Chapter 19

How to Classify Black and White Photos of Clothing

The Fashion-MNIST clothing classification problem is a new standard dataset used in computer vision and deep learning. Although the dataset is relatively simple, it can be used as the basis for learning and practicing how to develop, evaluate, and use deep convolutional neural networks for image classification from scratch. This includes how to develop a robust test harness for estimating the performance of the model, how to explore improvements to the model, and how to save the model and later load it to make predictions on new data. In this tutorial, you will discover how to develop a convolutional neural network for clothing classification from scratch. After completing this tutorial, you will know:

- How to develop a test harness to develop a robust evaluation of a model and establish a baseline of performance for a classification task.
- How to explore extensions to a baseline model to improve learning and model capacity.
- How to develop a finalized model, evaluate the performance of the final model, and use it to make predictions on new images.

Let's get started.

19.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. Fashion-MNIST Clothing Classification
2. Model Evaluation Methodology
3. How to Develop a Baseline Model
4. How to Develop an Improved Model
5. How to Finalize the Model and Make Predictions

19.2 Fashion-MNIST Clothing Classification

The Fashion-MNIST dataset is proposed as a more challenging replacement dataset for the MNIST dataset. It is a dataset comprised of 60,000 small square 28×28 pixel grayscale images of items of 10 types of clothing, such as shoes, t-shirts, dresses, and more. The mapping of all 0-9 integers to class labels is listed below.

- 0: T-shirt/top
- 1: Trouser
- 2: Pullover
- 3: Dress
- 4: Coat
- 5: Sandal
- 6: Shirt
- 7: Sneaker
- 8: Bag
- 9: Ankle boot

It is a more challenging classification problem than MNIST and top results are achieved by deep learning convolutional neural networks with a classification accuracy of about 90% to 95% on the hold out test dataset. The example below loads the Fashion-MNIST dataset using the Keras API and creates a plot of the first nine images in the training dataset.

```
# example of loading the fashion mnist dataset
from matplotlib import pyplot
from keras.datasets import fashion_mnist
# load dataset
(trainX, trainy), (testX, testy) = fashion_mnist.load_data()
# summarize loaded dataset
print('Train: X=%s, y=%s' % (trainX.shape, trainy.shape))
print('Test: X=%s, y=%s' % (testX.shape, testy.shape))
# plot first few images
for i in range(9):
  # define subplot
  pyplot.subplot(330 + 1 + i)
  # plot raw pixel data
  pyplot.imshow(trainX[i], cmap=pyplot.get_cmap('gray'))
# show the figure
pyplot.show()
```

Listing 19.1: Example of loading and summarizing the Fashion-MNIST dataset.

Running the example loads the Fashion-MNIST train and test dataset and prints their shape. We can see that there are 60,000 examples in the training dataset and 10,000 in the test dataset and that images are indeed square with 28×28 pixels.

```
Train: X=(60000, 28, 28), y=(60000,)
Test: X=(10000, 28, 28), y=(10000,)
```

Listing 19.2: Example output from loading and summarizing the Fashion-MNIST dataset.

A plot of the first nine images in the dataset is also created showing that indeed the images are grayscale photographs of items of clothing.

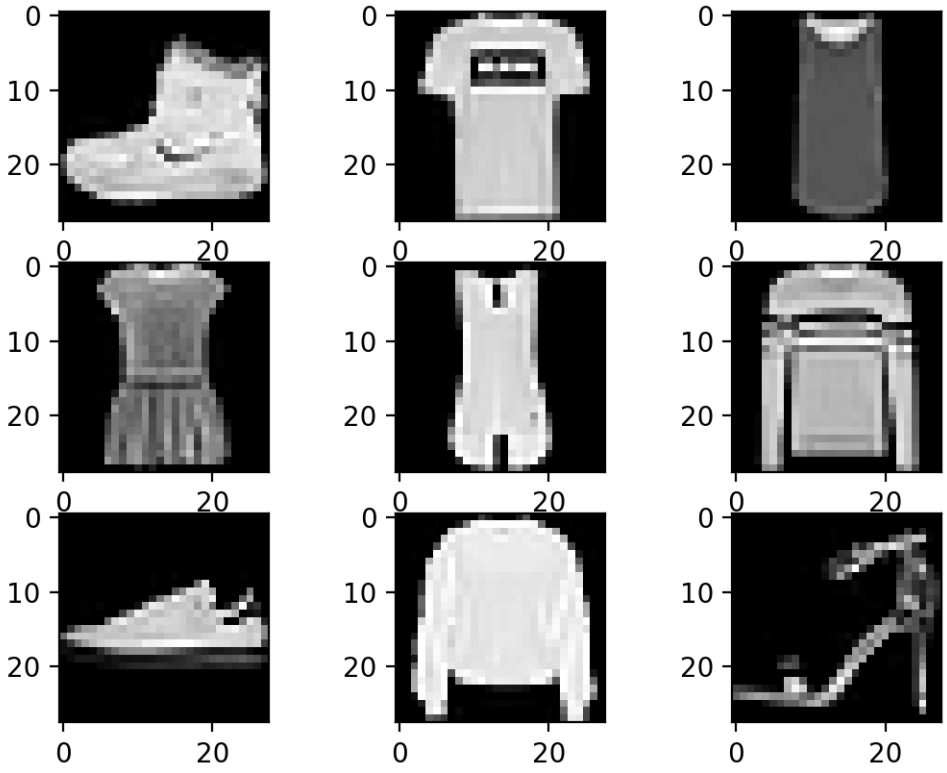

Figure 19.1: Plot of a Subset of Images From the Fashion-MNIST Dataset.

19.3 Model Evaluation Methodology

The Fashion-MNIST dataset was developed as a response to the wide use of the MNIST dataset, that has been effectively *solved* given the use of modern convolutional neural networks. Fashion-MNIST was proposed to be a replacement for MNIST, and although it has not been solved, it is possible to routinely achieve error rates of 10% or less. Like MNIST, it can be a useful starting point for developing and practicing a methodology for solving image classification using convolutional neural networks. Instead of reviewing the literature on well-performing models on the dataset, we can develop a new model from scratch.

The dataset already has a well-defined train and test dataset that we can use. In order to estimate the performance of a model for a given training run, we can further split the training set

into a train and validation dataset. Performance on the train and validation dataset over each run can then be plotted to provide learning curves and insight into how well a model is learning the problem. The Keras API supports this by specifying the `validation_data` argument to the `model.fit()` function when training the model, that will, in turn, return an object that describes model performance for the chosen loss and metrics on each training epoch.

```
...
# record model performance on a validation dataset during training
history = model.fit(..., validation_data=(valX, valY))
```

Listing 19.3: Example of fitting a model with a validation dataset.

In order to estimate the performance of a model on the problem in general, we can use k-fold cross-validation, perhaps 5-fold cross-validation. This will give some account of the model's variance with both respect to differences in the training and test datasets and the stochastic nature of the learning algorithm. The performance of a model can be taken as the mean performance across k-folds, given with the standard deviation, that could be used to estimate a confidence interval if desired. We can use the `KFold` class from the scikit-learn API to implement the k-fold cross-validation evaluation of a given neural network model. There are many ways to achieve this, although we can choose a flexible approach where the `KFold` is only used to specify the row indexes used for each split.

```
...
# example of k-fold cv for a neural net
data = ...
# prepare cross validation
kfold = KFold(5, shuffle=True, random_state=1)
# enumerate splits
for train_ix, test_ix in kfold.split(data):
  model = ...
  ...
```

Listing 19.4: Example of evaluating a model with k-fold cross-validation.

