5

Classification

**Classification** is a supervised learning method for predicting a class label for a given example of input data. Although we introduced classification with MNIST, we work through the famous Fashion-MNIST dataset to delve deeper into the topic.

Fashion-MNIST is intended as a direct replacement for MNIST to better benchmark machine learning algorithms. It shares the same image size and structure of training and test splits, but is a more challenging classification problem.

MNIST benchmarking has several associated problems. It’s far too easy for standard machine learning algorithms to achieve over 97% accuracy. It’s even easier for deep learning models to achieve over 99% accuracy. The dataset is overused. Finally, MINST cannot represent modern computer vision tasks.

Notebooks for this chapter are located at the following URL: <https://github.com/paperd/tensorflow>.

We need to set up the GPU for each notebook. So, click the Runtime tab, click Change runtime type from the drop-down menu, choose GPU from the Hardware Accelerator drop-down menu, and click Save.

Display the current version of TensorFlow and enable the GPU in Google Colab:

import tensorflow as tf

# display tf version and test if GPU is active

tf.\_\_version\_\_, tf.test.gpu\_device\_name()

The GPU is active if you see '/device:GPU:0' displayed.

# Fashion-MNIST Dataset

**Fashion-MNIST** is a dataset of clothing article images created by Zalando Research consisting of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28×28 grayscale image associated with a label from 10 classes.

Zalando Research is an organization that utilizes an agile design process that combines invaluable human experience with the power of machine learning. Zalando concentrates on exploring novel ways to use generative models in fashion design for rapid visualizations and prototyping.

# Load Fashion-MNIST as a TFDS

Since Fashion-MNIST is a tfds.data.Dataset, we can easily load it with tfds.load(). To get a list of all TFDS datasets, just run the tfds.list\_builders() method as demonstrated in chapter 3.

Load train and test examples as a tf.data.Dataset:

import tensorflow\_datasets as tfds

train, info = tfds.load('fashion\_mnist', split='train',

                        with\_info=True, shuffle\_files=True)

test = tfds.load('fashion\_mnist', split='test')

Since we already have info from the train data, we don’t need to load it again for the test data.

Verify train and test data:

train.element\_spec, test.element\_spec

Each image consists of 28 x 28 pixels. The 1 dimension indicates that images are grayscale. Each label is a scalar.

## Explore the Dataset

Display information about the dataset:

info

We see the name, description, and homepage. We also see the shape and datatype of feature images and labels. We see that we have 70,000 examples with train and test splits of 60,000 and 10,000 respectively. A lot of other information is also included.

Extract number of classes and class labels:

br = '\n'

num\_classes = info.features['label'].num\_classes

class\_labels = info.features['label'].names

print ('number of classes:', num\_classes, br)

print ('class labels:', class\_labels)

We have 10 classes representing ten clothing articles.

Let’s see some training examples:

fig = tfds.show\_examples(info, train)

The show\_examples() method displays sample images and labels. The label name and associated class number is displayed under each image. For example, the class number for Pullover is 2 because it is the third label in the class labels list.

Listing 5-1 is a custom function that displays samples from the train dataset:

import matplotlib.pyplot as plt, numpy as np

def display\_samples(data, num, cmap):

  for example in data.take(num):

    image, label = example['image'], example['label']

    print ('Label:', class\_labels[label.numpy()], end=', ')

    print ('Index:', label.numpy())

    plt.imshow(image.numpy()[:, :, 0].astype(np.float32),

               cmap=plt.get\_cmap(cmap))

    plt.show()

Listing 5-1. Custom function for displaying sample data

Import a couple of libraries. The function accepts a dataset, number of samples to display, and a colormap. A **colormap** is an array of colors used to map pixel data to the actual color values. The matplotlib library provides a variety of built-in colormaps.

We assign an example image and label to variables. We then display the label name and its associated class number. We end by displaying the image with the imshow() function. We use [:, :, 0] to grab all pixels from each image.

