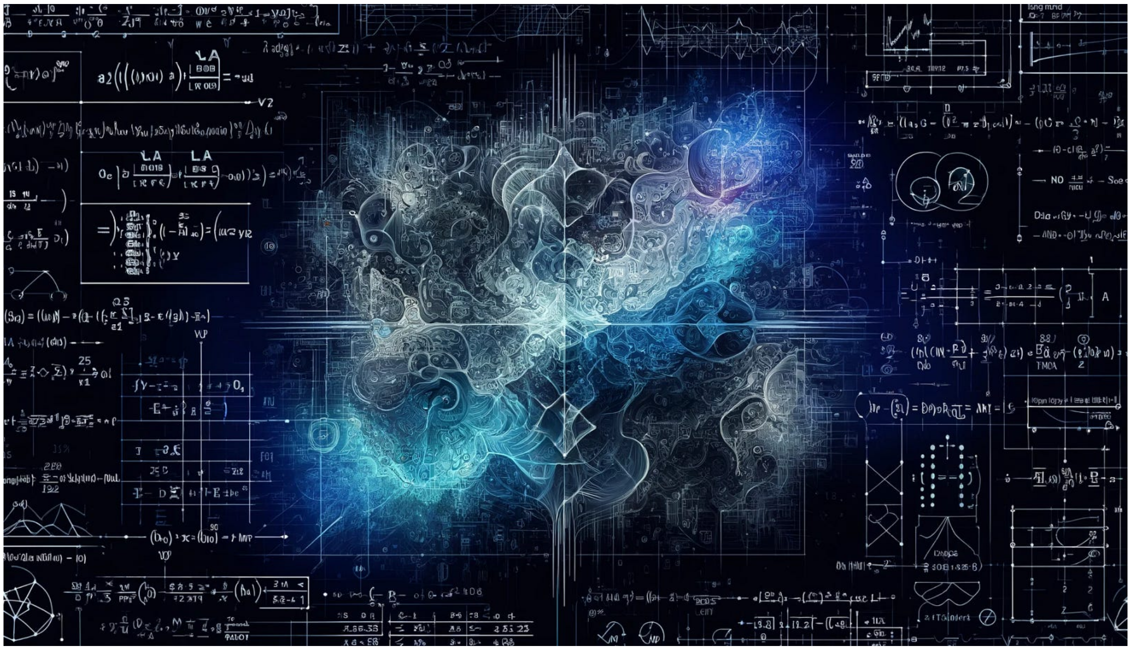
Batch Normalization

May 15, 2024

1 Batch Normalization From Scratch in Python

By Cristian Leo



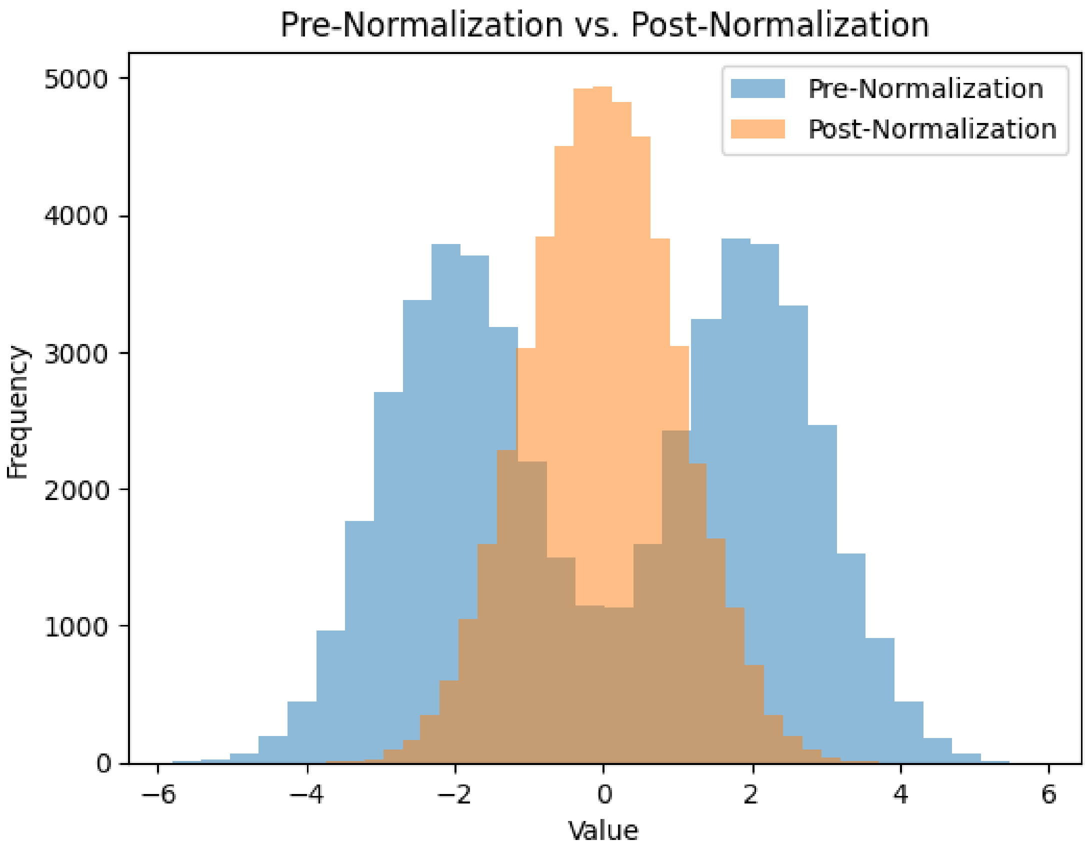
[1]:
```
import numpy as np
import matplotlib.pyplot as plt

# Custom classes (built from scratch)
from src.model import WeightInitializer
from src.trainer import PlotManager, LSTMTrainer, TimeSeriesDataset
```

[2]:
```
class BatchNorm:
    def __init__(self, hidden_size):
        self.hidden_size = hidden_size
        self.x = None
        self.gamma = np.ones((hidden_size, 1))
        self.beta = np.zeros((hidden_size, 1))
```

