Deriving the CDF of the Normal Distribution

using an Integrating Neural Network

John Morrow

1 Introduction

This article presents a method for training a neural network to derive the integral of a function. The technique works not only with analytically-solvable integrals but also with integrals that do not have a closed-form solution and are typically solved by numerical methods. An example is the normal distribution's cumulative density function (CDF). Equation 1 is this distribution's probability density function (PDF), and Equation 2 is its CDF, the integral of the PDF. Figure 1 is a plot of these functions. After being trained, the resulting network can serve as a stand-alone function generator that delivers points on the CDF curve given points from the domain of the distribution's PDF.

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

$$\Phi(x) = \int_{-\infty}^{x} \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}t^2} \qquad \text{CDF} \tag{2}$$

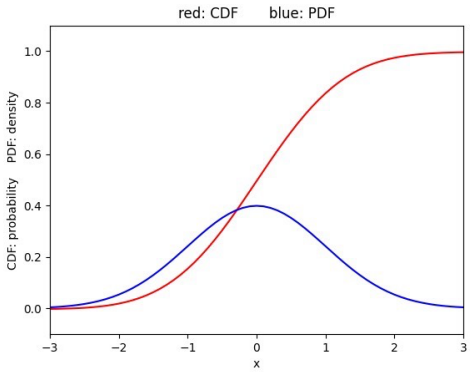

Figure 1: PDF and CDF of the normal distribution

2 Integrating neural network

An integrating neural network is trained to produce the integral of a function $y = f(x)$. Expressed in terms of the network's input and output:

$$h = \int f(x) \tag{3}$$

where h and x are the network's output and input, respectively. For the normal distribution, $f(x)$ is given by Equation 1, the PDF of the distribution.

Integration of the function is accomplished by training the neural network such that *the derivative of the network's output is equal to the function's output, resulting in the network's output becoming the integral of the function*.

2.1 Neural network training

Following are the steps of the training procedure:

1. Apply a training point, x_i, to the function $y = f(x)$:

$$y_i = f(x_i) \tag{4}$$

2. Also apply x_i to the input of the neural network:

   (Note: The neural network model comprises a single input, x, two hidden layers, and a single output, h, and is represented by $h(x) = nn\_model(x)$.)

$$h_i = nn\_model(x_i) \tag{5}$$

3. Take the derivative of h_i:

$$g_i = \frac{d\,h_i}{d\,x_i} \tag{6}$$

   (Differentiation is provided in TensorFlow and PyTorch via their automatic differentiation function. In this article, the neural network is developed with TensorFlow GradientTape.)

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

$$g_i = y_i \tag{7}$$

Then after the neural network is trained, since $g = y$, and substituting g and y from Equation 6 and Equation 4, respectively:

$$\frac{d\,h}{d\,x} = f(x) \tag{8}$$

Integrating both sides of Equation 8 confirms that the neural network's output is the integral of the function $f(x)$:

$$h = \int f(x)\,d\,x + C \tag{9}$$

where C is the constant of integration.

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 integrating neural network uses a custom loss function to constrain the neural network to produce an output that complies with the output of the integrated function.

The loss function for the integrating neural network, Figure 2, has three components. Loss 2, described in the training procedure above (Section 2.1), forces the output of the neural network to comply with the integral of $f(x)$.

Loss 3 forces the neural network to comply with the initial condition $h(x\_init2) = h\_init2$. For the CDF model, this condition is $h(-10) = 0$, which sets $C = 0$ (Equation 9). For the purpose of this model, the responses of the PDF and CDF at $x = -10$ approximate the responses at $x = -\infty$.

Setting the initial condition in Loss 3 to $h(-\infty) = 0$ also simplifies the CDF calculation. Expanding the definite integral of Equation 2:

$$\Phi(x) = \left[\int_{-\infty}^{x} \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}t^2}\right]_{t=x} - \left[\int_{-\infty}^{x} \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}t^2}\right]_{t=-\infty} \tag{10}$$

The initial condition, $h(-\infty) = 0$, means that the second term equals zero, and the output of the trained neural network is the value of the CDF for the corresponding x input:

$$\Phi(x) = \left[\int_{-\infty}^{x} \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}t^2}\right]_{t=x} \tag{11}$$

Loss 1, with condition $h(10) = 1$, stabilizes the training process for points near the right tail of the

distribution. For the purpose of this model, the responses of the PDF and CDF at $x = 10$ approximate the responses at $x = \infty$.

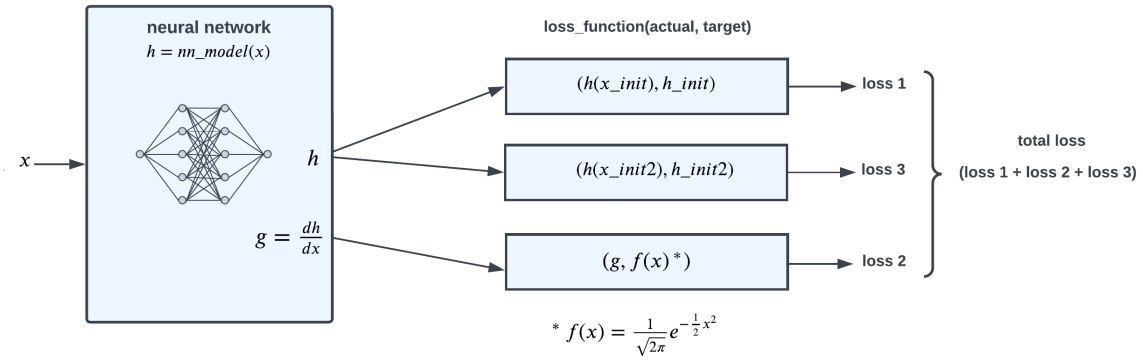

Figure 2: Loss function

3 Integrating neural network implementation

Following is the Python code for the integrating neural network implementation of the normal distribution CDF. The complete code is available here.