Invoke the function to display a couple of training images:

# choose colormap by changing 'indx'

indx = 5

cmap = ['coolwarm', 'viridis', 'plasma',

        'seismic', 'copper', 'twilight']

samples = 2

display\_samples(train, samples, cmap[indx])

Now, let’s build a custom function to display a grid of samples.

Before we create the function, take 30 samples from the train set as shown in Listing 5-2:

num = 30

images, labels = [], []

for example in train.take(num):

  image, label = example['image'], example['label']

  images.append(tf.squeeze(image.numpy()))

  labels.append(tf.squeeze(label.numpy()))

Listing 5-2. Take samples from the train set

Create the function as shown in Listing 5-3:

def display\_grid(feature, target, n\_rows, n\_cols, cl):

  plt.figure(figsize=(n\_cols \* 1.5, n\_rows \* 1.5))

  for row in range(n\_rows):

    for col in range(n\_cols):

      index = n\_cols \* row + col

      plt.subplot(n\_rows, n\_cols, index + 1)

      plt.imshow(feature[index], cmap='binary',

                 interpolation='nearest')

      plt.axis('off')

      plt.title(cl[target[index]], fontsize=12)

  plt.subplots\_adjust(wspace=0.2, hspace=0.5)

Listing 5-3. Function that displays a grid of examples

Invoke the function:

rows = 5

cols = 6

display\_grid(images, labels, rows, cols, class\_labels)

We can also pinpoint metadata with DatasetInfo:

print ('Number of training examples:', end=' ')

print (info.splits['train'].num\_examples)

print ('Number of test examples:', end=' ')

print (info.splits['test'].num\_examples)

## Build the Input Pipeline

Build the input pipeline for train and test data as shown in Listing 5-4:

bs = 128

ss = 5000

train\_fs = train.shuffle(ss).batch(bs)

train\_fs = train\_fs.map(lambda items: (

    tf.cast(items['image'], tf.float32) / 255., items['label']))

train\_fs = train\_fs.cache().prefetch(1)

test\_fs = test.batch(bs)

test\_fs = test\_fs.map(lambda items: (

    tf.cast(items['image'], tf.float32) / 255., items['label']))

test\_fs = test\_fs.cache().prefetch(1)

Listing 5-4. Build the input pipeline

Shuffle and batch train data. Scale train images. Cache and prefetch train images. Batch test data. Scale test images. Cache and prefetch test images. Use batch size of 128 and shuffle buffer size of 5,000 for this experiment.

**Note:** Do not shuffle test data because it is considered new to the neural network model.

Caching a TFDS can significantly improve performance. The cache method of a tf.data.Dataset can cache a dataset either in memory or on local storage, which saves operations like file opening and data reading from being executed during each epoch.

Adding prefectch(1) is a good idea because it adds efficiency to the batching process. While our training algorithm is working on one batch, TensorFlow is working on the dataset in parallel to get the next batch ready. So, prefectch can dramatically improve training performance.

For more information on TFDS performance improvement peruse:

* <https://www.tensorflow.org/datasets/performances>
* <https://www.tensorflow.org/guide/data_performance>

Verify train and test tensors:

train\_fs.element\_spec, test\_fs.element\_spec

Both train and test images are 28 x 28 x 1. The 1 value means that images are grayscale. That is, images are in black and white.

## Build the Model

Let’s build a simple feedforward neural net as shown in Listing 5-5:

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Flatten, Dropout

from tensorflow import keras

# clear previous model and plant a seed

keras.backend.clear\_session()

np.random.seed(0)

tf.random.set\_seed(0)

model = Sequential([

  Flatten(input\_shape=[28, 28, 1]),

  Dense(512, activation='relu'),

  Dropout(0.4),

  Dense(10, activation='softmax')

])

Listing 5-5. Simple feedforward neural network

Import requisite libraries. Clear any previous model sessions and plant random seeds to facilitate reproducibility of results. The first layer flattens images. The second layer uses relu activation on 512 neurons to process the data. The third layer uses dropout to reduce overfitting. The fourth layer uses softmax activation on 10 neurons to account for class labels.