We will hold back the actual test dataset and use it as an evaluation of our final model.

19.4 How to Develop a Baseline Model

The first step is to develop a baseline model. This is critical as it both involves developing the infrastructure for the test harness so that any model we design can be evaluated on the dataset, and it establishes a baseline in model performance on the problem, by which all improvements can be compared. The design of the test harness is modular, and we can develop a separate function for each piece. This allows a given aspect of the test harness to be modified or inter-changed, if we desire, separately from the rest. We can develop this test harness with five key elements. They are the loading of the dataset, the preparation of the dataset, the definition of the model, the evaluation of the model, and the presentation of results.

19.4.1 Load Dataset

We know some things about the dataset. For example, we know that the images are all pre-segmented (e.g. each image contains a single item of clothing), that the images all have the

same square size of 28 × 28 pixels, and that the images are grayscale. Therefore, we can load the images and reshape the data arrays to have a single color channel.

```
...
# load dataset
(trainX, trainY), (testX, testY) = fashion_mnist.load_data()
# reshape dataset to have a single channel
trainX = trainX.reshape((trainX.shape[0], 28, 28, 1))
testX = testX.reshape((testX.shape[0], 28, 28, 1))
```

Listing 19.5: Example of adding a channels dimension to the loaded dataset.

We also know that there are 10 classes and that classes are represented as unique integers. We can, therefore, use a one hot encoding for the class element of each sample, transforming the integer into a 10 element binary vector with a 1 for the index of the class value. We can achieve this with the `to_categorical()` utility function.

```
...
# one hot encode target values
trainY = to_categorical(trainY)
testY = to_categorical(testY)
```

Listing 19.6: Example of one hot encoding the target variable.

The `load_dataset()` function implements these behaviors and can be used to load the dataset.

```
# load train and test dataset
def load_dataset():
  # load dataset
  (trainX, trainY), (testX, testY) = fashion_mnist.load_data()
  # reshape dataset to have a single channel
  trainX = trainX.reshape((trainX.shape[0], 28, 28, 1))
  testX = testX.reshape((testX.shape[0], 28, 28, 1))
  # one hot encode target values
  trainY = to_categorical(trainY)
  testY = to_categorical(testY)
  return trainX, trainY, testX, testY
```

Listing 19.7: Example of a function for loading the dataset.

19.4.2 Prepare Pixel Data

We know that the pixel values for each image in the dataset are unsigned integers in the range between black and white, or 0 and 255. We do not know the best way to scale the pixel values for modeling, but we know that some scaling will be required. A good starting point is to normalize the pixel values of grayscale images, e.g. rescale them to the range [0,1]. This involves first converting the data type from unsigned integers to floats, then dividing the pixel values by the maximum value.

```
...
# convert from integers to floats
train_norm = train.astype('float32')
test_norm = test.astype('float32')
# normalize to range 0-1
```

```
train_norm = train_norm / 255.0
test_norm = test_norm / 255.0
```

Listing 19.8: Example of normalizing the pixel values.

The `prep_pixels()` function below implements these behaviors and is provided with the pixel values for both the train and test datasets that will need to be scaled.

```
# scale pixels
def prep_pixels(train, test):
  # convert from integers to floats
  train_norm = train.astype('float32')
  test_norm = test.astype('float32')
  # normalize to range 0-1
  train_norm = train_norm / 255.0
  test_norm = test_norm / 255.0
  # return normalized images
  return train_norm, test_norm
```

Listing 19.9: Example of a function for scaling the pixel values.

This function must be called to prepare the pixel values prior to any modeling.

19.4.3 Define Model

Next, we need to define a baseline convolutional neural network model for the problem. The model has two main aspects: the feature extraction front end comprised of convolutional and pooling layers, and the classifier backend that will make a prediction. For the convolutional front end, we can start with a single convolutional layer with a small filter size (3,3) and a modest number of filters (32) followed by a max pooling layer. The filter maps can then be flattened to provide features to the classifier.

Given that the problem is a multiclass classification, we know that we will require an output layer with 10 nodes in order to predict the probability distribution of an image belonging to each of the 10 classes. This will also require the use of a softmax activation function. Between the feature extractor and the output layer, we can add a dense layer to interpret the features, in this case with 100 nodes. All layers will use the ReLU activation function and the He weight initialization scheme, both best practices.

We will use a conservative configuration for the stochastic gradient descent optimizer with a learning rate of 0.01 and a momentum of 0.9. The categorical cross-entropy loss function will be optimized, suitable for multiclass classification, and we will monitor the classification accuracy metric, which is appropriate given we have the same number of examples in each of the 10 classes. The `define_model()` function below will define and return this model.

```
# define cnn model
def define_model():
  model = Sequential()
  model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',
      input_shape=(28, 28, 1)))
  model.add(MaxPooling2D((2, 2)))
  model.add(Flatten())
  model.add(Dense(100, activation='relu', kernel_initializer='he_uniform'))
  model.add(Dense(10, activation='softmax'))
  # compile model
```

```
  opt = SGD(lr=0.01, momentum=0.9)
  model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
  return model
```

Listing 19.10: Example of a function for defining the model.

19.4.4 Evaluate Model

After the model is defined, we need to evaluate it. The model will be evaluated using 5-fold cross-validation. The value of `k=5` was chosen to provide a baseline for both repeated evaluation and to not be too large as to require a long running time. Each test set will be 20% of the training dataset, or about 12,000 examples, close to the size of the actual test set for this problem. The training dataset is shuffled prior to being split and the sample shuffling is performed each time so that any model we evaluate will have the same train and test datasets in each fold, providing an apples-to-apples comparison.

We will train the baseline model for a modest 10 training epochs with a default batch size of 32 examples. The test set for each fold will be used to evaluate the model both during each epoch of the training run, so we can later create learning curves, and at the end of the run, so we can estimate the performance of the model. As such, we will keep track of the resulting history from each run, as well as the classification accuracy of the fold. The `evaluate_model()` function below implements these behaviors, taking the training dataset as arguments and returning a list of accuracy scores and training histories that can be later summarized.

```
# evaluate a model using k-fold cross-validation
def evaluate_model(dataX, dataY, n_folds=5):
  scores, histories = list(), list()
  # prepare cross validation
  kfold = KFold(n_folds, shuffle=True, random_state=1)
  # enumerate splits
  for train_ix, test_ix in kfold.split(dataX):
    # define model
    model = define_model()
    # select rows for train and test
    trainX, trainY, testX, testY = dataX[train_ix], dataY[train_ix], dataX[test_ix],
        dataY[test_ix]
    # fit model
    history = model.fit(trainX, trainY, epochs=10, batch_size=32, validation_data=(testX,
        testY), verbose=0)
    # evaluate model
    _, acc = model.evaluate(testX, testY, verbose=0)
    print('> %.3f' % (acc * 100.0))
    # append scores
    scores.append(acc)
    histories.append(history)
  return scores, histories
```

Listing 19.11: Example of a function for evaluating the performance of a model.

