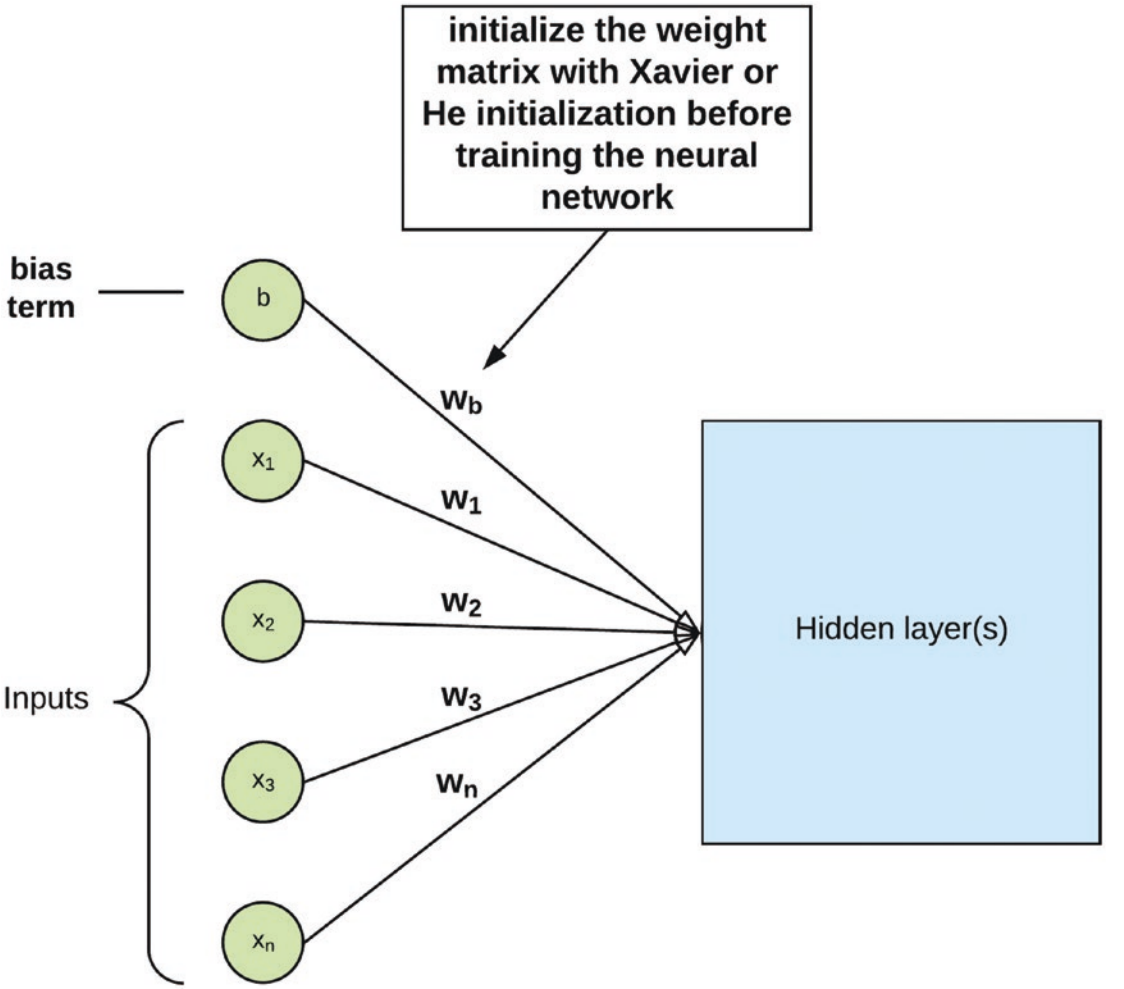

Figure 32-1. *Weight initialization*

Batch Normalization

The technique of batch normalization involves normalizing the data (to have zero mean and unit variance), as well as scaling and shifting the data batch at each layer of the neural network during the training phase. Batch normalization occurs after the affine transformation of the input matrix and their weights, but before passing the transformation into the activation function (see Figure 32-2).