## Compile the Model

Define a gradient descent optimizer that tweaks model parameters to minimize the loss function:

model.compile(optimizer='adam',

              loss='sparse\_categorical\_crossentropy',

              metrics=['accuracy'])

Display a summary of the model:

model.summary()

Output shape of the first layer is (None, 784). We get 784 by multiplying 28 by 28. We have no parameters at this layer because it is only used to bring data into the model. Output shape of the second layer is (None, 512) because we have 512 neurons at this layer. We get parameters of 401,920 by multiplying 512 neurons at this layer by 784 neurons from the previous layer, and adding 512 at this layer. Output shape of the third layer is (None, 512) and parameters are 0 because dropout doesn’t impact neurons or parameters. Output shape of the fourth layer is (None, 10) because we have 10 neurons at this layer to deal with 10 output classes. We get parameters of 5,130 by multiplying 10 neurons at this layer by 512 from the previous layer, and adding 10 neurons at this layer.

**Note:** None is used because TensorFlow models can accept any batch size.

We can also display information about the layers:

model.layers

## Train the Model

Train the model with 10 epochs:

epochs = 10

history = model.fit(train\_fs, epochs=epochs,

                    verbose=1, validation\_data=test\_fs)

We get pretty good accuracy with not much overfitting.

## Generalize on Test Data

Although training provides accuracy and loss metrics, it is always a good idea to explicitly evaluate the model on test data to see how well the model generalizes to new data:

print('Test accuracy:', end=' ')

test\_loss, test\_acc = model.evaluate(test\_fs, verbose=2)

## Visualize Performance

The fit method automatically records the history of the training process as a dictionary. So, we can assign training information to a variable. In this case, we assign it to history. The history attribute of the variable contains the dictionary information.

Display the dictionary keys to inform us on how to plot the results:

hist\_dict = history.history

print (hist\_dict, '\n')

print (hist\_dict.keys())

The dictionary history.history contains loss, accuracy, val\_loss, and val\_accuracy metrics that the model measures at the end of each epoch on the training set and validation (or test) set.

The params method provides all parameters involved with training:

history.params

The epoch method displays the number of epochs:

history.epoch

Plot training performance as shown in Listing 5-6:

plt.plot(history.history['accuracy'], label='accuracy')

plt.plot(history.history['val\_accuracy'], label = 'val\_accuracy')

plt.xlabel('Epoch')

plt.ylabel('Accuracy')

plt.ylim([0.5, 1])

plt.legend(loc='lower right')

plt.show()

plt.plot(history.history['loss'], label='loss')

plt.plot(history.history['val\_loss'], label = 'val\_loss')

plt.xlabel('Epoch')

plt.ylabel('Loss')

plt.ylim([0.05, .7])

plt.legend(loc='lower right')

plt.show()

Listing 5-6. Plot training performance

We don’t have much overfitting and our model accuracy is pretty good for such a simple neural network.

We can also use pandas to plot training performance:

import pandas as pd

pd.DataFrame(history.history).plot(figsize=(8, 5))

plt.grid(True)

plt.gca().set\_ylim(0, 1)

## Predict Labels for Test Images

Now that we have a trained model, we can make predictions based on test images. We predict from test images because the model sees these images as new data.

Begin by predicting the label for each image in the test set:

tf.random.set\_seed(0)

predictions = model.predict(test\_fs)

We use the predict method on the processed test set test\_fs.

Since Fashion-MNIST has ten class labels, each prediction consists of an array of 10 numbers that represent the model’s confidence in how well the image corresponds to each of the 10 different articles of clothing.

Let’s take a look at the first prediction:

predictions[0]

The first prediction is at index 0 because Python indexes range from 0 to 9999 for test set size of 10,000. It’s hard to tell which of the values has the highest number by looking at the array of float number values.

Round the numbers in the predictions array to make it easier to see the position in the array with the highest confidence:

np.round(predictions[0], 2)

Now, we can clearly see the value with the highest number. The position with the highest confidence corresponds to the position in the class labels array.