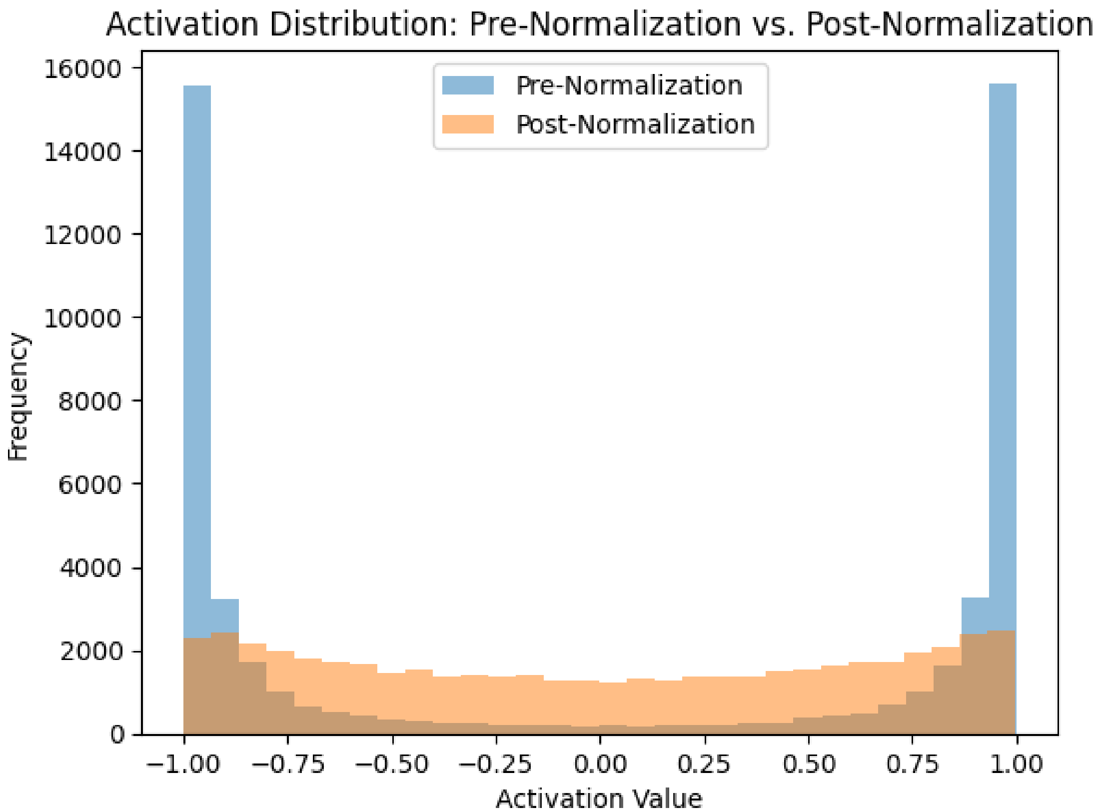
```
    def forward(self, x):
        self.x = x
        self.mu = np.mean(x, axis=0)
        self.var = np.var(x, axis=0)
        self.x_norm = (x - self.mu) / np.sqrt(self.var + 1e-6)
        out = self.gamma * self.x_norm + self.beta
        return out

    def backward(self, dout):
        N = dout.shape[0]
        dgamma = np.sum(dout * self.x_norm, axis=0)
        dbeta = np.sum(dout, axis=0)
        dx_norm = dout * self.gamma
        dvar = np.sum(dx_norm * (self.x - self.mu) * -0.5 * (self.var + 1e-8)**-1.5, axis=0)
        dmu = np.sum(dx_norm * -1 / np.sqrt(self.var + 1e-8), axis=0) + dvar * np.mean(-2 * (self.x - self.mu), axis=0)
        dx = dx_norm / np.sqrt(self.var + 1e-8) + dvar * 2 * (self.x - self.mu) / N + dmu / N
        return dx, dgamma, dbeta

    def plot_batch_norm(self):
        # Compute the histograms of the pre-normalized and post-normalized data
        pre_norm_hist, pre_norm_bins = np.histogram(self.x, bins=30)
        post_norm_hist, post_norm_bins = np.histogram(self.x_norm, bins=30)

        # Plot the pre-normalized data
        plt.hist(pre_norm_bins[:-1], pre_norm_bins, weights=pre_norm_hist, alpha=0.5, label='Pre-Normalization')

        # Plot the post-normalized data
        plt.hist(post_norm_bins[:-1], post_norm_bins, weights=post_norm_hist, alpha=0.5, label='Post-Normalization')

        # Add labels, a title, and a legend
        plt.xlabel('Value')
        plt.ylabel('Frequency')
        plt.title('Pre-Normalization vs. Post-Normalization')
        plt.legend()

        # Display the plot
        plt.show()

    def plot_activation_distribution(self, activation_function):
        # Compute the activation before batch normalization
        pre_norm_activation = activation_function(self.x)
```

```
        # Compute the activation after batch normalization
        post_norm_activation = activation_function(self.x_norm)

        # Compute the histograms of the pre-normalized and post-normalized␣
↪activations
        pre_norm_hist, pre_norm_bins = np.histogram(pre_norm_activation,␣
↪bins=30)
        post_norm_hist, post_norm_bins = np.histogram(post_norm_activation,␣
↪bins=30)

        # Plot the pre-normalized activation
        plt.hist(pre_norm_bins[:-1], pre_norm_bins, weights=pre_norm_hist,␣
↪alpha=0.5, label='Pre-Normalization')

        # Plot the post-normalized activation
        plt.hist(post_norm_bins[:-1], post_norm_bins, weights=post_norm_hist,␣
↪alpha=0.5, label='Post-Normalization')

        # Add labels, a title, and a legend
        plt.xlabel('Activation Value')
        plt.ylabel('Frequency')
        plt.title('Activation Distribution: Pre-Normalization vs.␣
↪Post-Normalization')
        plt.legend()

        # Display the plot
        plt.show()
```

