflow.keras.callbacks import EarlyStopping
model = nn.RealNVP(input_shape=2, num_coupling=10, units_coupling=32, depth_coupling=2, reg_coup
model.compile(optimizer='Adam')
cb = [EarlyStopping(monitor='loss', patience=40, min_delta=0.0001, restore_best_weights=True)]
history = model.fit(data_s, batch_size=256, epochs=200, verbose=1, callbacks=cb)
```

```
Epoch 1/200
12/12 [==============================] - 4s 4ms/step - loss: 2.9582
Epoch 2/200
12/12 [==============================] - 0s 4ms/step - loss: 2.7342
Epoch 3/200
12/12 [==============================] - 0s 4ms/step - loss: 2.5690
Epoch 4/200
12/12 [==============================] - 0s 4ms/step - loss: 2.5072
Epoch 5/200
12/12 [==============================] - 0s 4ms/step - loss: 2.4623
Epoch 6/200
12/12 [==============================] - 0s 4ms/step - loss: 2.4218
Epoch 7/200
12/12 [==============================] - 0s 4ms/step - loss: 2.3766
Epoch 8/200
12/12 [==============================] - 0s 4ms/step - loss: 2.3357
Epoch 9/200
12/12 [==============================] - 0s 4ms/step - loss: 2.2893