We can directly derive the confidence in the prediction with the following code:

c = 100\*np.max(predictions[0])

Convert the value to a percentage:

str(np.round(c, 2)) + '%'

Display the prediction for the first image in the test set:

np.argmax(predictions[0])

The prediction can only be between 0 and 9 because our target class consists of ten classes.

Use the class\_labels array we created earlier to see the actual fashion article:

class\_labels[np.argmax(predictions[0])]

So, we have the predicted clothing article for the first image in the test set.

Display the first actual test image:

# take the first batch of images

for image, label in test\_fs.take(1):

  label

class\_labels[label[0].numpy()]

We take the first image from a batch of 128 images because we set batch size to 128. If the prediction matches the test image, it was a correct prediction.

## Build a Prediction Plot

Now that we have a trained model, we can build a prediction plot.

Since we want to see how well the model performs on new data, take 20 samples from the test set as shown in Listing 5-7:

num = 20

images, labels = [], []

for example in test.take(num):

  image, label = example['image'], example['label']

  images.append(tf.squeeze(image.numpy()))

  labels.append(tf.squeeze(label.numpy()))

Listing 5-7. Take samples from the test set

Squeeze each image to remove the 1 dimension for plotting purposes.

Build a plotting function as shown in Listing 5-8:

def display\_test(feature, target, num\_images,

                 n\_rows, n\_cols, cl, p):

  for i in range(num\_images):

    plt.subplot(n\_rows, 2\*n\_cols, 2\*i+1)

    if cl[target[i]] != cl[np.argmax(p[i])]:

      plt.imshow(feature[i], cmap='Reds')

    else:

      plt.imshow(feature[i], cmap='Blues')

    val = 100\*np.max(p[i])

    rounded = str(np.round(val, 2)) + '%'

    plt.title(cl[target[i]] + ' (' +\

              cl[np.argmax(p[i])] + ') ' +\

              rounded )

  plt.tight\_layout()

plt.show()

Listing 5-8. Function to build a prediction plot

Invoke the function as shown in Listing 5-9:

num\_rows, num\_cols = 5, 4

num\_images = num\_rows\*num\_cols

plt.figure(figsize=(2\*2\*num\_cols, 2\*num\_rows))

display\_test(images, labels, num\_images, num\_rows,

             num\_cols, class\_labels, predictions)

Listing 5-9. Invoke the prediction plot function

Any clothing article in red means that the prediction was incorrect. Above each article of clothing is the actual label, prediction in parentheses, and confidence in the prediction.

# Load Fashion-MNIST as a Keras Dataset

Although TFDS are recommended for TensorFlow 2.x application, we show you how to work with a Keras dataset because of its popularity in industry. The newness of TensorFlow 2.x means that it doesn't yet have the industry penetration of Keras.

Load the Keras dataset:

train, test = tf.keras.datasets.fashion\_mnist.load\_data()

## Explore the Data

Let’s look at the data shapes:

print ('train data:', br)

print (train[0].shape)

print (train[1].shape, br)

print ('test data:', br)

print (test[0].shape)

print (test[1].shape)

The Keras dataset contains the same data as the Fashion-MNIST TFDS. Train data consists of 60,000 28x28 feature images and 60,000 labels. Test data consists of 10,000 28x28 feature images and 10,000 labels. Train images are contained in the train tuple. So, train[0] represents images and train[1] represents labels.

Let’s see what the first image represents:

class\_labels[train[1][0]]

Since we access training labels with train[1], we grab the first label with train[1][0].