19.4.5 Present Results

Once the model has been evaluated, we can present the results. There are two key aspects to present: the diagnostics of the learning behavior of the model during training and the estimation

of the model performance. These can be implemented using separate functions. First, the diagnostics involve creating a line plot showing model performance on the train and test set during each fold of the k-fold cross-validation. These plots are valuable for getting an idea of whether a model is overfitting, underfitting, or has a good fit for the dataset. We will create a single figure with two subplots, one for loss and one for accuracy. Blue lines will indicate model performance on the training dataset and orange lines will indicate performance on the hold out test dataset. The `summarize_diagnostics()` function below creates and shows this plot given the collected training histories.

```
# plot diagnostic learning curves
def summarize_diagnostics(histories):
  for i in range(len(histories)):
    # plot loss
    pyplot.subplot(211)
    pyplot.title('Cross Entropy Loss')
    pyplot.plot(histories[i].history['loss'], color='blue', label='train')
    pyplot.plot(histories[i].history['val_loss'], color='orange', label='test')
    # plot accuracy
    pyplot.subplot(212)
    pyplot.title('Classification Accuracy')
    pyplot.plot(histories[i].history['acc'], color='blue', label='train')
    pyplot.plot(histories[i].history['val_acc'], color='orange', label='test')
  pyplot.show()
```

Listing 19.12: Example of a function for plotting learning curves.

Next, the classification accuracy scores collected during each fold can be summarized by calculating the mean and standard deviation. This provides an estimate of the average expected performance of the model trained on this dataset, with an estimate of the average variance in the mean. We will also summarize the distribution of scores by creating and showing a box and whisker plot. The `summarize_performance()` function below implements this for a given list of scores collected during model evaluation.

```
# summarize model performance
def summarize_performance(scores):
  # print summary
  print('Accuracy: mean=%.3f std=%.3f, n=%d' % (mean(scores)*100, std(scores)*100,
      len(scores)))
  # box and whisker plots of results
  pyplot.boxplot(scores)
  pyplot.show()
```

Listing 19.13: Example of a function for summarizing model performance.

19.4.6 Complete Example

We need a function that will drive the test harness. This involves calling all of the defined functions.

```
# run the test harness for evaluating a model
def run_test_harness():
  # load dataset
  trainX, trainY, testX, testY = load_dataset()
  # prepare pixel data
```

```
  trainX, testX = prep_pixels(trainX, testX)
  # evaluate model
  scores, histories = evaluate_model(model, trainX, trainY)
  # learning curves
  summarize_diagnostics(histories)
  # summarize estimated performance
  summarize_performance(scores)
```

Listing 19.14: Example of a function for driving the test harness.

We now have everything we need; the complete code example for a baseline convolutional neural network model on the MNIST dataset is listed below.

```
# baseline cnn model for fashion mnist
from numpy import mean
from numpy import std
from matplotlib import pyplot
from sklearn.model_selection import KFold
from keras.datasets import fashion_mnist
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import Dense
from keras.layers import Flatten
from keras.optimizers import SGD

# load train and test dataset
def load_dataset():
  # load dataset
  (trainX, trainY), (testX, testY) = fashion_mnist.load_data()
  # reshape dataset to have a single channel
  trainX = trainX.reshape((trainX.shape[0], 28, 28, 1))
  testX = testX.reshape((testX.shape[0], 28, 28, 1))
  # one hot encode target values
  trainY = to_categorical(trainY)
  testY = to_categorical(testY)
  return trainX, trainY, testX, testY

# scale pixels
def prep_pixels(train, test):
  # convert from integers to floats
  train_norm = train.astype('float32')
  test_norm = test.astype('float32')
  # normalize to range 0-1
  train_norm = train_norm / 255.0
  test_norm = test_norm / 255.0
  # return normalized images
  return train_norm, test_norm

# define cnn model
def define_model():
  model = Sequential()
  model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',
      input_shape=(28, 28, 1)))
  model.add(MaxPooling2D((2, 2)))
  model.add(Flatten())
```

```
  model.add(Dense(100, activation='relu', kernel_initializer='he_uniform'))
  model.add(Dense(10, activation='softmax'))
  # compile model
  opt = SGD(lr=0.01, momentum=0.9)
  model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
  return model

# evaluate a model using k-fold cross-validation
def evaluate_model(dataX, dataY, n_folds=5):
  scores, histories = list(), list()
  # prepare cross validation
  kfold = KFold(n_folds, shuffle=True, random_state=1)
  # enumerate splits
  for train_ix, test_ix in kfold.split(dataX):
    # define model
    model = define_model()
    # select rows for train and test
    trainX, trainY, testX, testY = dataX[train_ix], dataY[train_ix], dataX[test_ix],
        dataY[test_ix]
    # fit model
    history = model.fit(trainX, trainY, epochs=10, batch_size=32, validation_data=(testX,
        testY), verbose=0)
    # evaluate model
    _, acc = model.evaluate(testX, testY, verbose=0)
    print('> %.3f' % (acc * 100.0))
    # append scores
    scores.append(acc)
    histories.append(history)
  return scores, histories

# plot diagnostic learning curves
def summarize_diagnostics(histories):
  for i in range(len(histories)):
    # plot loss
    pyplot.subplot(211)
    pyplot.title('Cross Entropy Loss')
    pyplot.plot(histories[i].history['loss'], color='blue', label='train')
    pyplot.plot(histories[i].history['val_loss'], color='orange', label='test')
    # plot accuracy
    pyplot.subplot(212)
    pyplot.title('Classification Accuracy')
    pyplot.plot(histories[i].history['acc'], color='blue', label='train')
    pyplot.plot(histories[i].history['val_acc'], color='orange', label='test')
  pyplot.show()

# summarize model performance
def summarize_performance(scores):
  # print summary
  print('Accuracy: mean=%.3f std=%.3f, n=%d' % (mean(scores)*100, std(scores)*100,
      len(scores)))
  # box and whisker plots of results
  pyplot.boxplot(scores)
  pyplot.show()

# run the test harness for evaluating a model
def run_test_harness():
```

```
  # load dataset
  trainX, trainY, testX, testY = load_dataset()
  # prepare pixel data
  trainX, testX = prep_pixels(trainX, testX)
  # evaluate model
  scores, histories = evaluate_model(trainX, trainY)
  # learning curves
  summarize_diagnostics(histories)
  # summarize estimated performance
  summarize_performance(scores)

# entry point, run the test harness
run_test_harness()
```

Listing 19.15: Example of defining and evaluating a baseline model on the dataset.

Running the example prints the classification accuracy for each fold of the cross-validation process. This is helpful to get an idea that the model evaluation is progressing. We can see that for each fold, the baseline model achieved an error rate below 10%, and in two cases 98% and 99% accuracy. These are good results.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

```
> 91.200
> 91.217
> 90.958
> 91.242
> 91.317
```

Listing 19.16: Example output from during the evaluation of each baseline model.

Next, a diagnostic plot is shown, giving insight into the learning behavior of the model across each fold. In this case, we can see that the model generally achieves a good fit, with train and test learning curves converging. There may be some signs of slight overfitting.