Epoch 10/200
12/12 [==============================] - 0s 4ms/step - loss: 2.2384
Epoch 11/200
```

Training

As usual with NNs, choosing the right architecture can be complicated

```
model = RealNVP(input_shape=2, num_coupling=16, units_coupling=32, depth_coupling=2, reg_
upling=0.01)
```

- We went for a relatively deep model (10 affine coupling)
- Each coupling has also a good degree of non-linearity (2 hidden layers)
- We used a small degree of L2 regularization to stabilize the training process

We also use relatively large batch size

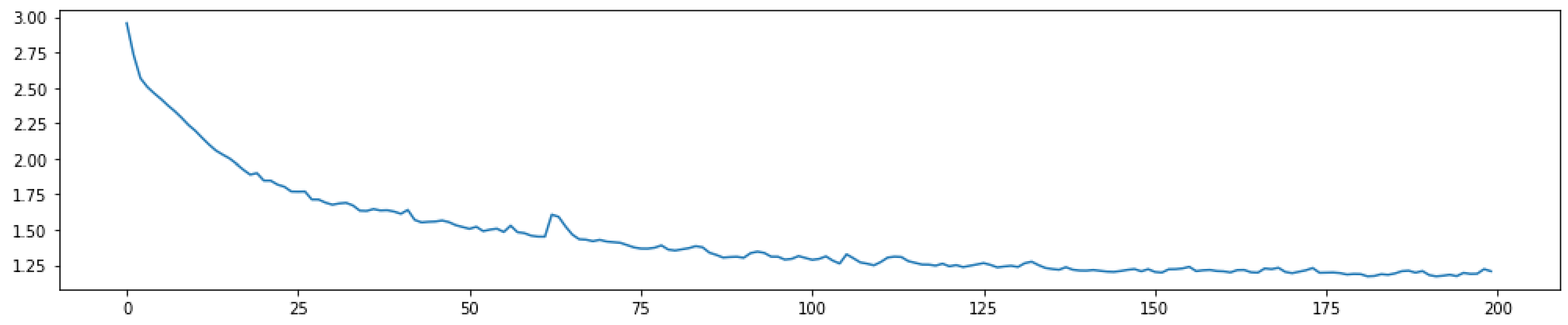
```
history = model.fit(data_s, batch_size=256, epochs=200, verbose=2, callbacks=cb)
```

- Large batch sizes are usually a good choice with density estimation approaches
- Batches should be ideally be representative of the distribution

Training

Let's see the evolution of the training loss over time

```
In [9]: nn.plot_training_history(history, figsize=figsize)
```

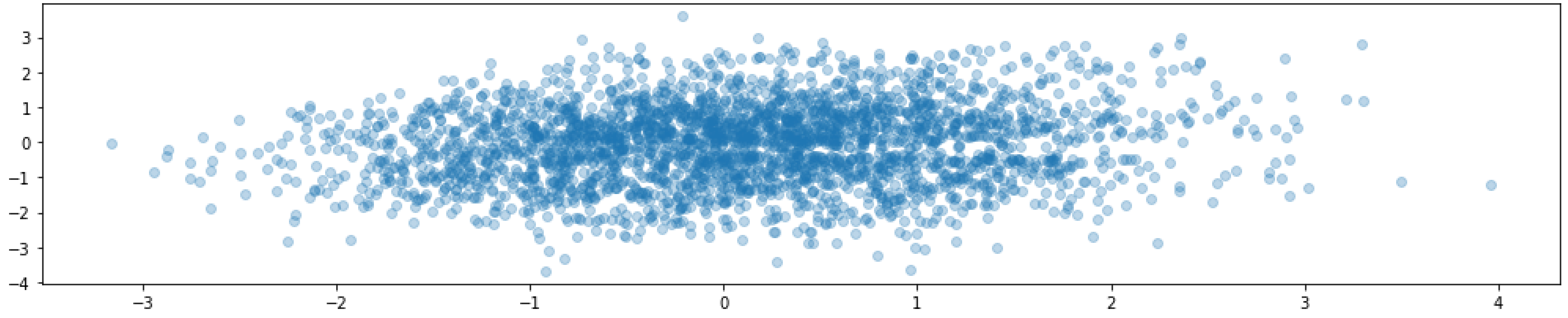


Latent Space Representation

We can obtain the latent space representation by calling the trained model