## Visualize the First Image

Plot the first image with Matplotlib’s imshow() function:

plt.imshow(train[0][0], cmap='binary')

plt.axis('off')

Assign images and labels to variables for convenience:

train\_images = train[0]

train\_labels = train[1]

Plot the first image based on the train\_images variable:

plt.imshow(train\_images[0], cmap='binary')

plt.axis('off')

Access the label name of the first image:

class\_labels[train\_labels[0]]

Labels are the class IDs ranging from 0 to 9:

train\_labels

## Visualize Sample Images

Since we can’t use the TFDS show\_examples method, we create code as shown in Listing 5-10:

n\_rows = 5

n\_cols = 6

plt.figure(figsize=(n\_cols \* 1.5, n\_rows \* 1.5))

for row in range(n\_rows):

  for col in range(n\_cols):

    index = n\_cols \* row + col

    plt.subplot(n\_rows, n\_cols, index + 1)

    plt.imshow(train\_images[index], cmap='binary',

               interpolation='nearest')

    plt.axis('off')

    plt.title(class\_labels[train\_labels[index]], fontsize=12)

plt.subplots\_adjust(wspace=0.2, hspace=0.5)

Listing 5-10. Code to visualize examples

## Prepare Data for Training

To prepare training data for TensorFlow consumption, we need to grab images and labels from train and test data. We scale images to ensure that each input parameter (a pixel, in our case) has a similar data distribution. The distribution of such data resembles a Gaussian curve centered at zero. Scaling data makes convergence faster while training the network.

As we know, pixel data is represented by a range from 0 to 255. So, we divide each feature image by 255 to scale it. Once images are scaled, we convert images and labels to a tf.Tensor object with from\_tensor\_slices as shown in Listing 5-11:

# add test images and labels to the mix

test\_images, test\_labels = test

train\_pictures = train\_images / 255.  # divide by 255 to scale

train\_targets = train\_labels.astype(np.int32)

test\_pictures = test\_images / 255.  # divide by 255 to scale

test\_targets = test\_labels.astype(np.int32)

print ('train images:', len(train\_pictures))

print ('train labels:', len(train\_targets), br)

print ('test images', len(test\_pictures))

print ('test labels', len(test\_targets))

train\_ds = tf.data.Dataset.from\_tensor\_slices(

    (train\_pictures, train\_targets))

test\_ds = tf.data.Dataset.from\_tensor\_slices(

    (test\_pictures, test\_targets))

Listing 5-11. Prepare the input pipeline

Display the tensors:

train\_ds.element\_spec, test\_ds.element\_spec

Finish the input pipeline by shuffling, batching, and prefetching train data, and batching and prefetching test data:

BATCH\_SIZE = 128

SHUFFLE\_BUFFER\_SIZE = 5000

train\_ks = train\_ds.shuffle(

    SHUFFLE\_BUFFER\_SIZE).batch(BATCH\_SIZE).prefetch(1)

test\_ks = test\_ds.batch(BATCH\_SIZE).prefetch(1)

Set BATCH\_SIZE to 128 so the model will run faster than with a smaller batch size. Experiment with this number and see what happens. We also set SHUFFLE\_BUFFER\_SIZE to 5000 so the shuffle method works well. Again, experiment with this number to see what happens.

Display the finalized input pipeline tensors:

train\_ks.element\_spec, test\_ks.element\_spec

## Build the Model

Create the same model that with did for the TFDS as shown in Listing 5-12:

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Flatten, Dropout

from tensorflow import keras

# clear previous model and plant a seed

keras.backend.clear\_session()

np.random.seed(0)

tf.random.set\_seed(0)

model = Sequential([

  Flatten(input\_shape=[28, 28]),

  Dense(512, activation='relu'),

  Dropout(0.4),

  Dense(10, activation='softmax')

])

Listing 5-12. Keras model

As we already know, it is always a good idea to clear any previous modeling sessions. Remember that we already created a model and trained it earlier in the chapter. Also, planting a random seed ensures that the results are consistent. In machine learning, such consistency is called reproducibility. That is, the seed allows us to reproduce results in a consistent manner. You can use any integer value for the random seed number. We use 0.

The Flatten() layer flattens the 2D matrix of 28x28 images to a 1D array of 784 pixels. The second layer is the first true layer. It accepts input images into 128 neurons and performs relu activation. The final layer is the output layer. It accepts output from the input layer with 10 neurons that represent the ten classes of clothing articles and performs softmax activation.