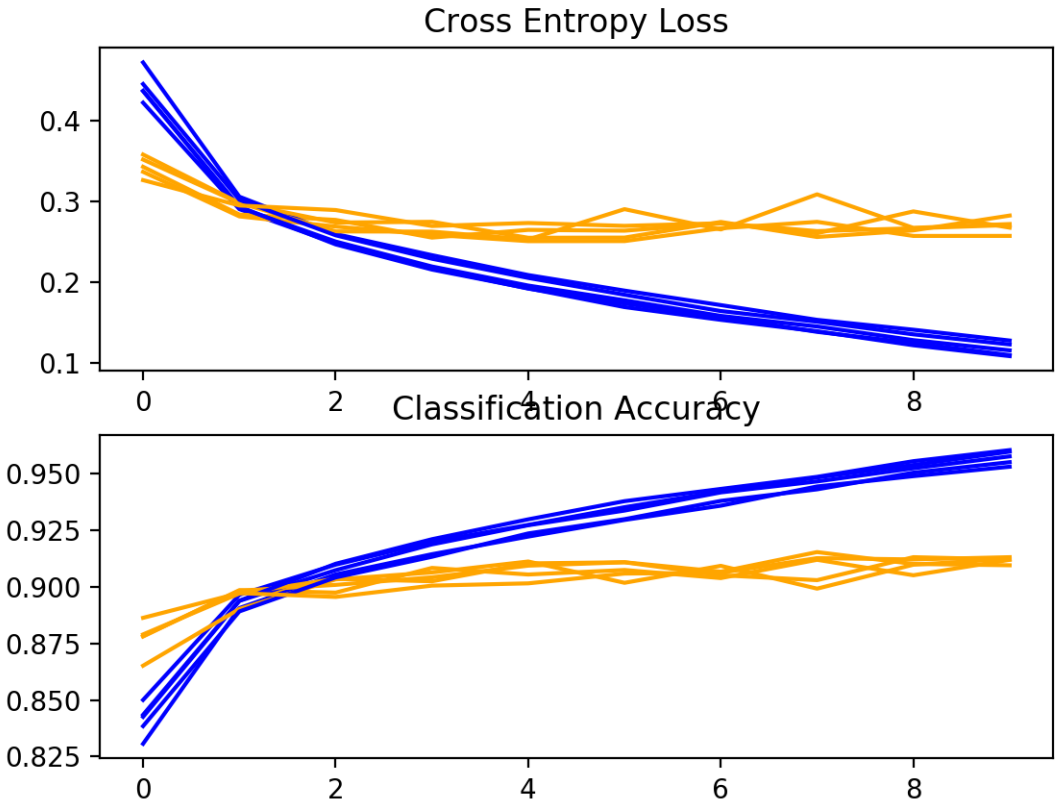

Figure 19.2: Loss and Accuracy Learning Curves for the Baseline Model on the Fashion-MNIST Dataset During k-Fold Cross-Validation.

Next, the summary of the model performance is calculated. We can see in this case, the model has an estimated skill of about 91%, which is impressive.

```
Accuracy: mean=91.187 std=0.121, n=5
```

Listing 19.17: Example output from the final evaluation of the baseline model.

Finally, a box and whisker plot is created to summarize the distribution of accuracy scores.

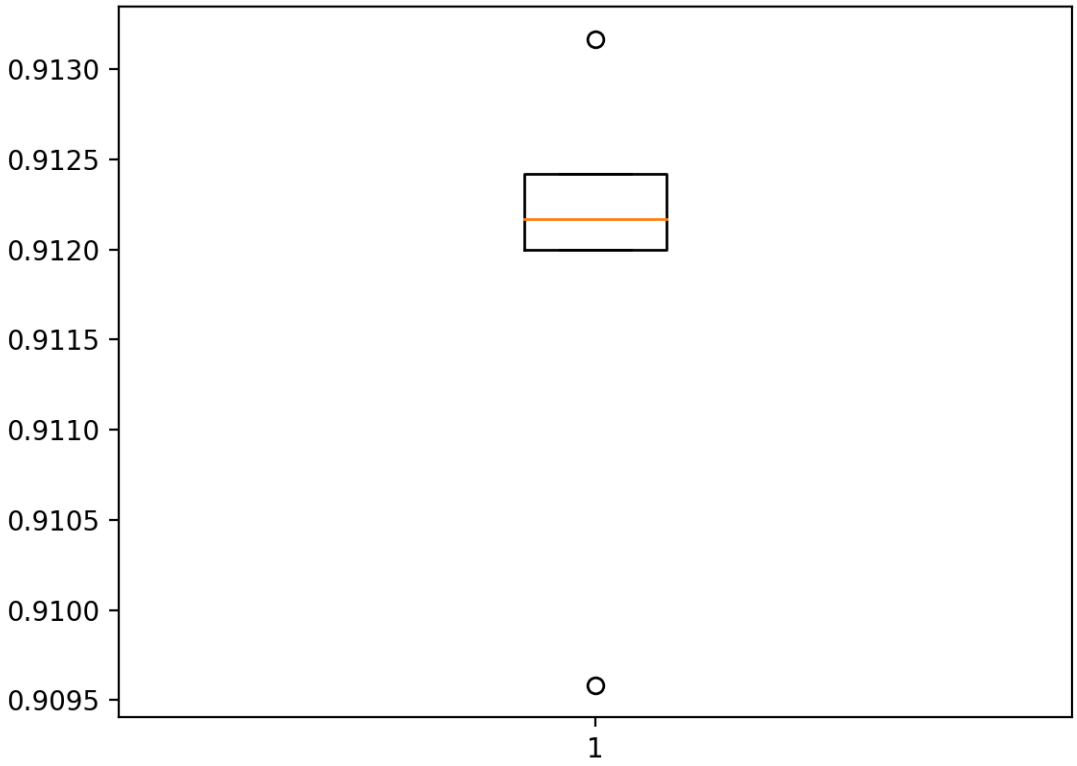

Figure 19.3: Box and Whisker Plot of Accuracy Scores for the Baseline Model on the Fashion-MNIST Dataset Evaluated Using k-Fold Cross-Validation.

As we would expect, the distribution spread across the low-nineties. We now have a robust test harness and a well-performing baseline model.

19.5 How to Develop an Improved Model

There are many ways that we might explore improvements to the baseline model. We will look at areas that often result in an improvement, so-called low-hanging fruit. The first will be a change to the convolutional operation to add padding and the second will build on this to increase the number of filters.

19.5.1 Padding Convolutions

Adding padding to the convolutional operation can often result in better model performance, as more of the input image of feature maps are given an opportunity to participate or contribute to the output. By default, the convolutional operation uses `'valid'` padding, which means that convolutions are only applied where possible. This can be changed to `'same'` padding so that zero values are added around the input such that the output has the same size as the input.

```
...
```

```
model.add(Conv2D(32, (3, 3), padding='same', activation='relu',
    kernel_initializer='he_uniform', input_shape=(28, 28, 1)))
```

Listing 19.18: Example of padding convolutional layers.