This will trigger the `call` method with default parameters (i.e. `training=True`)

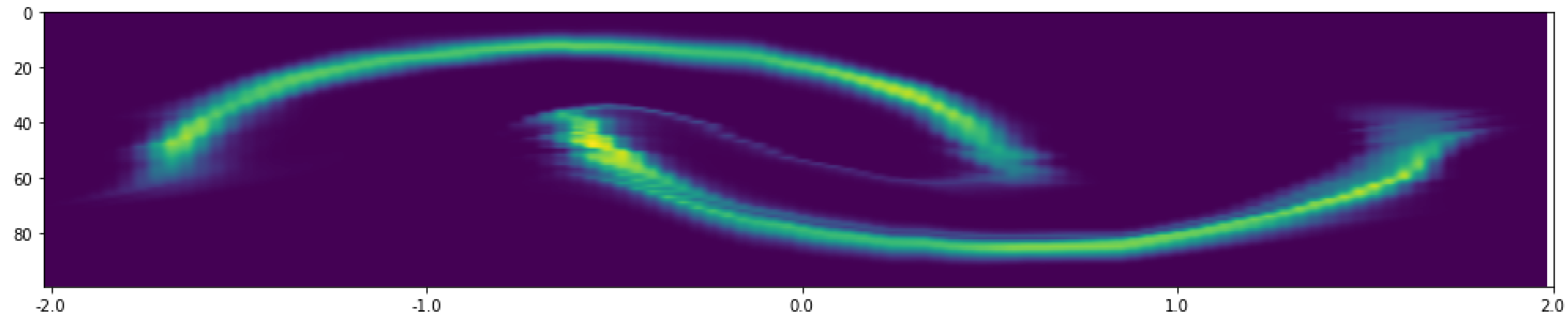
```
In [10]: z, _ = model(data_s)
         nn.plot_distribution_2D(samples=z, figsize=figsize)
```



Density Estimation

We can estimate the density of any data point

```
In [11]: nn.plot_distribution_2D(estimator=model, xr=np.linspace(-2, 2, 100, dtype=np.float32),
                                                  yr=np.linspace(-2, 2, 100, dtype=np.float32), figsize=f
```



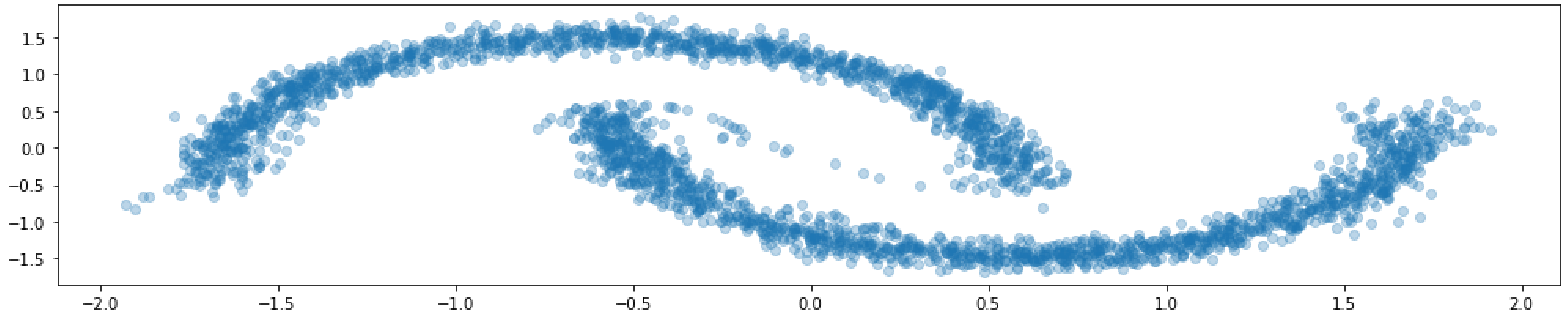
- A good approximation! With a strange low-density connection between the moons

Data Generation

We can also generate data, by sampling from p_z and then calling `predict`

This will trigger the `call` method with `training=False`

```
In [12]: samples = model.distribution.sample(3000)
         x, _ = model.predict(samples)
         nn.plot_distribution_2D(samples=x, figsize=figsize)
```

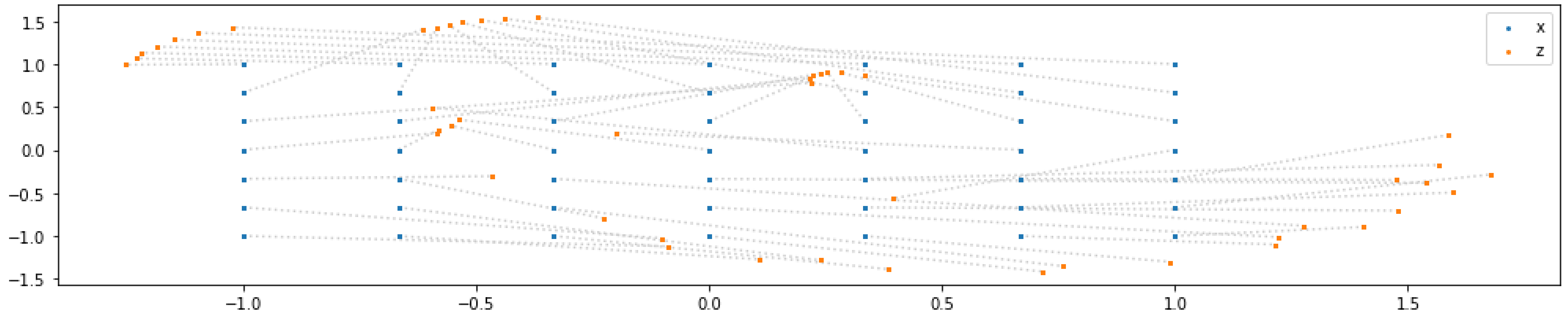


Data Generation

We can also plot the mapping for selected data points...