## Compile the Model

model.compile(optimizer='adam',

              loss='sparse\_categorical\_crossentropy',

              metrics=['accuracy'])

## Train the Model

Fit the model. Passing a dataset of (feature, label) pairs is all that's needed for Model.fit and Model.evaluate:

history = model.fit(train\_ks, epochs=10, validation\_data=test\_ks)

## Model Summary

model.summary()

## Model Layers

model.layers

## Generalize on Test Data

Although model fit information provides validation loss and accuracy during training, it is always a good idea to explicitly evaluate the model on test data because accuracy and loss values may differ:

print('Test accuracy:', end=' ')

test\_loss, test\_acc = model.evaluate(test\_ks, verbose=2)

## Visualize Training

The fit method automatically records the history of the training process as a dictionary. So, we can assign training history to a variable. In this case, we assigned it to history. The history attribute of the variable contains the dictionary information.

Display the dictionary keys to inform us on how to plot results:

hist\_dict = history.history

print (hist\_dict, '\n')

print (hist\_dict.keys())

Dictionary history.history contains loss and other metrics the model measures at the end of each epoch on the training set and validation set.

The params method provides all parameters involved with the model:

history.params

The epoch method contains epoch history:

history.epoch

Listing 5-13 plots training history:

plt.plot(history.history['accuracy'], label='accuracy')

plt.plot(history.history['val\_accuracy'], label = 'val\_accuracy')

plt.xlabel('Epoch')

plt.ylabel('Accuracy')

plt.ylim([0.5, 1])

plt.legend(loc='lower right')

plt.show()

plt.plot(history.history['loss'], label='loss')

plt.plot(history.history['val\_loss'], label = 'val\_loss')

plt.xlabel('Epoch')

plt.ylabel('Loss')

plt.ylim([0.05, .7])

plt.legend(loc='lower right')

plt.show()

Listing 5-13. Training history plots

Overfitting is minimal because training accuracy is closely aligned with test accuracy. We did employ dropout in the model to reduce overfitting. **Dropout** is a regularization method that approximates training a large number of neural networks with different architectures in parallel. The definition doesn't make it easier to understand for anyone new to deep learning. So, let's explain how it works.

During training, some number of layer outputs are randomly ignored or dropped out. By randomly dropping out layer outputs, effectively the layer looks like and is treated like a layer with a different number of nodes and connectivity to the prior layer. So, each update to a layer during training is performed with a different view of the configured layer. Dropout is a simple, but very effective technique to reduce overfitting.

Setting dropout at 0.4 means that we randomly dropped out 40% of the layer outputs. You can easily experiment with dropout by changing this value. However, you should keep dropout rate at or less than 0.5 because otherwise you are removing too much data!

**Tip:** Experiment with dropout levels, but keep it at or less than 0.5 to avoid removing too much data.

If the model is overfitting (train accuracy is greater than test accuracy) with the dropout value you set, you can increase it. If the model is underfitting (train accuracy is less than test accuracy), you can decrease it.

Use Pandas to plot:

import pandas as pd

pd.DataFrame(history.history).plot(figsize=(8, 5))

plt.grid(True)

plt.gca().set\_ylim(0, 1)

## Predict Labels for Test Images

We can make predictions on test images since we have a trained model.

Predict the label for each image in the test set:

predictions = model.predict(test\_ks)

As with the TFDS, a prediction is an array of 10 numbers that represent the model’s confidence that the image corresponds to each of the 10 articles of clothing.

### Predict the First Image

Let’s look at the first prediction:

predictions[0]

Round the array values for convenience:

np.round(predictions[0], 2)

Directly access the confidence in the prediction:

100\*np.max(predictions[0])

Convert the value to a percentage:

str(np.round(100\*np.max(predictions[0]), 2)) + '%'

Display the prediction for the first image:

np.argmax(predictions[0])

The prediction can only be between 0 and 9 because our target consists of ten classes.

Use the class\_labels array created earlier to see the actual fashion article:

class\_labels[np.argmax(predictions[0])]

Display the first actual test label for comparison:

class\_labels[test\_labels[0]]

If the prediction and actual images match, the prediction is correct.

