# Deriving the Normal Distribution Inverse CDF

using an Inverting Neural Network

John Morrow

## 1 Introduction

While it is straightforward to invert a function like $y = mx$ to produce the inverse, $x = \frac{y}{m}$, some functions can't be easily inverted. One such function is the cumulative distribution function (CDF) of the normal probability distribution (Equation 1), where neither the CDF nor the inverse CDF (quantile function) can be expressed in a closed form. This article presents a method for inverting a function using a neural network regardless of whether the problem can be solved analytically.

$$\Phi(x) = \int_{-\infty}^{x} \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}t^2} \qquad \text{(with } \mu = 0,\ \sigma = 1\text{)} \tag{1}$$

A previous article presented the integrating neural network as a method for integrating a function with no analytical solution. As an example, the CDF of the normal probability function was derived by training a neural network to produce the integral of the probability density function (PDF). In this article, the quantile function will be derived by training a neural network to invert the CDF. Then random samples from the normal distribution will be generated from the quantile function through inverse transform sampling.

For reference, plots of the PDF, CDF, and quantile functions are presented in Figure 1 and Figure 2.

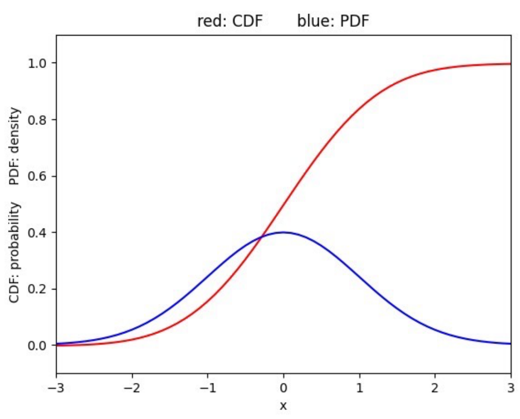

Figure 1: **Normal distribution: PDF and CDF**

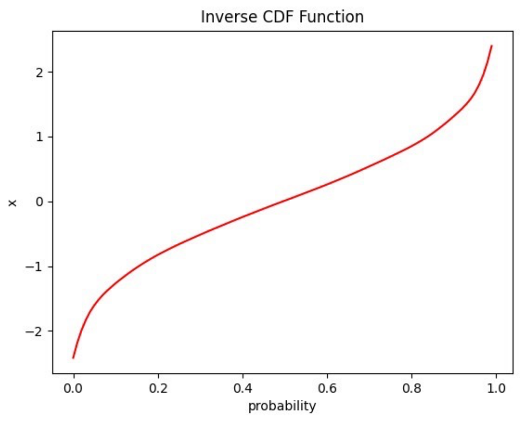

Figure 2: **Normal distribution: Quantile**

## 2 Inverting neural network

An inverting neural network is trained to produce the inverse of a function. Inverting a function $y = f(x)$ is accomplished by training the inverting neural network such that *the output of the function, with input supplied by the network's output, is equal to the network's input, resulting in the network's output becoming the inverse of the function, i.e.,* $x = f^{-1}(y)$.

### 2.1 Neural network training

Following are the steps of the training procedure:

1. Apply a training point, $y_i$, to the input of the neural network:

   (Note: The neural network model, $x = nn\_model(y)$, comprises a single input, $y$, multiple hidden layers, and a single output, $x$. The variables $\hat{x}_i$ and $\hat{y}_i$ in the following equations are estimates based on the response of the untrained neural network.)

$$\hat{x}_i = nn\_model(y_i) \tag{2}$$

2. Apply the output of the neural network, $\hat{x}_i$, to the input of the function to be inverted, $y = f(x)$:

$$\hat{y}_i = f(\hat{x}_i) \tag{3}$$

3. Train the neural network with a loss function (loss 2 in Section 2.2) that forces the following relationship:

$$\hat{y}_i = y_i \tag{4}$$

Then, after the neural network is trained:

$$nn\_model(y) = x \tag{5}$$

Substituting, $y = f(x)$:

$$nn\_model(f(x)) = x \tag{6}$$

Equation 6 is the definition of the inverse function, therefore:

$$nn\_model(y) \equiv f^{-1}(y) \tag{7}$$

## 2.2 Neural network loss function

Typically, a neural network is trained with pairs of known input and output data. The training input data is presented to the neural network, and the resulting output is compared to the training output data using a loss function. The loss returned by this function is used via backpropagation to adjust the network's weights to reduce the loss. An inverting neural network uses a custom loss function to constrain the neural network to produce a response that complies with the inverse function, $x = f^{-1}(y)$.