[3]:
```
# Create an instance of BatchNorm with a hypothetical hidden size, for example,␣
↪50.
bn_layer = BatchNorm(50)

# Pass some data from a mixture of Gaussians through the BatchNorm layer
# Generate data from two different normal distributions and combine them
data1 = np.random.normal(-2, 1, size=(500, 50))
data2 = np.random.normal(2, 1, size=(500, 50))
data = np.concatenate([data1, data2])

bn_layer.forward(data.T)

# Now, call the visualization methods
bn_layer.plot_batch_norm()
```



[4]:
```
# Create an instance of BatchNorm with a hypothetical hidden size, for example,␣
↪50.
bn_layer = BatchNorm(50)

bn_layer.forward(data.T)

# Now, call the visualization methods
bn_layer.plot_activation_distribution(np.tanh)  # Use the tanh activation␣
↪function
```



[5]:
```
class LSTM:
    """
    Long Short-Term Memory (LSTM) network.

    Parameters:
    - input_size: int, dimensionality of input space
    - hidden_size: int, number of LSTM units
    - output_size: int, dimensionality of output space
    - init_method: str, weight initialization method (default: 'xavier')
    """
    def __init__(self, input_size, hidden_size, output_size,␣
↪init_method='xavier'):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.weight_initializer = WeightInitializer(method=init_method)

        # Initialize weights
        self.wf = self.weight_initializer.initialize((hidden_size, hidden_size␣
↪+ input_size))
```

```python
        self.wi = self.weight_initializer.initialize((hidden_size, hidden_size␣
↪+ input_size))
        self.wo = self.weight_initializer.initialize((hidden_size, hidden_size␣
↪+ input_size))
        self.wc = self.weight_initializer.initialize((hidden_size, hidden_size␣
↪+ input_size))

        # Initialize biases
        self.bf = np.zeros((hidden_size, 1))
        self.bi = np.zeros((hidden_size, 1))
        self.bo = np.zeros((hidden_size, 1))
        self.bc = np.zeros((hidden_size, 1))

        # Initialize output layer weights and biases
        self.why = self.weight_initializer.initialize((output_size,␣
↪hidden_size))
        self.by = np.zeros((output_size, 1))

        # Initialize batch normalization layers
        self.bn_f = BatchNorm(hidden_size)
        self.bn_i = BatchNorm(hidden_size)
        self.bn_o = BatchNorm(hidden_size)
        self.bn_c = BatchNorm(hidden_size)


    @staticmethod
    def sigmoid(z):
        """
        Sigmoid activation function.

        Parameters:
        - z: np.ndarray, input to the activation function

        Returns:
        - np.ndarray, output of the activation function
        """
        return 1 / (1 + np.exp(-z))

    @staticmethod
    def dsigmoid(y):
        """
        Derivative of the sigmoid activation function.

        Parameters:
        - y: np.ndarray, output of the sigmoid activation function

        Returns:
```

```
        - np.ndarray, derivative of the sigmoid function
        """
        return y * (1 - y)

    @staticmethod
    def dtanh(y):
        """
        Derivative of the hyperbolic tangent activation function.

        Parameters:
        - y: np.ndarray, output of the hyperbolic tangent activation function

        Returns:
        - np.ndarray, derivative of the hyperbolic tangent function
        """
        return 1 - y * y

    def forward(self, x):
        """
        Forward pass through the LSTM network.

        Parameters:
        - x: np.ndarray, input to the network

        Returns:
        - np.ndarray, output of the network
        - list, caches containing intermediate values for backpropagation
        """
        caches = []
        h_prev = np.zeros((self.hidden_size, 1))
        c_prev = np.zeros((self.hidden_size, 1))
        h = h_prev
        c = c_prev

        for t in range(x.shape[0]):
            x_t = x[t].reshape(-1, 1)
            combined = np.vstack((h_prev, x_t))

            f = self.sigmoid(self.bn_f.forward(np.dot(self.wf, combined) + self.
↪bf))
            i = self.sigmoid(self.bn_i.forward(np.dot(self.wi, combined) + self.
↪bi))
            o = self.sigmoid(self.bn_o.forward(np.dot(self.wo, combined) + self.
↪bo))
            c_ = np.tanh(self.bn_c.forward(np.dot(self.wc, combined) + self.bc))
```

```
            c = f * c_prev + i * c_
            h = o * np.tanh(c)

            cache = (h_prev, c_prev, f, i, o, c_, x_t, combined, c, h)
            caches.append(cache)