### Predict Four Images

Make four predictions from the test set:

pred\_4 = predictions[:4]

Show the predictions:

ls = [np.argmax(row) for row in pred\_4]

ls

Get class labels:

np.array(class\_labels)[ls]

Compare predictions to the test dataset:

actual\_4 = test\_labels[:4]

actual\_4

Show actuals as class labels:

np.array(class\_labels)[actual\_4]

Visualize as shown in Listing 5-14:

# slice off the first four images from the test data

img\_4 = test\_images[:4]

# plot images

plt.figure(figsize=(7.2, 2.4))

for index, image in enumerate(img\_4):

  plt.subplot(1, 4, index + 1)

  plt.imshow(image, cmap='twilight', interpolation='nearest')

  plt.axis('off')

  plt.title(class\_labels[actual\_4[index]], fontsize=12)

plt.subplots\_adjust(wspace=0.2, hspace=0.5)

Listing 5-14. Visualize the first four actual test images

## Explore Misclassifications

Let’s explore predictions and actual test image labels to find some misclassifications.

The first step is to identify prediction labels:

rng = (len(predictions))

y\_pred = [np.argmax(row) for row in predictions]

The y\_pred variable holds a list of prediction labels.

Next, grab some predictions and actual targets:

# find first n predictions and actual targets

n = 20

y\_n = [y\_pred[i] for i, row in enumerate(range(n))]

y\_actual = [test\_labels[i] for i, row in enumerate(range(n))]

y\_n, y\_actual

Find misclassifications:

# compare predictions against actual targets

miss\_indx\_list = [index for index, (x, y) in

                  enumerate(zip(y\_n, y\_actual))

                  if x != y]

miss\_indx\_list

If you don’t get any misclassifications, increase the size of n and rerun the last two code snippets.

We now have an array of misclassifications by their index.

Listing 5-15 displays the confidence for each misclassification:

# display confidence for each misclassfication:

for row in miss\_indx\_list:

  val = 100\*np.max(predictions[row])

  rounded = str(round(val, 2)) + '%'

  print ('index:', row, 'confidence:', rounded,

         'pred:', class\_labels[np.argmax(predictions[row])],

         'actual:', class\_labels[test\_labels[row]])

Listing 5-15. Confidence in misclassifications

## Visualize Misclassifications

Create a function to visualize misclassifications as shown in Listing 5-16:

def see\_misses(indx):

  plt.imshow(test\_images[indx], cmap='nipy\_spectral')

  plt.show()

  print ('actual:', class\_labels[test\_labels[indx]])

  print ('predicted:',

         class\_labels[np.argmax(predictions[indx])])

  print ('confidence', rounded)

Listing 5-16. Function to visualize misclassifications

Invoke the function:

for row in miss\_indx\_list:

  val = 100\*np.max(predictions[row])

  rounded = str(round(val, 2)) + '%'

  see\_misses(row)

The confidence may still be pretty high even though the prediction was incorrect.

Let’s create a more sophisticated visualization as shown in Listing 5-17:

# Plot the first X test images, their true labels,

# their predicted labels, and prediction confidence.

num\_rows = 5

num\_cols = 4

num\_images = num\_rows\*num\_cols

plt.figure(figsize=(2\*2\*num\_cols, 2\*num\_rows))

for i in range(num\_images):

  plt.subplot(num\_rows, 2\*num\_cols, 2\*i+1)

  if class\_labels[test\_labels[i]] != class\_labels[y\_pred[i]]:

    plt.imshow(test\_images[i], cmap='Reds')

  else:

    plt.imshow(test\_images[i], cmap='Blues')

  val = 100\*np.max(predictions[i])

  rounded = str(np.round(val, 2)) + '%'

  plt.title(class\_labels[test\_labels[i]] + ' (' +\

            class\_labels[y\_pred[i]] + ') ' + rounded )

plt.tight\_layout()

plt.show()

Listing 5-17. Sophisticated visualization of misclassifications

Misclassifications, if any, are in red. Above each article of clothing is the actual label, prediction in parentheses, and confidence in the prediction.