...Which gives and intuition of how the transformation works

```
In [13]: nn.plot_rnvp_transformation(model, figsize=figsize)
```



RNVP for Anomaly Detection

RNVP for Anomaly Detection

RNVPs can be used for anomaly detection like any other density estimator

First, we build and compile the model (for the HPC data)

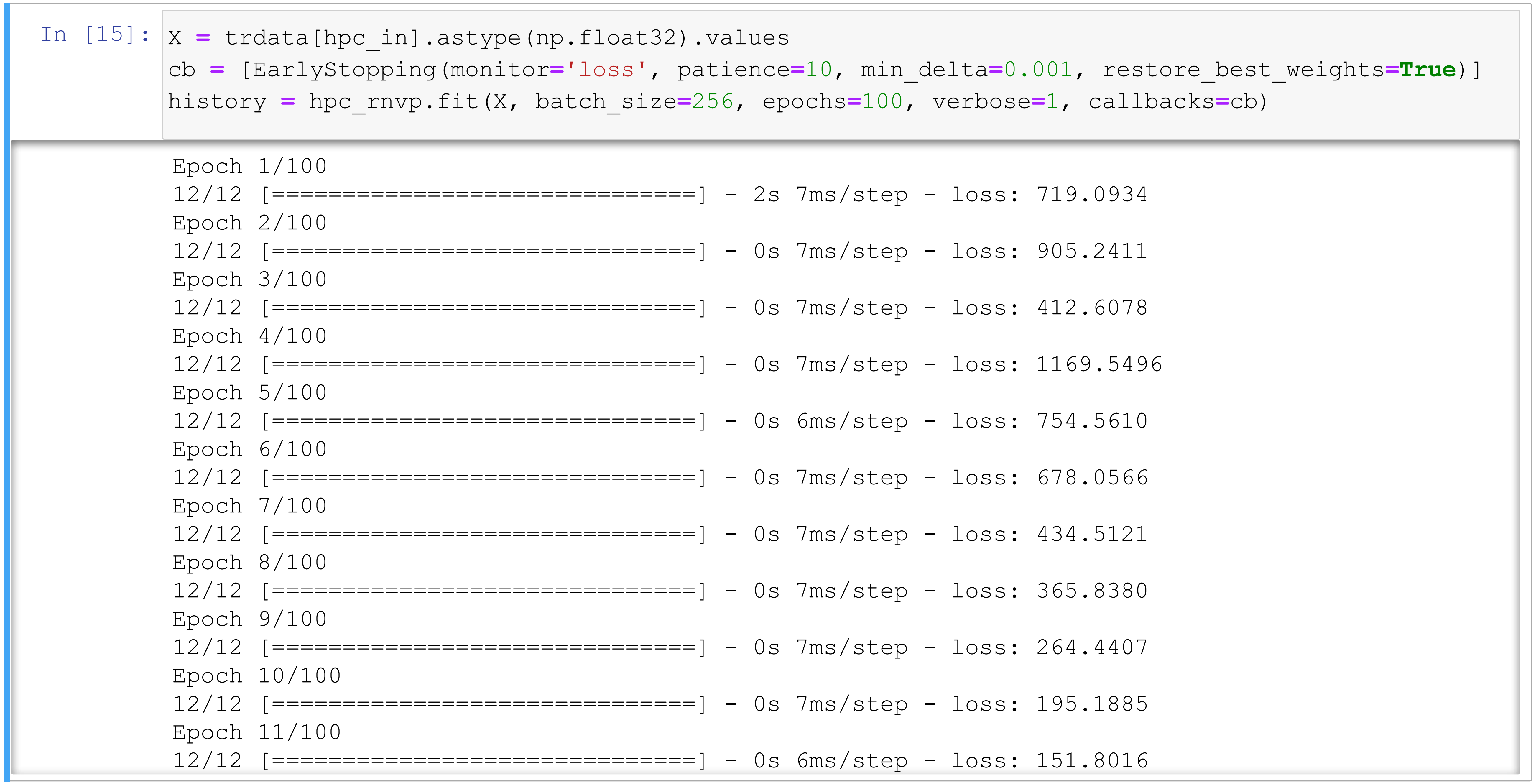
```
In [14]: input_shape = len(hpc_in)
         hpc_rnvp = nn.RealNVP(input_shape=input_shape,
                         num_coupling=6, units_coupling=32, depth_coupling=1, reg_coupling=0.01)
         hpc_rnvp.compile(optimizer='Adam')
```

We chose a simpler architecture this time

- With RNVP, dealing with higher dimensional data has actually some advantage
- In particular, we have richer input for the s and t functions
  - In the "moons" dataset, s and t had 2/2 = 1 input feature
  - Now we have 159/2 = 79--80 features

RNVP for Anomaly Detection

Then we perform training as usual



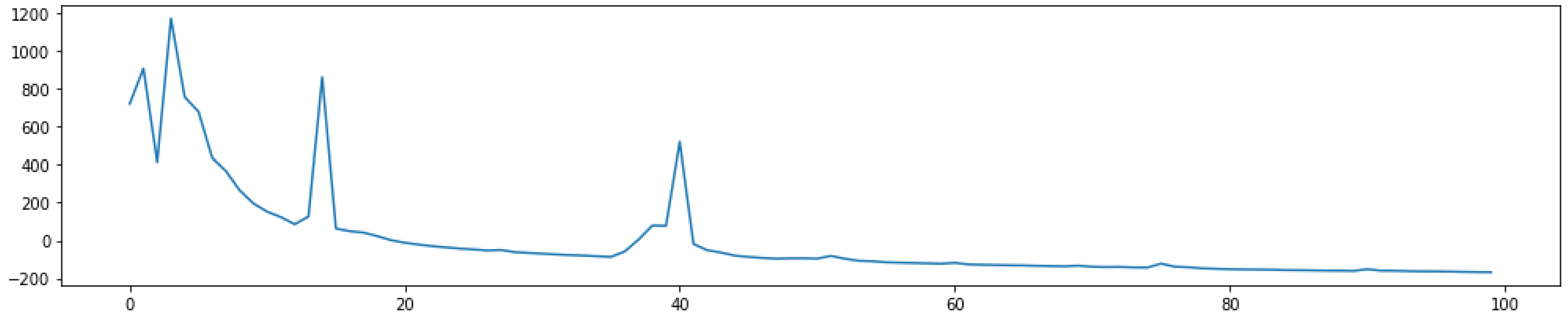
```
In [15]: X = trdata[hpc_in].astype(np.float32).values
         cb = [EarlyStopping(monitor='loss', patience=10, min_delta=0.001, restore_best_weights=True)]
         history = hpc_rnvp.fit(X, batch_size=256, epochs=100, verbose=1, callbacks=cb)
