Too little noise has no effect, whereas too much noise makes the mapping function too challenging to learn.

In TensorFlow 2.0, noise injection can be added to the model as a form of data augmentation using the method **'tf.keras.layers.GaussianNoise()'**. The **'stddev'** parameter of the method controls the standard deviation of the noise distribution. The following code listing shows an MLP Keras model with Gaussian noise applied to the model.

```
# 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 Gaussian Noise
    model.add(tf.keras.layers.GaussianNoise(stddev=1.0))
    # Add another densely-connected layer with 64 units:
    model.add(tf.keras.layers.Dense(64, 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.RMSprop(),
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])
    return model
```

Early Stopping

Early stopping involves storing the model parameters each time there is an improvement in the loss (or error) estimate on the validation dataset. At the end of the training phase, the stored model parameters are used rather than the last known parameter before termination.

The technique of early stopping is based on the observation that for a sufficiently complex classifier, as the training phase progresses, the error estimate on the training data continues to decrease, whereas the validation data will see an increase in the model error measure. This is illustrated in Figure 34-2.

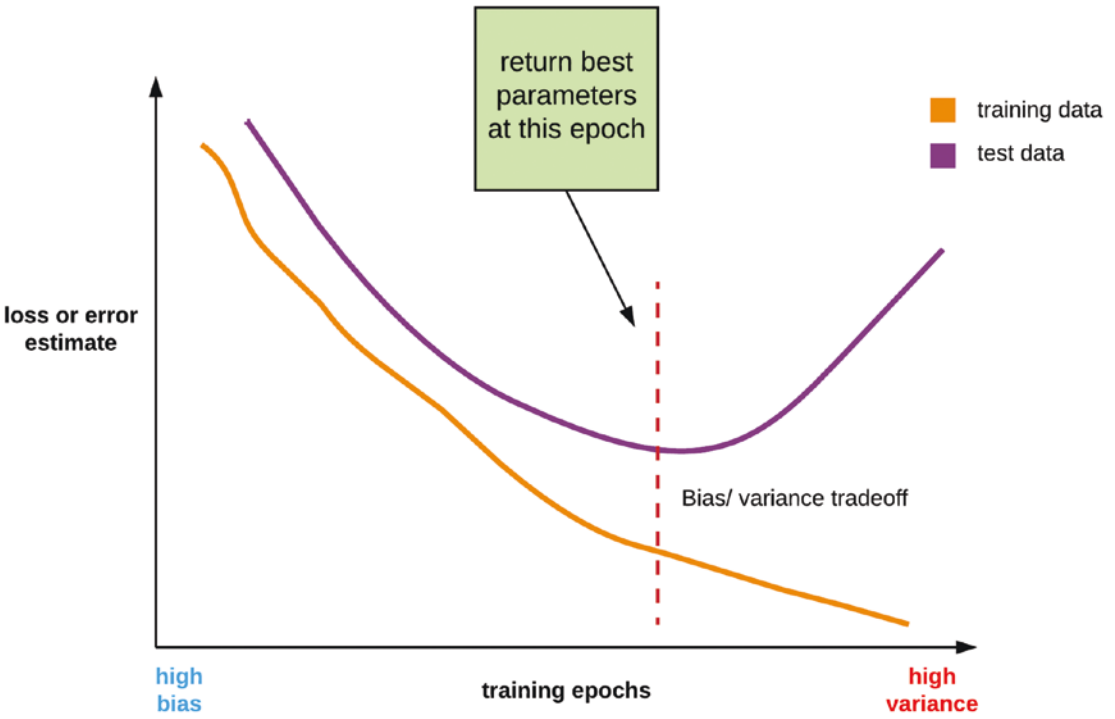

Figure 34-2. *Early stopping*

In TensorFlow 2.0, early stopping can be applied to stop training when there is no improvement in the validation accuracy or loss by applying the **'tf. keras.callbacks. EarlyStopping()'** method as a callback when training the model. For completeness sake, we will produce a complete code listing with early stopping applied to the MLP Fashion-MNIST model.

```
# install tensorflow 2.0
!pip install -q tensorflow==2.0.0-beta0

# import packages
import tensorflow as tf
import numpy as np

# import dataset
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.fashion_mnist.
load_data()

# flatten the 28*28 pixel images into one long 784 pixel vector
```

```
x_train = np.reshape(x_train, (-1, 784)).astype('float32')
x_test = np.reshape(x_test, (-1, 784)).astype('float32')

# scale dataset from 0 -> 255 to 0 -> 1
x_train /= 255
x_test /= 255

# one-hot encode targets
y_train = tf.keras.utils.to_categorical(y_train)
y_test = tf.keras.utils.to_categorical(y_test)

# 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 another densely-connected layer with 128 units:
    model.add(tf.keras.layers.Dense(128, activation='relu'))