The full code listing including the change to padding is provided below for completeness.

```
# model with padded convolutions for the fashion mnist dataset
from numpy import mean
from numpy import std
from matplotlib import pyplot
from sklearn.model_selection import KFold
from keras.datasets import fashion_mnist
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import Dense
from keras.layers import Flatten
from keras.optimizers import SGD

# load train and test dataset
def load_dataset():
  # load dataset
  (trainX, trainY), (testX, testY) = fashion_mnist.load_data()
  # reshape dataset to have a single channel
  trainX = trainX.reshape((trainX.shape[0], 28, 28, 1))
  testX = testX.reshape((testX.shape[0], 28, 28, 1))
  # one hot encode target values
  trainY = to_categorical(trainY)
  testY = to_categorical(testY)
  return trainX, trainY, testX, testY

# scale pixels
def prep_pixels(train, test):
  # convert from integers to floats
  train_norm = train.astype('float32')
  test_norm = test.astype('float32')
  # normalize to range 0-1
  train_norm = train_norm / 255.0
  test_norm = test_norm / 255.0
  # return normalized images
  return train_norm, test_norm

# define cnn model
def define_model():
  model = Sequential()
  model.add(Conv2D(32, (3, 3), padding='same', activation='relu',
      kernel_initializer='he_uniform', input_shape=(28, 28, 1)))
  model.add(MaxPooling2D((2, 2)))
  model.add(Flatten())
  model.add(Dense(100, activation='relu', kernel_initializer='he_uniform'))
  model.add(Dense(10, activation='softmax'))
  # compile model
  opt = SGD(lr=0.01, momentum=0.9)
  model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
  return model
```

```
# evaluate a model using k-fold cross-validation
def evaluate_model(dataX, dataY, n_folds=5):
  scores, histories = list(), list()
  # prepare cross validation
  kfold = KFold(n_folds, shuffle=True, random_state=1)
  # enumerate splits
  for train_ix, test_ix in kfold.split(dataX):
    # define model
    model = define_model()
    # select rows for train and test
    trainX, trainY, testX, testY = dataX[train_ix], dataY[train_ix], dataX[test_ix],
        dataY[test_ix]
    # fit model
    history = model.fit(trainX, trainY, epochs=10, batch_size=32, validation_data=(testX,
        testY), verbose=0)
    # evaluate model
    _, acc = model.evaluate(testX, testY, verbose=0)
    print('> %.3f' % (acc * 100.0))
    # append scores
    scores.append(acc)
    histories.append(history)
  return scores, histories

# plot diagnostic learning curves
def summarize_diagnostics(histories):
  for i in range(len(histories)):
    # plot loss
    pyplot.subplot(211)
    pyplot.title('Cross Entropy Loss')
    pyplot.plot(histories[i].history['loss'], color='blue', label='train')
    pyplot.plot(histories[i].history['val_loss'], color='orange', label='test')
    # plot accuracy
    pyplot.subplot(212)
    pyplot.title('Classification Accuracy')
    pyplot.plot(histories[i].history['acc'], color='blue', label='train')
    pyplot.plot(histories[i].history['val_acc'], color='orange', label='test')
  pyplot.show()

# summarize model performance
def summarize_performance(scores):
  # print summary
  print('Accuracy: mean=%.3f std=%.3f, n=%d' % (mean(scores)*100, std(scores)*100,
      len(scores)))
  # box and whisker plots of results
  pyplot.boxplot(scores)
  pyplot.show()

# run the test harness for evaluating a model
def run_test_harness():
  # load dataset
  trainX, trainY, testX, testY = load_dataset()
  # prepare pixel data
  trainX, testX = prep_pixels(trainX, testX)
  # evaluate model
  scores, histories = evaluate_model(trainX, trainY)
```

```
  # learning curves
  summarize_diagnostics(histories)
  # summarize estimated performance
  summarize_performance(scores)

# entry point, run the test harness
run_test_harness()
```

Listing 19.19: Example of evaluating the baseline model with padded convolutional layers.

Running the example again reports model performance for each fold of the cross-validation process.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see perhaps a small improvement in model performance as compared to the baseline across the cross-validation folds.

```
> 90.875
> 91.442
> 91.242
> 91.275
> 91.450
```

Listing 19.20: Example output from during the evaluation of each model.

A plot of the learning curves is created. As with the baseline model, we may see some slight overfitting. This could be addressed perhaps with use of regularization or the training for fewer epochs.

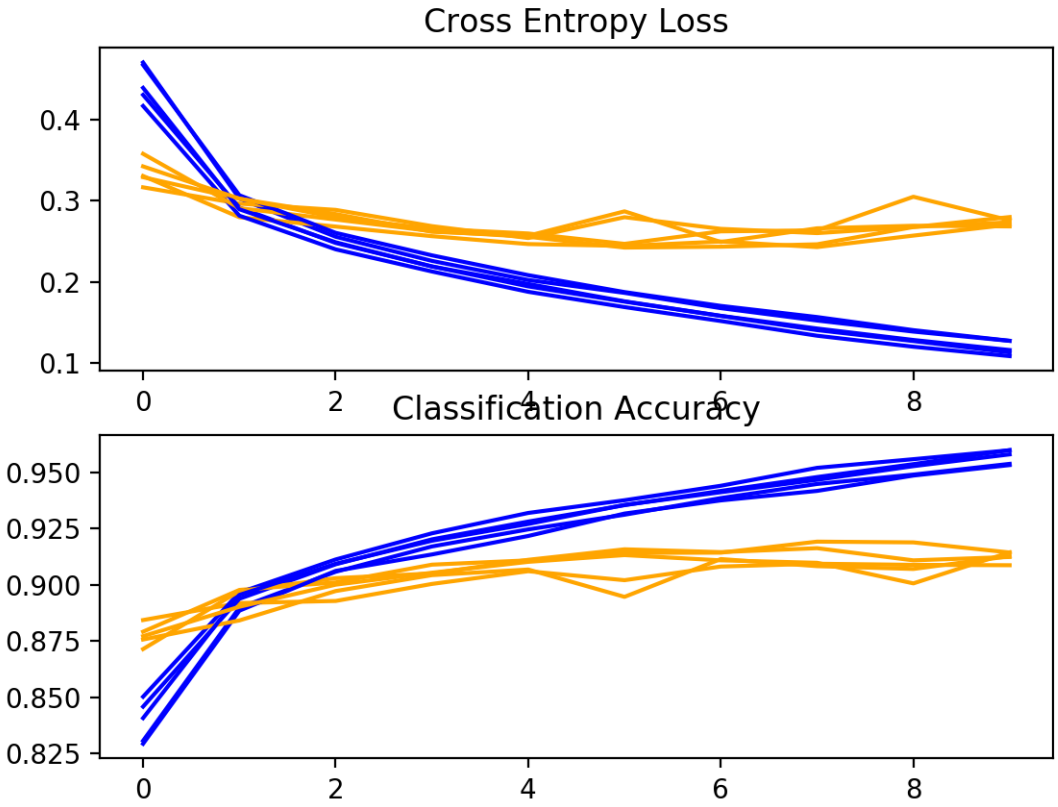

Figure 19.4: Loss and Accuracy Learning Curves for the Same Padding on the Fashion-MNIST Dataset During k-Fold Cross-Validation.