```

```
Epoch 1/100
12/12 [==============================] - 2s 7ms/step - loss: 719.0934
Epoch 2/100
12/12 [==============================] - 0s 7ms/step - loss: 905.2411
Epoch 3/100
12/12 [==============================] - 0s 7ms/step - loss: 412.6078
Epoch 4/100
12/12 [==============================] - 0s 7ms/step - loss: 1169.5496
Epoch 5/100
12/12 [==============================] - 0s 6ms/step - loss: 754.5610
Epoch 6/100
12/12 [==============================] - 0s 7ms/step - loss: 678.0566
Epoch 7/100
12/12 [==============================] - 0s 7ms/step - loss: 434.5121
Epoch 8/100
12/12 [==============================] - 0s 7ms/step - loss: 365.8380
Epoch 9/100
12/12 [==============================] - 0s 7ms/step - loss: 264.4407
Epoch 10/100
12/12 [==============================] - 0s 7ms/step - loss: 195.1885
Epoch 11/100
12/12 [==============================] - 0s 6ms/step - loss: 151.8016
```

RNVP for Anomaly Detection

Here is the loss evolution over time

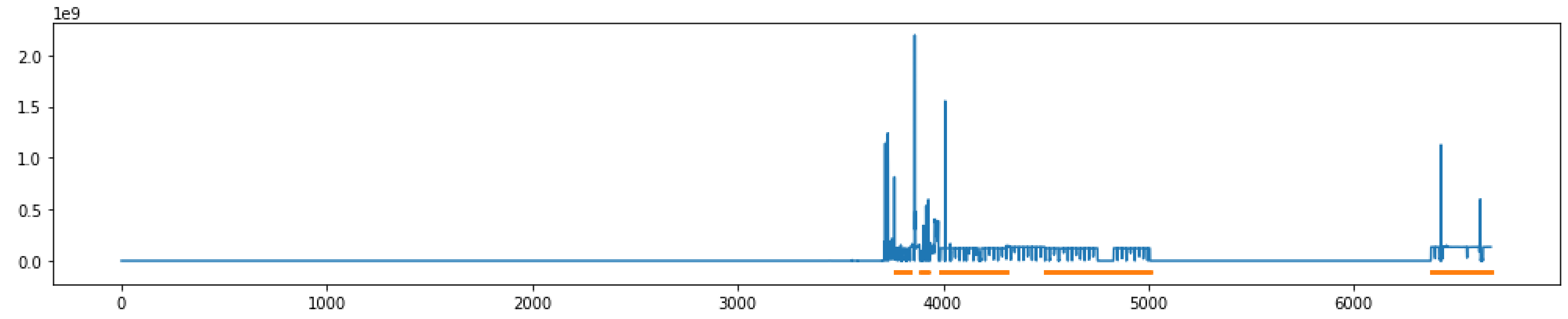
```
In [16]: nn.plot_training_history(history, figsize=figsize)
```



RNVP for Anomaly Detection

Then we can generate a signal as usual