    # Add another densely-connected layer with 64 units:
    model.add(tf.keras.layers.Dense(64, activation='relu'))
    # 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.RMSprop(),
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])
    return model

# use tf.data to batch and shuffle the dataset
train_ds = tf.data.Dataset.from_tensor_slices(
    (x_train, y_train)).shuffle(len(x_train)).repeat().batch(32)
test_ds = tf.data.Dataset.from_tensor_slices((x_test, y_test)).batch(32)

# build model
model = model_fn()
```

```
# early stopping
checkpoint = tf.keras.callbacks.EarlyStopping(
    monitor='val_loss',
    mode='auto',
    patience=5)

# assign callback
callbacks = [checkpoint]

# train the model
history = model.fit(train_ds, epochs=10,
                    steps_per_epoch=100,
                    validation_data=test_ds,
                    callbacks=callbacks)

# evaluate the model
score = model.evaluate(test_ds)
print('Test loss: {:.2f} \nTest accuracy: {:.2f}%'.format(score[0],
score[1]*100))
```

With early stopping applied to the preceding code, the training will stop once there is no improvement to the loss on the validation dataset. The **'patience'** parameter in the EarlyStopping method represents the number of epochs with no improvement, after which training will be stopped.

This chapter surveys some techniques to tackle the problem of overfitting when training with a deep neural network. In the next chapter, we will discuss on convolutional neural networks for building predictive models for computer vision use cases such as image recognition with TensorFlow 2.0.

CHAPTER 35

Convolutional Neural Networks (CNN)

Convolutional neural networks (CNN) are a specific type of neural network systems that are particularly suited for computer vision problems such as image recognition. In such tasks, the dataset is represented as a 2-D grid of pixels. See Figure 35-1.

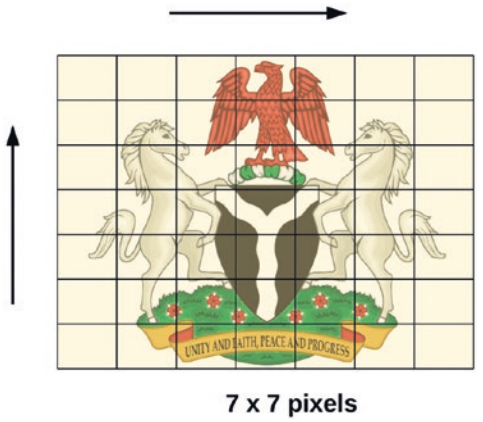

Figure 35-1. *2-D representation of an image*

An image is depicted in the computer as a matrix of pixel intensity values ranging from 0 to 255. A grayscale (or black and white) image consists of a single channel with 0 representing the black areas and 255 the white regions with the values in between for various shades of gray.

For example, the image in Figure 35-2 is a 10 x 10 grayscale image with its matrix representation.

E. Bisong, *Building Machine Learning and Deep Learning Models on Google Cloud Platform*,
https://doi.org/10.1007/978-1-4842-4470-8_35