For the normal distribution's CDF, $f(x)$ is given by Equation 1. As there is no closed-form solution for this equation, the trained neural network from the previous article is used to supply this function. It is represented by $cdf = nn\_cdf(x)$, where $x$ is the network's input and $cdf$ is the CDF output.

The loss function for the inverting neural network, Figure 3, has two components. Loss 2, described in the training procedure above (Section 2.1), forces the output of the neural network to comply with the inverse function, $x = f^{-1}(y)$. Loss 1, with a single point condition of $(x\_init, y\_init)$, stabilizes the training process.

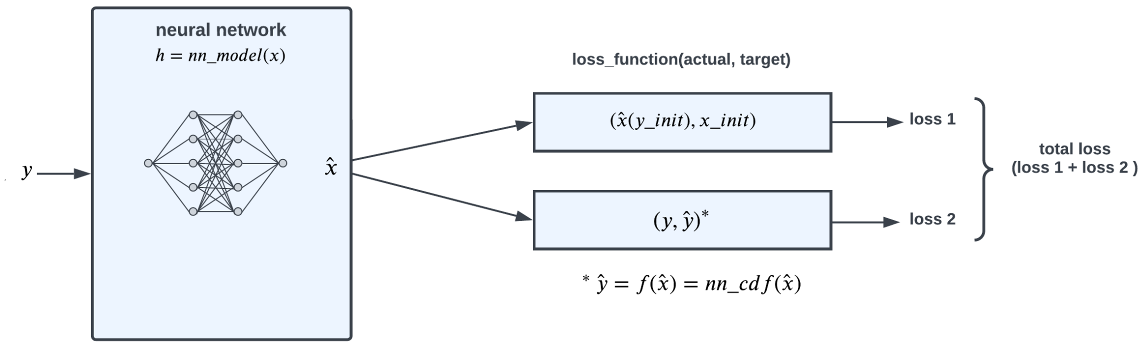
neural network
$h = nn\_model(x)$
$y$
$\hat{x}$
loss_function(actual, target)
$(\hat{x}(y\_init), x\_init)$
$(y, \hat{y})^*$
$^* \hat{y} = f(\hat{x}) = nn\_cdf(\hat{x})$
loss 1
loss 2
total loss
(loss 1 + loss 2 )

Figure 3: **Loss function**

## 3 Inverting neural network implementation

Following is the Python code for the inverting neural network implementation of the normal distribution quantile function. The complete code is available here.

### 3.1 Neural network model definition

The inverting neural network model has two fully-connected hidden layers, each with 256 neurons. There is a single input for domain points and a single output for the corresponding values from the inverted function.

Listing 1: TensorFlow neural network model

```
# Hyperparameters
batch_size = 1
epochs = 500
optimizer = Adam(learning_rate=0.001)
weight_init = RandomNormal()

# Build model
inputs = tf.keras.Input(shape=(1,))
x = layers.Dense(256, activation='gelu', name='H1', kernel_initializer=weight_init,\
               kernel_regularizer=None)(inputs)
x = layers.Dense(256, activation='gelu', name='H2', kernel_initializer=weight_init,
               kernel_regularizer=None)(x)
output = layers.Dense(1, activation='linear', name='Out', kernel_initializer=weight_init)
    (x)
model = tf.keras.Model(inputs, output)
```

## 3.2 Initialization

The neural network model trained with the CDF function, $cdf = nn_{cdf}(x)$, is loaded in line 3. The model is used in the definition of the function to be inverted in lines 6-8.