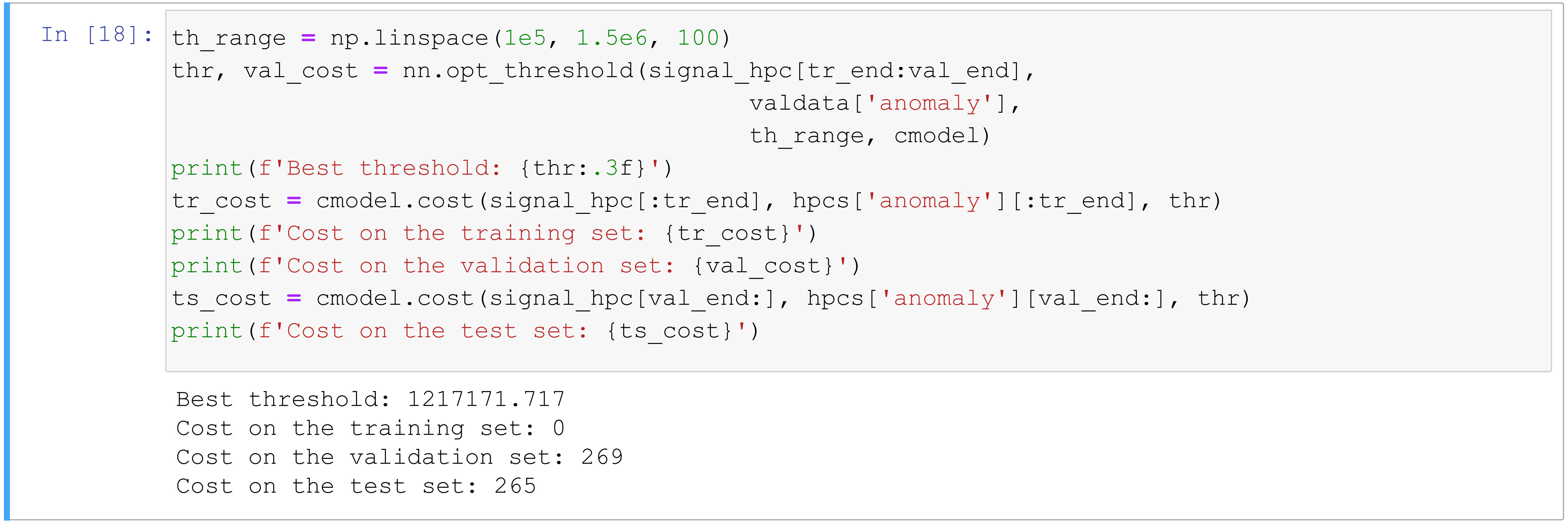
```
In [17]: X = hpcs[hpc_in].astype(np.float32).values
         signal_hpc = pd.Series(index=hpcs.index, data=-hpc_rnvp.score_samples(X))
         nn.plot_signal(signal_hpc, hpc_labels, figsize=figsize)
```



- The signal is very similar to that of KDE (not a surprise)

RNVP for Anomaly Detection

Finally, we can tune the threshold



```
In [18]: th_range = np.linspace(1e5, 1.5e6, 100)
         thr, val_cost = nn.opt_threshold(signal_hpc[tr_end:val_end],
                                          valdata['anomaly'],
                                          th_range, cmodel)
         print(f'Best threshold: {thr:.3f}')
         tr_cost = cmodel.cost(signal_hpc[:tr_end], hpcs['anomaly'][:tr_end], thr)
         print(f'Cost on the training set: {tr_cost}')
         print(f'Cost on the validation set: {val_cost}')
         ts_cost = cmodel.cost(signal_hpc[val_end:], hpcs['anomaly'][val_end:], thr)
         print(f'Cost on the test set: {ts_cost}')
```

```
Best threshold: 1217171.717
Cost on the training set: 0
Cost on the validation set: 269
Cost on the test set: 265
```

- Once again, the performance is on par with KDE
- ...But we have better support for high-dimensional data!