Next, the estimated performance of the model is presented, showing performance with a very slight increase in the mean accuracy of the model, 91.257% as compared to 91.187% with the baseline model. This may or may not be a real effect as it is within the bounds of the standard deviation. Perhaps more repeats of the experiment could tease out this fact.

```
Accuracy: mean=91.257 std=0.209, n=5
```

Listing 19.21: Example output from the final evaluation of the model.

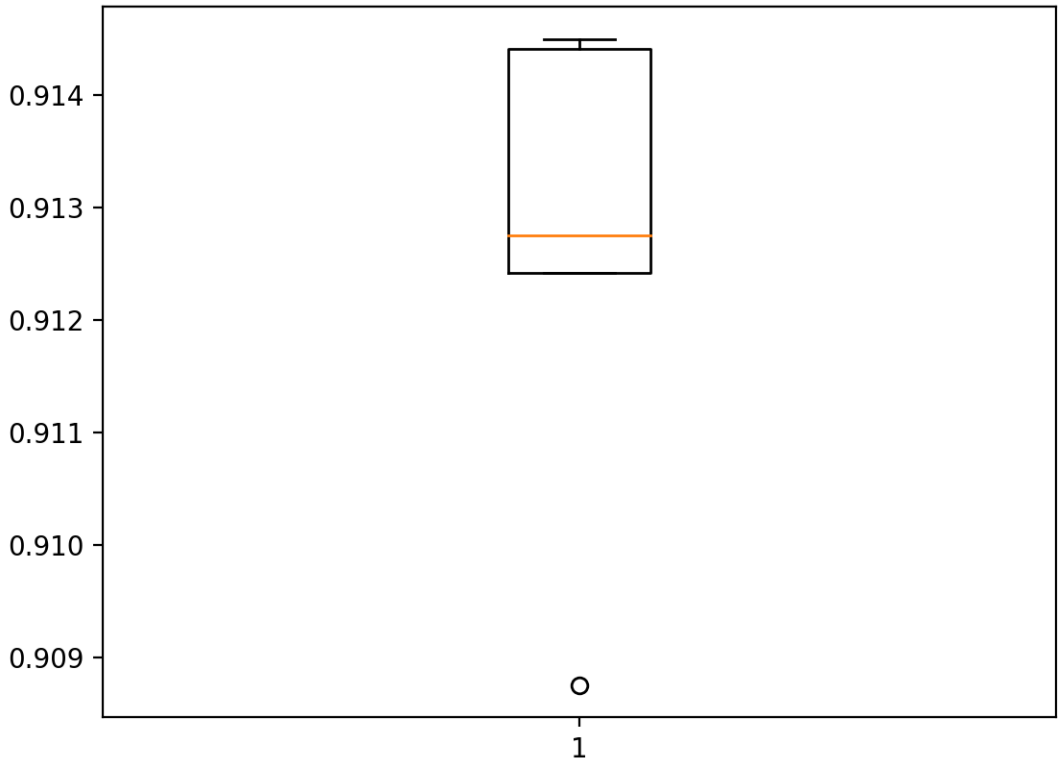

Figure 19.5: Box and Whisker Plot of Accuracy Scores for Same Padding on the Fashion-MNIST Dataset Evaluated Using k-Fold Cross-Validation.

19.5.2 Increasing Filters

An increase in the number of filters used in the convolutional layer can often improve performance, as it can provide more opportunity for extracting simple features from the input images. This is especially relevant when very small filters are used, such as 3×3 pixels. In this change, we can increase the number of filters in the convolutional layer from 32 to double that at 64. We will also build upon the possible improvement offered by using `'same'` padding.

```
...
model.add(Conv2D(64, (3, 3), padding='same', activation='relu',
    kernel_initializer='he_uniform', input_shape=(28, 28, 1)))
```

Listing 19.22: Example of increasing the number of filters.

The full code listing including the change to padding is provided below for completeness.

```
# model with double the filters for the fashion mnist dataset
from numpy import mean
from numpy import std
from matplotlib import pyplot
from sklearn.model_selection import KFold
from keras.datasets import fashion_mnist
```

```
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import Dense
from keras.layers import Flatten
from keras.optimizers import SGD

# load train and test dataset
def load_dataset():
  # load dataset
  (trainX, trainY), (testX, testY) = fashion_mnist.load_data()
  # reshape dataset to have a single channel
  trainX = trainX.reshape((trainX.shape[0], 28, 28, 1))
  testX = testX.reshape((testX.shape[0], 28, 28, 1))
  # one hot encode target values
  trainY = to_categorical(trainY)
  testY = to_categorical(testY)
  return trainX, trainY, testX, testY

# scale pixels
def prep_pixels(train, test):
  # convert from integers to floats
  train_norm = train.astype('float32')
  test_norm = test.astype('float32')
  # normalize to range 0-1
  train_norm = train_norm / 255.0
  test_norm = test_norm / 255.0
  # return normalized images
  return train_norm, test_norm

# define cnn model
def define_model():
  model = Sequential()
  model.add(Conv2D(64, (3, 3), padding='same', activation='relu',
      kernel_initializer='he_uniform', input_shape=(28, 28, 1)))
  model.add(MaxPooling2D((2, 2)))
  model.add(Flatten())
  model.add(Dense(100, activation='relu', kernel_initializer='he_uniform'))
  model.add(Dense(10, activation='softmax'))
  # compile model
  opt = SGD(lr=0.01, momentum=0.9)
  model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
  return model

# evaluate a model using k-fold cross-validation
def evaluate_model(dataX, dataY, n_folds=5):
  scores, histories = list(), list()
  # prepare cross validation
  kfold = KFold(n_folds, shuffle=True, random_state=1)
  # enumerate splits
  for train_ix, test_ix in kfold.split(dataX):
    # define model
    model = define_model()
    # select rows for train and test
    trainX, trainY, testX, testY = dataX[train_ix], dataY[train_ix], dataX[test_ix],
```

```
        dataY[test_ix]
    # fit model
    history = model.fit(trainX, trainY, epochs=10, batch_size=32, validation_data=(testX,
        testY), verbose=0)
    # evaluate model
    _, acc = model.evaluate(testX, testY, verbose=0)
    print('> %.3f' % (acc * 100.0))
    # append scores
    scores.append(acc)
    histories.append(history)
  return scores, histories

# plot diagnostic learning curves
def summarize_diagnostics(histories):
  for i in range(len(histories)):
    # plot loss
    pyplot.subplot(211)
    pyplot.title('Cross Entropy Loss')
    pyplot.plot(histories[i].history['loss'], color='blue', label='train')
    pyplot.plot(histories[i].history['val_loss'], color='orange', label='test')
    # plot accuracy
    pyplot.subplot(212)
    pyplot.title('Classification Accuracy')
    pyplot.plot(histories[i].history['acc'], color='blue', label='train')
    pyplot.plot(histories[i].history['val_acc'], color='orange', label='test')
  pyplot.show()

# summarize model performance
def summarize_performance(scores):
  # print summary
  print('Accuracy: mean=%.3f std=%.3f, n=%d' % (mean(scores)*100, std(scores)*100,
      len(scores)))
  # box and whisker plots of results
  pyplot.boxplot(scores)
  pyplot.show()

# run the test harness for evaluating a model
def run_test_harness():
  # load dataset
  trainX, trainY, testX, testY = load_dataset()
  # prepare pixel data
  trainX, testX = prep_pixels(trainX, testX)
  # evaluate model
  scores, histories = evaluate_model(trainX, trainY)
  # learning curves
  summarize_diagnostics(histories)
  # summarize estimated performance
  summarize_performance(scores)

# entry point, run the test harness
run_test_harness()
```

Listing 19.23: Example of evaluating the baseline model with padded convolutional layers and more filters.