The $y_i$ training points from Section 2.1 for loss 2 are defined in lines 11-15 below. The order of these points is randomly shuffled on line 17 to promote stable training of the neural network. The condition for loss 1 is defined in lines 21-22.

Listing 2: Initialization

```
1  # load trained CDF function model
2  dir_path = '[insert path here]'
3  cdf_model = load_model(dir_path + "normal-cdf-model.h5", compile=False)
4
5  # function to be inverted
6  def f_tbi(x):
7      prob = cdf_model(x)
8      return prob
9
10 # define domain of inverse function
11 y_coloc = np.arange(0.1, 0.9, 0.01)
12 y_coloc_densel = np.arange(0.0, 0.1, 0.001)
13 y_coloc_denser = np.arange(0.9, 1.0, 0.001)
14 y_coloc = np.append(y_coloc, y_coloc_densel)
15 y_coloc = np.append(y_coloc, y_coloc_denser)
16 rng = np.random.default_rng()
17 rng.shuffle(y_coloc)
18
19 # initial condition (point on inverse function)
20 # (stabilizes training)
21 x_init = np.array([0.0])
22 y_init = cdf_model.predict(x_init)
```

## 3.3 Batch training steps

Listing 3 is the training step function applied to each batch of training points. The total loss (the sum of loss 1 and loss 2) in line 21 is used to update the neural network's weights via gradient descent (lines 23-27). Each training epoch includes multiple batches, which collectively use all the training points in model updates.

Line 9 produces the neural network's response to the $y\_init$ initial condition. The response is compared to the corresponding initial condition, $x\_init$, producing loss 1 (line 10).

Line 13 produces the network's response to training point $y_i$ (Equation 2). The network's response is applied to the input of the function to be inverted in line 14 (Equation 3), and lines 16-17 compare the output of the function to be inverted to the training point $y_i$ (Equation 4), producing loss 2.

Listing 3: Batch training step

```
# training step function for each batch
def step(y, y_init, x_init):
    y = tf.convert_to_tensor(y)
    y = tf.reshape(y, [batch_size, 1]) # required by keras input
    y = tf.Variable(y)
    with tf.GradientTape(persistent=False) as tape: #false - no higher gradients

        #model_loss1: single point condition
        pred_init = model(y_init)
        model_loss1 = math_ops.squared_difference(pred_init, x_init)

        # model_loss2: collocation points
        pred_x = model(y)
        func = f_tbi(pred_x)
        func = tf.cast(func, tf.float64)
        residual = func - y
        model_loss2 = K.mean(math_ops.square(residual), axis=-1)
        model_loss2 = tf.cast(model_loss2, tf.float32)

        #total loss
        model_loss = model_loss1 + model_loss2 * 10

        trainable = model.trainable_variables
        model_gradients = tape.gradient(model_loss, trainable)

        # Update model
        optimizer.apply_gradients(zip(model_gradients, trainable))
        return np.mean(model_loss)
```

## 4 Results

Figure 4 is the quantile response (red trace) from the output of the trained quantile neural network. As verification of the accuracy of the results, the quantile response from the *ppf.cdf* function in the Python SciPy library is included (green dots).

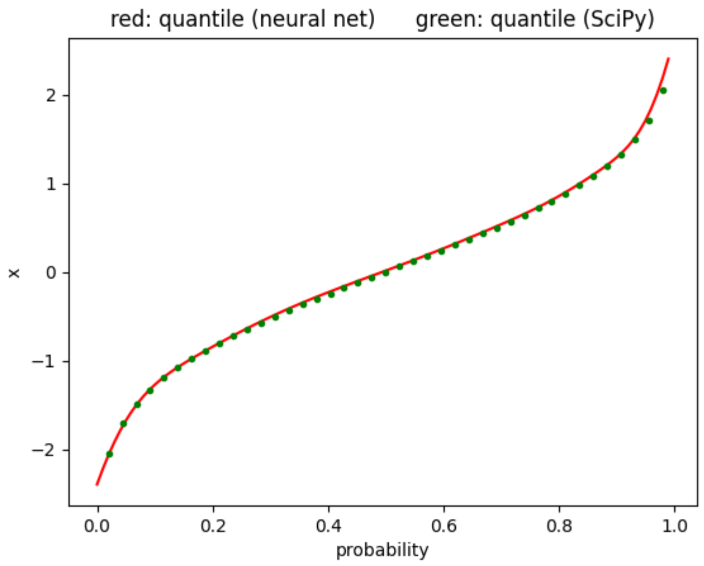

Figure 4: **Trained quantile neural network output**

Figure 5 is the loss from the total loss function v.s. epoch logged during the training process.