            h_prev, c_prev = h, c

        y = np.dot(self.why, h) + self.by
        return y, caches

    def backward(self, dy, caches, clip_value=1.0):
        """
        Backward pass through the LSTM network.

        Parameters:
        - dy: np.ndarray, gradient of the loss with respect to the output
        - caches: list, caches from the forward pass
        - clip_value: float, value to clip gradients to (default: 1.0)

        Returns:
        - tuple, gradients of the loss with respect to the parameters
        """
        dWf, dWi, dWo, dWc = [np.zeros_like(w) for w in (self.wf, self.wi, self.
↪wo, self.wc)]
        dbf, dbi, dbo, dbc = [np.zeros_like(b) for b in (self.bf, self.bi, self.
↪bo, self.bc)]
        dWhy = np.zeros_like(self.why)
        dby = np.zeros_like(self.by)

        dgamma_f, dbeta_f = np.zeros_like(self.bn_f.gamma), np.zeros_like(self.
↪bn_f.beta)
        dgamma_i, dbeta_i = np.zeros_like(self.bn_i.gamma), np.zeros_like(self.
↪bn_i.beta)
        dgamma_o, dbeta_o = np.zeros_like(self.bn_o.gamma), np.zeros_like(self.
↪bn_o.beta)
        dgamma_c, dbeta_c = np.zeros_like(self.bn_c.gamma), np.zeros_like(self.
↪bn_c.beta)

        dy = dy.reshape(self.output_size, -1)
        dh_next = np.zeros((self.hidden_size, 1))
        dc_next = np.zeros_like(dh_next)

        for cache in reversed(caches):
            h_prev, c_prev, f, i, o, c_, x_t, combined, c, h = cache
```

```
            dh = np.dot(self.why.T, dy) + dh_next
            dc = dc_next + (dh * o * self.dtanh(np.tanh(c)))

            df = dc * c_prev * self.dsigmoid(f)
            di = dc * c_ * self.dsigmoid(i)
            do = dh * self.dtanh(np.tanh(c))
            dc_ = dc * i * self.dtanh(c_)

            df, dgamma_f_, dbeta_f_ = self.bn_f.backward(df)
            di, dgamma_i_, dbeta_i_ = self.bn_i.backward(di)
            do, dgamma_o_, dbeta_o_ = self.bn_o.backward(do)
            dc_, dgamma_c_, dbeta_c_ = self.bn_c.backward(dc_)

            dgamma_f += dgamma_f_
            dbeta_f += dbeta_f_
            dgamma_i += dgamma_i_
            dbeta_i += dbeta_i_
            dgamma_o += dgamma_o_
            dbeta_o += dbeta_o_
            dgamma_c += dgamma_c_
            dbeta_c += dbeta_c_

            dcombined_f = np.dot(self.wf.T, df)
            dcombined_i = np.dot(self.wi.T, di)
            dcombined_o = np.dot(self.wo.T, do)
            dcombined_c = np.dot(self.wc.T, dc_)

            dcombined = dcombined_f + dcombined_i + dcombined_o + dcombined_c
            dh_next = dcombined[:self.hidden_size]
            dc_next = f * dc

            dWf += np.dot(df, combined.T)
            dWi += np.dot(di, combined.T)
            dWo += np.dot(do, combined.T)
            dWc += np.dot(dc_, combined.T)

            dbf += df.sum(axis=1, keepdims=True)
            dbi += di.sum(axis=1, keepdims=True)
            dbo += do.sum(axis=1, keepdims=True)
            dbc += dc_.sum(axis=1, keepdims=True)

        dWhy += np.dot(dy, h.T)
        dby += dy

        gradients = (dWf, dWi, dWo, dWc, dbf, dbi, dbo, dbc, dWhy, dby,
dgamma_f, dbeta_f, dgamma_i, dbeta_i, dgamma_o, dbeta_o, dgamma_c, dbeta_c)
```

```
        for i in range(len(gradients)):
            np.clip(gradients[i], -clip_value, clip_value, out=gradients[i])

        return gradients

    def update_params(self, grads, learning_rate):
        """
        Update the parameters of the network using the gradients.
        """
        dWf, dWi, dWo, dWc, dbf, dbi, dbo, dbc, dWhy, dby, dgamma_f, dbeta_f,␣
↪dgamma_i, dbeta_i, dgamma_o, dbeta_o, dgamma_c, dbeta_c = grads

        self.wf -= learning_rate * dWf
        self.wi -= learning_rate * dWi
        self.wo -= learning_rate * dWo
        self.wc -= learning_rate * dWc
        self.bf -= learning_rate * dbf

        self.bi -= learning_rate * dbi
        self.bo -= learning_rate * dbo
        self.bc -= learning_rate * dbc

        self.why -= learning_rate * dWhy
        self.by -= learning_rate * dby

        self.bn_f.gamma -= learning_rate * dgamma_f
        self.bn_f.beta -= learning_rate * dbeta_f
        self.bn_i.gamma -= learning_rate * dgamma_i
        self.bn_i.beta -= learning_rate * dbeta_i
        self.bn_o.gamma -= learning_rate * dgamma_o
        self.bn_o.beta -= learning_rate * dbeta_o
        self.bn_c.gamma -= learning_rate * dgamma_c
        self.bn_c.beta -= learning_rate * dbeta_c
```