Running the example reports model performance for each fold of the cross-validation process.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, the per-fold scores may suggest some further improvement over the baseline and using same padding alone.

```
> 90.917
> 90.908
> 90.175
> 91.158
> 91.408
```

Listing 19.24: Example output from during the evaluation of each model.

A plot of the learning curves is created, in this case showing that the models still have a reasonable fit on the problem, with a small sign of some of the runs overfitting.

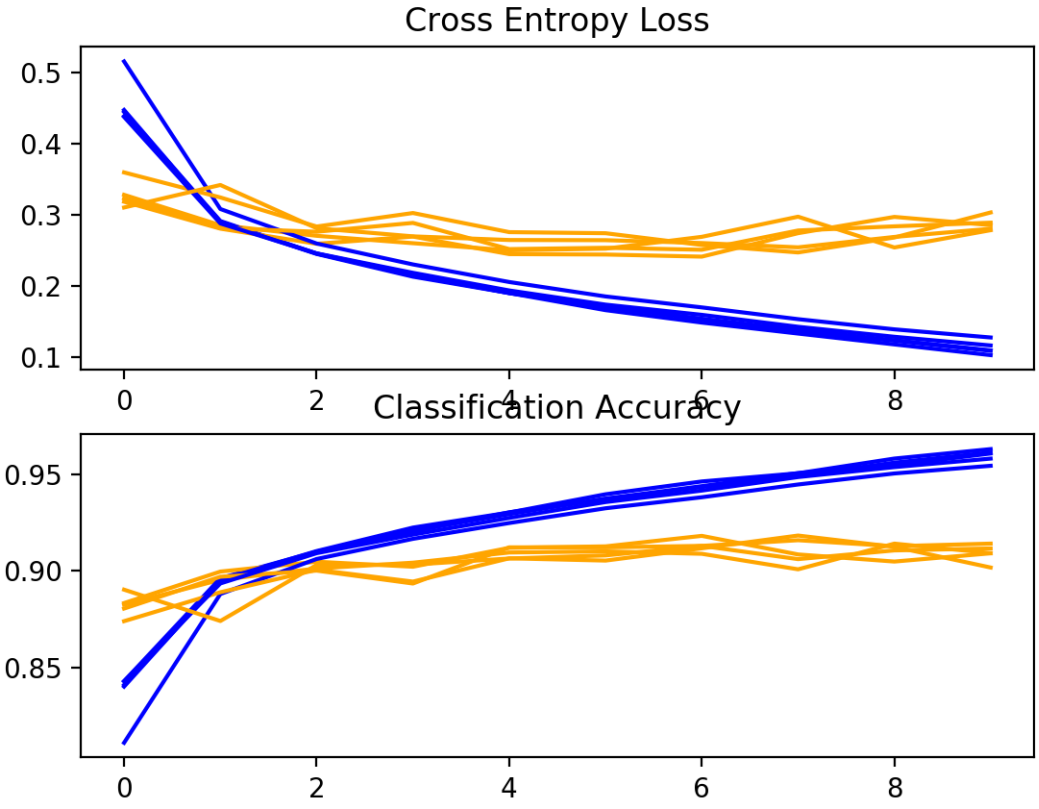

Figure 19.6: Loss and Accuracy Learning Curves for the More Filters and Padding on the Fashion-MNIST Dataset During k-Fold Cross-Validation.

Next, the estimated performance of the model is presented, showing a possible decrease in performance as compared to the baseline with padding from 90.913% to 91.257%. Again, the change is still within the bounds of the standard deviation, and it is not clear whether the effect is real.

```
Accuracy: mean=90.913 std=0.412, n=5
```

Listing 19.25: Example output from the final evaluation of the model.

19.6 How to Finalize the Model and Make Predictions

The process of model improvement may continue for as long as we have ideas and the time and resources to test them out. At some point, a final model configuration must be chosen and adopted. In this case, we will keep things simple and use the baseline model as the final model. First, we will finalize our model, by fitting a model on the entire training dataset and saving the model to file for later use. We will then load the model and evaluate its performance on the hold out test dataset, to get an idea of how well the chosen model actually performs in practice. Finally, we will use the saved model to make a prediction on a single image.

19.6.1 Save Final Model

A final model is typically fit on all available data, such as the combination of all train and test dataset. In this tutorial, we are intentionally holding back a test dataset so that we can estimate the performance of the final model, which can be a good idea in practice. As such, we will fit our model on the training dataset only.

```
...
# fit model
model.fit(trainX, trainY, epochs=10, batch_size=32, verbose=0)
```

Listing 19.26: Example of fitting the final model.

Once fit, we can save the final model to an h5 file by calling the `save()` function on the model and passing in the chosen filename.

```
...
# save model
model.save('final_model.h5')
```

Listing 19.27: Example of saving the final model.

Note: saving and loading a Keras model requires that the `h5py` library is installed on your workstation. The complete example of fitting the final model on the training dataset and saving it to file is listed below.

```
# save the final model to file
from keras.datasets import fashion_mnist
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import Dense
from keras.layers import Flatten
from keras.optimizers import SGD

# load train and test dataset
def load_dataset():
```

```
  # load dataset
  (trainX, trainY), (testX, testY) = fashion_mnist.load_data()
  # reshape dataset to have a single channel
  trainX = trainX.reshape((trainX.shape[0], 28, 28, 1))
  testX = testX.reshape((testX.shape[0], 28, 28, 1))
  # one hot encode target values
  trainY = to_categorical(trainY)
  testY = to_categorical(testY)
  return trainX, trainY, testX, testY

# scale pixels
def prep_pixels(train, test):
  # convert from integers to floats
  train_norm = train.astype('float32')
  test_norm = test.astype('float32')
  # normalize to range 0-1
  train_norm = train_norm / 255.0
  test_norm = test_norm / 255.0
  # return normalized images
  return train_norm, test_norm

# define cnn model
def define_model():
  model = Sequential()
  model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',
      input_shape=(28, 28, 1)))
  model.add(MaxPooling2D((2, 2)))
  model.add(Flatten())
  model.add(Dense(100, activation='relu', kernel_initializer='he_uniform'))
  model.add(Dense(10, activation='softmax'))
  # compile model
  opt = SGD(lr=0.01, momentum=0.9)
  model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
  return model

# run the test harness for evaluating a model
def run_test_harness():
  # load dataset
  trainX, trainY, testX, testY = load_dataset()
  # prepare pixel data
  trainX, testX = prep_pixels(trainX, testX)
  # define model
  model = define_model()
  # fit model
  model.fit(trainX, trainY, epochs=10, batch_size=32, verbose=0)
  # save model
  model.save('final_model.h5')

# entry point, run the test harness
run_test_harness()
```

Listing 19.28: Example of fitting and saving the final model.

