of non-linearities among the data features, then it is recommended to add more layers while taking care to ensure that the model performs well on test data. Choosing the number of neurons in a hidden layer and the number of hidden layers is usually a case of a trial-and-error heuristics and presents the case of applying hyper-parameter tuning to improve the network performance. Using a grid search for hyper-parameter tuning is a good way to approximate an optimal neural network architecture that performs well on test data.

Multilayer Perceptron (MLP) with Keras

In this section, we examine a motivating example by building an MLP model with Keras. In doing so, we'll go through the following steps:

- Import and transform the dataset.
- Build and compile the model.
- Train the data using **'Model.fit()'**.
- Evaluate the model using **'Model.evaluate()'**.
- Predict on unseen data using **'Model.predict()'**.

The dataset used for this example is the Fashion-MNIST database of fashion articles. This dataset contains 60,000 28 x 28 pixel grayscale images of ten clothing items (the target classes). This dataset is downloaded from the **'tf.keras.datasets'** package. The following code example will build a simple MLP neural network for the computer to classify an image of a clothing item into its appropriate class. The network architecture has the following layers:

- A dense hidden layer with 250 neurons
- A second hidden layer with 64 neurons
- A third hidden layer with 32 neurons
- An output layer with 10 output classes

```
# 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 Dense 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.SGD(0.01),
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])
    return model

# build model
model = model_fn()
```

```
# 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)

# train the model
model.fit(train_ds, epochs=10,
          steps_per_epoch=2000)

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

'Ouput:'
Test loss: 0.35
Test accuracy: 87.36%
```

Observe the following from the preceding code:

- A Keras Sequential Model is built by calling the **'tf.keras.Sequential()'** method from which layers are then added to the model.
- After constructing the model layers, the model is compiled by calling the method '.compile()'.
- The model is trained by calling the **'.fit()'** method which receives the training features and targets from the **'tf.data.Dataset'** pipeline.
- The method **'.evaluate()'** is used to get the final metric estimate and the loss score of the model after training.

In this chapter, we introduced the multilayer perceptron network and how it achieves good performance on complex learning problems by stacking layers of neurons together to form a deep representational hierarchy. By doing this, the network learns what features are relevant and also learns what weights of the network will best approximate the target function.

In the next chapter, we will discuss on other considerations for training deep neural network.

CHAPTER 32

Other Considerations for Training the Network

In this chapter, we will cover some other important techniques to consider when training a deep neural network.

Weight Initialization

Weight initialization is a technique for assigning initial values to the weights (parameters) of the neural network before training (see Figure 32-1). Proper weight initialization may mitigate the effects of vanishing and exploding gradients when training the network. It may also speed up the training process. Two commonly used methods for weight initializations are the Xavier and the He techniques. We will not go into the technical explanation of these initialization strategies. However, they are implemented in the standard deep learning framework libraries such as TensorFlow and Keras. In TensorFlow 2.0, the dense layer in **'tf.keras.layers.Dense()'** has the Glorot uniform initializer, also called Xavier uniform initializer as its default kernel initializer.

E. Bisong, *Building Machine Learning and Deep Learning Models on Google Cloud Platform*,
https://doi.org/10.1007/978-1-4842-4470-8_32