3.1 Neural network model definition

The neural network for the PINN has two fully-connected hidden layers, each with 512 neurons. There is a single input for domain points and a single output for the corresponding integral values.

Listing 1: TensorFlow neural network model

```
# Hyperparameters
batch_size = 1
epochs = 1000
optimizer = Adam(learning_rate=0.001)
weight_init = RandomNormal()

# Build model
inputs = tf.keras.Input(shape=(1,))
x = layers.Dense(512, activation='gelu', kernel_initializer=weight_init,
                 kernel_regularizer=None)(inputs)
x = layers.Dense(512, activation='gelu', kernel_initializer=weight_init,
                 kernel_regularizer=None)(x)
output = layers.Dense(1, activation='linear', kernel_initializer=weight_init)(x)
model = tf.keras.Model(inputs, output)
```

3.2 Initialization

The x_i training points from Section 2.1 for loss 2 are defined on line 9, below. The order of these points is randomly shuffled on line 11 to promote stable training of the neural network. On line 12 the points are applied to the PDF as described in Equation 4. The initial conditions for loss 1 and loss 3 are defined in lines 15-16 and lines 19-20, respectively.

Listing 2: Initialization

```
1  # function to be integrated (normal distribution PDF)
2  mu = 0.0
3  sigma = 1.0
4  def f_tbi(x_coloc):
5      return (1 / (sigma * np.sqrt(2 * math.pi)))\
6             * np.exp(-0.5 * ((x_coloc - mu ) / sigma)**2)
7
8  # create colocation points for function to be integrated
9  x_coloc = np.arange(-10, 10, 0.2) # define domain
10 rng = np.random.default_rng()
11 rng.shuffle(x_coloc)
12 y_coloc = f_tbi(x_coloc)
13
14 # initial condition for right tail stability
15 x_init = np.array([10.0])
16 h_init = np.array([1.0])
17
18 # integral(f(x)) initial condition
19 x_init2 = np.array([-10.0])
20 h_init2 = np.array([0.0])
```

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, loss 2, and loss 3) in line 24 is used to update the neural network's weights via gradient descent (lines 26 -30). 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 x_{init} initial condition. The response is compared to the corresponding initial condition, h_{init}, producing loss 1 (line 10).

Similarly, Line 13 produces the network's response to the x_{init2} initial condition. The response is compared to the corresponding initial condition, h_{init2}, producing loss 3 (line 14).

Line 17 produces the network's response to training point x_i (Equation 5). Line 18 extracts the gradient of the response (Equation 6), and lines 19–20 compare the gradient to $f(x_i)$ (Equation 7), producing loss 2.

Listing 3: Batch training step

```
# training step function for each batch
def step(x_co, y_co, x_init, h_init, x_init2, h_init2):
    x_co = tf.convert_to_tensor(x_co)
    x_co = tf.reshape(x_co, [batch_size, 1]) # required by keras input
    x_co = tf.Variable(x_co, name='x_co')
    with tf.GradientTape(persistent=True) as tape:

        # model_loss1: initial condition h_init @ x_init
        pred_init = model(x_init)
        model_loss1 = math_ops.squared_difference(pred_init, h_init)

        # model_loss3: initial condition h_init2 @ x_init2
        pred_init2 = model(x_init2)
        model_loss3 = math_ops.squared_difference(pred_init2, h_init2)

        # model_loss2: collocation points
        pred_h = model(x_co)
        dfdx = tape.gradient(pred_h, x_co)
        residual = dfdx - y_co
        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_loss3 + 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 3 shows the CDF response (red trace) from the output of the trained neural net. As verification of the accuracy of the results, the CDF response from the *norm.cdf* function in the Python SciPy library is included (green dots).

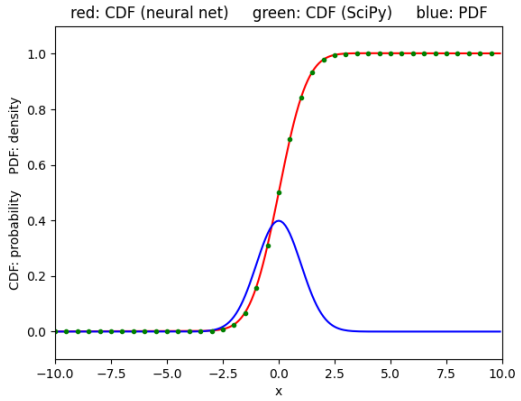

Figure 3: Trained CDF neural network output

Figure 4 is the loss from the total loss function v.s. epoch logged during the training process.

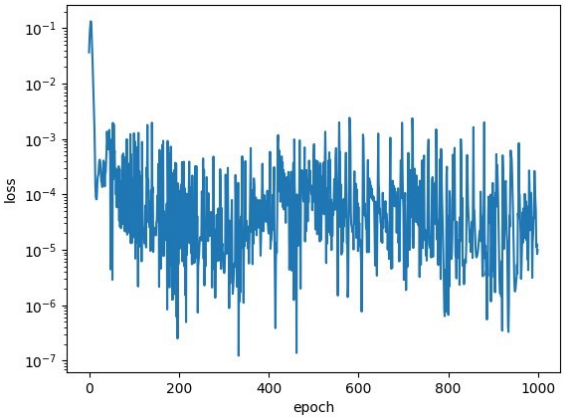

Figure 4: Training loss

5 Conclusion

This article demonstrates a method for training a neural network to integrate a function by using a custom loss function and automatic differentiation. Specifically, a neural network is trained to successfully integrate the PDF of the normal distribution to produce the CDF.