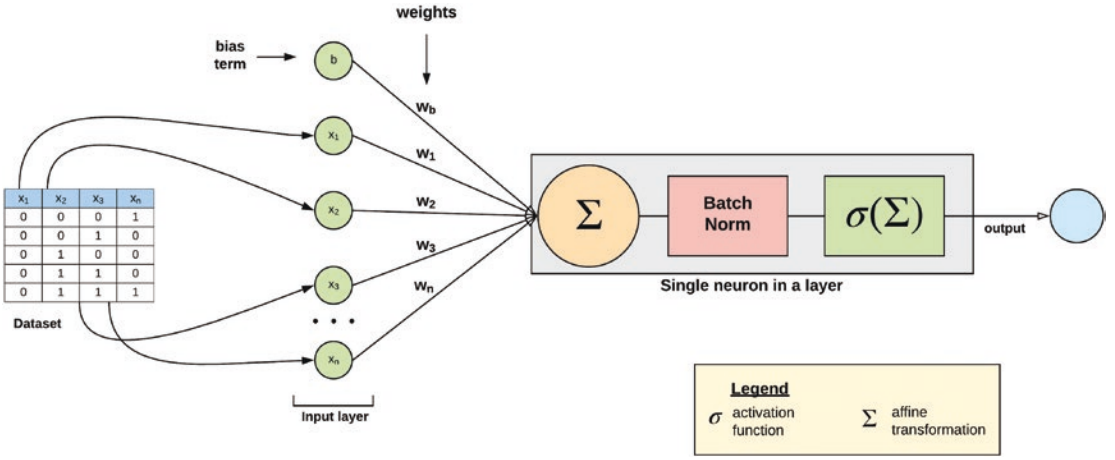

Figure 32-2. *Batch normalization, also known as batch norm*

The neural network learns the parameters for scaling and shifting the data at every layer during training. Also at the training phase, the score of the running mean and standard deviation of the data is maintained so that it can be used to normalize the test data before evaluation.

Batch normalization also mitigates the exploding and vanishing gradient problem irrespective of weight initialization. However, due to the added computational step at each layer, the network may be a bit slower. A batch normalization layer is added to a TensorFlow 2.0 network model by calling the method **'tf.keras.layers.BatchNormalization()'** as shown in the following code listing.

```
# create the model
def model_fn():
    model = tf.keras.Sequential()
    # Adds a densely-connected layer with 256 units to the model:
    model.add(tf.keras.layers.Dense(256, activation='relu', input_dim=784))
    # Add Dense layer with 64 units
    model.add(tf.keras.layers.Dense(64, activation='relu'))
    # Add a Batch Normalization layer
    model.add(tf.keras.layers.BatchNormalization())
    # Add another densely-connected layer with 32 units:
    model.add(tf.keras.layers.Dense(32, activation='relu'))
    # Add a softmax layer with 10 output units:
    model.add(tf.keras.layers.Dense(10, activation='softmax'))
```

```
    # compile the model
    model.compile(optimizer=tf.keras.optimizers.SGD(0.01),
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])
    return model
```

Gradient Clipping

Gradient clipping is another technique for hemming the problem of vanishing and exploding gradients mostly seen in recurrent networks due to training via backpropagation across a large number of deep recurrent layers. Gradient clipping involves trimming the computed gradients so that they remain within a specific range; in doing so, the gradients are prevented from saturating as the network trains across multiple deep layers.

Gradient clipping is implemented in TensorFlow 2.0 by adjusting the **'clipnorm'** or **'clipvalue'** parameters of the selected optimizer from the **'tf.keras.optimizers'** package. **'clipnorm'** clips the gradients by norm, while **'clipvalue'** clips the gradients by value.

This chapter introduces some important techniques that are employed to improve the performance of a neural network by further mitigating the issue of vanishing and exploding gradients. In the next chapter, we will see more optimization techniques for training deep neural network model.