[6]:

```
# Instantiate the dataset
dataset = TimeSeriesDataset('2005-01-01', '2020-12-31', train_size=0.7)
trainX, trainY, testX, testY = dataset.get_train_test()

# Plot the data
# Combine train and test data
combined = np.concatenate((trainY, testY))

# Plot the data
plt.figure(figsize=(14, 5))
plt.plot(combined, label='Google Stock Price', linewidth=2, color='dodgerblue')
plt.title('Google Stock Price', fontsize=20)
plt.xlabel('Time', fontsize=16)
```

```
plt.ylabel('Normalized Stock Price', fontsize=16)
plt.grid(True)
plt.legend(fontsize=14)
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
plt.show()
```

[7]:
```
# Reshape input to be [samples, time steps, features]
trainX = np.reshape(trainX, (trainX.shape[0], trainX.shape[1], 1))
testX = np.reshape(testX, (testX.shape[0], testX.shape[1], 1))

look_back = 1  # Number of previous time steps to include in each sample
hidden_size = 256  # Number of LSTM units
output_size = 1  # Dimensionality of the output space

lstm = LSTM(input_size=1, hidden_size=hidden_size, output_size=output_size,␣
 ↪init_method='xavier')

# Create and train the LSTM using LSTMTrainer
trainer = LSTMTrainer(lstm, learning_rate=1e-3, patience=10, verbose=True,␣
 ↪delta=0.001)
trainer.train(trainX, trainY, testX, testY, epochs=100, batch_size=32)
```

```
Epoch 1/100 - Loss: 0.57602, Val Loss: 1.29345
Epoch 11/100 - Loss: 0.00009, Val Loss: 0.41595
Epoch 21/100 - Loss: 0.00006, Val Loss: 0.38544
Epoch 31/100 - Loss: 0.00005, Val Loss: 0.38508
Early stopping
```

[9]:
```
plot_manager = PlotManager()
```

```python
# Inside your training loop
plot_manager.plot_losses(trainer.train_losses, trainer.val_losses)

# After your training loop
plot_manager.show_plots()
```