After running this example, you will now have a 4.2-megabyte file with the name `final_model.h5` in your current working directory.

19.6.2 Evaluate Final Model

We can now load the final model and evaluate it on the hold out test dataset. This is something we might do if we were interested in presenting the performance of the chosen model to project stakeholders. The model can be loaded via the `load_model()` function. The complete example of loading the saved model and evaluating it on the test dataset is listed below.

```
# evaluate the deep model on the test dataset
from keras.datasets import fashion_mnist
from keras.models import load_model
from keras.utils import to_categorical

# load train and test dataset
def load_dataset():
  # load dataset
  (trainX, trainY), (testX, testY) = fashion_mnist.load_data()
  # reshape dataset to have a single channel
  trainX = trainX.reshape((trainX.shape[0], 28, 28, 1))
  testX = testX.reshape((testX.shape[0], 28, 28, 1))
  # one hot encode target values
  trainY = to_categorical(trainY)
  testY = to_categorical(testY)
  return trainX, trainY, testX, testY

# scale pixels
def prep_pixels(train, test):
  # convert from integers to floats
  train_norm = train.astype('float32')
  test_norm = test.astype('float32')
  # normalize to range 0-1
  train_norm = train_norm / 255.0
  test_norm = test_norm / 255.0
  # return normalized images
  return train_norm, test_norm

# run the test harness for evaluating a model
def run_test_harness():
  # load dataset
  trainX, trainY, testX, testY = load_dataset()
  # prepare pixel data
  trainX, testX = prep_pixels(trainX, testX)
  # load model
  model = load_model('final_model.h5')
  # evaluate model on test dataset
  _, acc = model.evaluate(testX, testY, verbose=0)
  print('> %.3f' % (acc * 100.0))

# entry point, run the test harness
run_test_harness()
```

Listing 19.29: Example of loading and evaluating the final model.

Running the example loads the saved model and evaluates the model on the hold out test dataset. The classification accuracy for the model on the test dataset is calculated and printed.

Note: Your specific results may vary given the stochastic nature of the learning algorithm.

Consider running the example a few times and compare the average performance.

In this case, we can see that the model achieved an accuracy of 90.990%, or just less than 10% classification error, which is not bad.

Note: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

```
> 90.990
```

Listing 19.30: Example output from loading and evaluating the final model.

19.6.3 Make Prediction

We can use our saved model to make a prediction on new images. The model assumes that new images are grayscale, they have been segmented so that one image contains one centered piece of clothing on a black background, and that the size of the image is square with the size 28×28 pixels. Below is an image extracted from the MNIST test dataset.

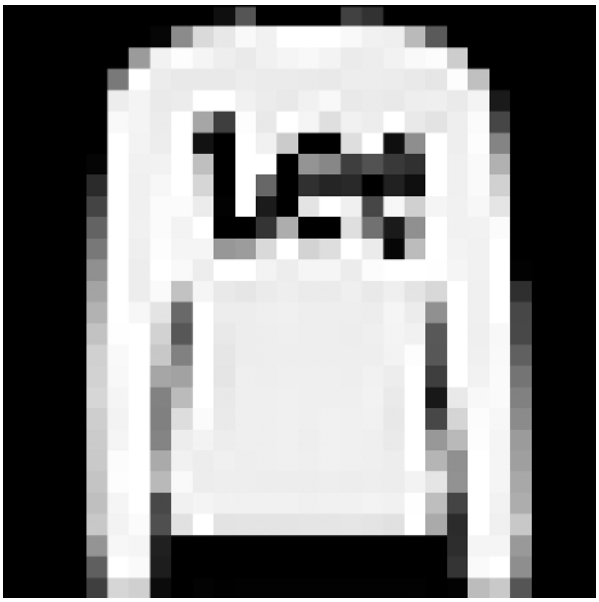

Figure 19.7: Sample Clothing (Pullover).

You can save it in your current working directory with the filename `sample_image.png`.

- Download Image (`sample_image.png`).[1]

We will pretend this is an entirely new and unseen image, prepared in the required way, and see how we might use our saved model to predict the integer that the image represents. For this example, we expect class `2` for `Pullover` (also called a jumper). First, we can load the image, force it to be grayscale format, and force the size to be 28×28 pixels. The loaded image

[1] https://machinelearningmastery.com/wp-content/uploads/2019/05/sample_image.png

can then be resized to have a single channel and represent a single sample in a dataset. The `load_image()` function implements this and will return the loaded image ready for classification. Importantly, the pixel values are prepared in the same way as the pixel values were prepared for the training dataset when fitting the final model, in this case, normalized.

```
# load and prepare the image
def load_image(filename):
  # load the image
  img = load_img(filename, grayscale=True, target_size=(28, 28))
  # convert to array
  img = img_to_array(img)
  # reshape into a single sample with 1 channel
  img = img.reshape(1, 28, 28, 1)
  # prepare pixel data
  img = img.astype('float32')
  img = img / 255.0
  return img
```

Listing 19.31: Example of a function for loading and preparing an image for prediction.

Next, we can load the model as in the previous section and call the `predict_classes()` function to predict the clothing in the image.

```
...
# predict the class
result = model.predict_classes(img)
```

Listing 19.32: Example of making a prediction with a prepared image.

The complete example is listed below.

```
# make a prediction for a new image.
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from keras.models import load_model

# load and prepare the image
def load_image(filename):
  # load the image
  img = load_img(filename, grayscale=True, target_size=(28, 28))
  # convert to array
  img = img_to_array(img)
  # reshape into a single sample with 1 channel
  img = img.reshape(1, 28, 28, 1)
  # prepare pixel data
  img = img.astype('float32')
  img = img / 255.0
  return img

# load an image and predict the class
def run_example():
  # load the image
  img = load_image('sample_image.png')
  # load model
  model = load_model('final_model.h5')
  # predict the class
  result = model.predict_classes(img)
```

```
  print(result[0])

# entry point, run the example
run_example()
```

Listing 19.33: Example of loading and making a prediction with the final model.

Running the example first loads and prepares the image, loads the model, and then correctly predicts that the loaded image represents a pullover or class 2.

```
2
```

Listing 19.34: Example output from loading and making a prediction with the final model.

19.7 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Regularization**. Explore how adding regularization impacts model performance as compared to the baseline model, such as weight decay, early stopping, and dropout.
- **Tune the Learning Rate**. Explore how different learning rates impact the model performance as compared to the baseline model, such as 0.001 and 0.0001.
- **Tune Model Depth**. Explore how adding more layers to the model impacts the model performance as compared to the baseline model, such as another block of convolutional and pooling layers or another dense layer in the classifier part of the model.

If you explore any of these extensions, I'd love to know.

19.8 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

19.8.1 APIs

- Keras Datasets API.
  https://keras.io/datasets/
- Keras Datasets Code.
  https://github.com/keras-team/keras/tree/master/keras/datasets
- `sklearn.model_selection.KFold` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.KFold.html

19.8.2 Articles

- Fashion-MNIST GitHub Repository.
  https://github.com/zalandoresearch/fashion-mnist