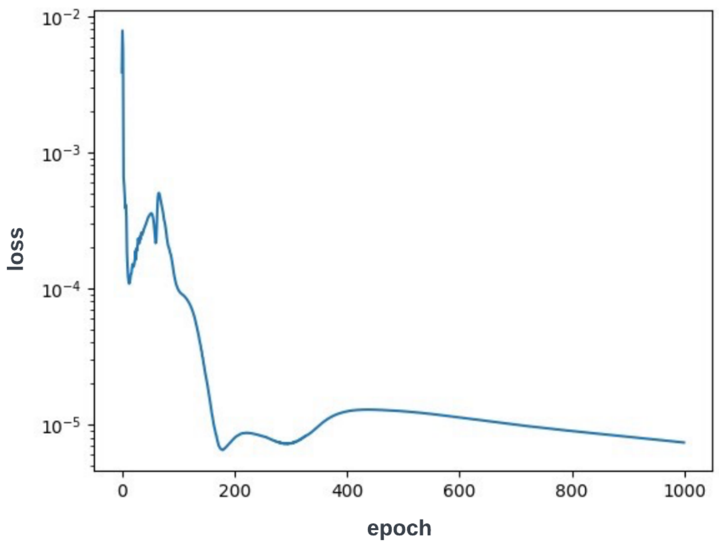

Figure 5: **Training loss**

Figure 6 is a histogram of normally distributed samples produced with inverse transform sampling. Sampling is accomplished by applying uniformly distributed samples between $0$ and $1$ to the probability input of the quantile neural network, resulting in normally distributed samples from the network's output. The histogram comprises $500$ bins and $30,000$ samples. For reference, the red trace is the PDF of the normal distribution.

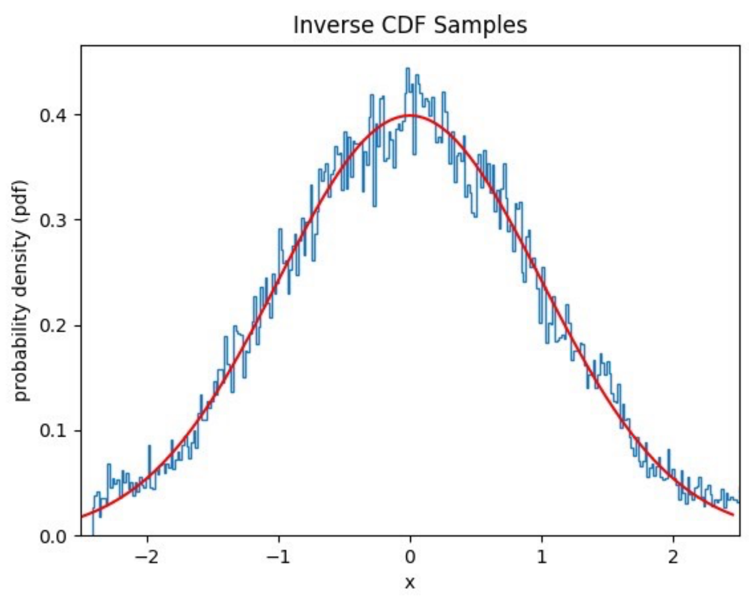

Figure 6: **Normally distributed samples**

## 5 Conclusion

This article demonstrates a method for training a neural network to invert a function by using a custom loss function. It was shown that the method is also effective for inverting functions with no closed-form solution. This was demonstrated by training a neural network to provide the quantile function of the normal distribution by inverting the CDF function.