## Predict from a Single Image

We can make a prediction on a single image. We choose a number between 0 and 9,999 because we want an image from the test set. Alternatively, we can generate a random number between 0 and 9,999.

Let’s choose an image from the test set randomly:

beg, end = 0, len(test\_images) - 1

rng = np.random.default\_rng()

indx = int(rng.uniform(beg, end, size=1))

indx

Grab the image from the test set:

# Grab the image from the test dataset

img = test\_images[indx]

label = class\_labels[test\_labels[indx]]

img.shape, label

TensorFlow models are optimized to make predictions on a batch or collection of examples at once. So, we must add the single image to a batch where it is the only member.

Create a batch with a single image:

img\_batch = (np.expand\_dims(img, 0))

img\_batch.shape

The expand\_dims method inserts a new axis that appears at the axis position in the expanded array shape. So, the new shape is (1, 28, 28).

Now, we can predict:

pred\_single = model.predict(img\_batch)

Display prediction:

np.argmax(pred\_single)

We see the index position in the prediction array.

To make it easier, display prediction by it label name:

class\_labels[np.argmax(pred\_single)]

Display actual label:

class\_labels[test\_labels[indx]]

## Visualize Single Image Prediction

Visualize the prediction as shown in Listing 5-18:

pred = class\_labels[np.argmax(pred\_single)]

actual = class\_labels[test\_labels[indx]]

# get confidence from the predictions object

val = 100\*np.max(predictions[indx])

rounded = str(np.round(val, 2)) + '%'

# display actual image

plt.imshow(test\_images[indx], cmap=plt.cm.binary)

plt.show()

print ('actual:', actual)

print ('predicted:', pred)

print ('confidence', rounded)

Listing 5-18. Visualization of single image prediction

Get prediction by label name and actual label name for the image. Get prediction confidence from the predictions object that we created earlier. The predictions object holds all predictions based on the test set. The indx value provides the position of the confidence value in the predictions object.

## Confusion Matrix

Create a confusion matrix to visually demonstrate how well the model classified articles of clothing from Fashion-MNIST:

tf.math.confusion\_matrix(test\_labels, y\_pred)

To understand the TensorFlow confusion matrix, imagine that above the first row of the matrix are classes 0 to 9 from left to right. These represent predictions. Imagine that to the left of the first column of the matrix are classes 0 to 9 from top to bottom. These represent actual labels. Correct classifications are along the diagonal. Misclassifications are not on the diagonal.

The first set of correct classifications are in position row one, column one and represent class 0. The second set are in position row two, column two and represent class 1. And so on…

An example misclassification set is row one, column two. This represents the number of predictions for class 1 that were misclassified as class 0. For a technical explanation of the TensorFlow confusion matrix, peruse the following URL:

<https://stackoverflow.com/questions/46616837/understanding-a-tensorflow-confusion-matrix-for-binary-classification>

For a general explanation of a confusion matrix, peruse the following URL:

<https://www.dataschool.io/simple-guide-to-confusion-matrix-terminology/>

# Number of Hidden Layers

For many problems, we can begin with a single hidden layer and get reasonable results as we did with our Fashion-MNIST model in this chapter. For more complex problems, we should add layers until we start overfitting the training set.

# Number of Neurons in Hidden Layers

The number of neurons in the input and output layers is based on the type of input and output your task requires. For example, the Fashion-MNIST task requires 28 x 28 = 784 input neurons and 10 output neurons. For hidden layers, it is very difficult to determine. You can try increasing the number of neurons gradually until the network starts overfitting. In practice, we typically pick a model with more layers and neurons than we need and then use early stopping and other regularization techniques to reduce overfitting. Since this is an introduction, we won't delve any deeper into tuning networks.

Generally, we get a better model by increasing the number of layers rather than the number of neurons per layer. Of course, the number of layers and neurons we include is limited by our available computational resources.