Swarali Chine - 1222583687

Handwritten Image Detection with Keras using MNIST and Fashion MNIST data

In this exercise we will work with image data: specifically the famous MNIST and Fashion MNIST data sets. MNIST data set contains 70,000 images of handwritten digits in grayscale (0=black, 255 = white). Fashion MNIST data set contains 70,000 images of clothing in grayscale (0=black, 255 = white). All the images are 28 pixels by 28 pixels for a total of 784 pixels. This is quite small by image standards. Also, the images are well centered and isolated. This makes this problem solvable with standard fully connected neural nets without too much pre-work.

We will use a Convolutional Neural Network and compare it with a linear neural network.

In the first part of this notebook, we will walk you through loading in the data, building a network, and training it. Then it will be your turn to try different models.

```
# Preliminaries

from __future__ import print_function

import keras
import tensorflow as tf
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Conv2D, MaxPooling2D, Flatten
from tensorflow.keras.optimizers import RMSprop
from keras.datasets import fashion_mnist

import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

Let's explore the dataset a little bit

```
# Load the data, shuffled and split between train and test sets
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11493376/11490434 [==============================] - 0s 0us/step
11501568/11490434 [==============================] - 0s 0us/step
```

```
x_train.shape
```

```
(60000, 28, 28)
```

```
#Let's just look at a particular example to see what is inside
```

```
x_train[333]  ## Just a 28 x 28 numpy array of ints from 0 to 255
```

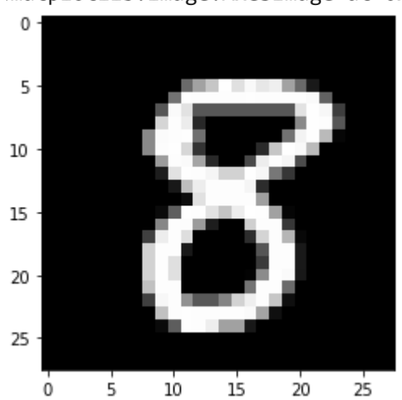
```
array([[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  87, 138,
        170, 253, 201, 244, 212, 222, 138,  86,  22,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  95, 253, 252,
        252, 252, 252, 253, 252, 252, 252, 252, 245,  80,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,  68, 246, 205,  69,
         69,  69,  69,  69,  69,  69,  69, 205, 253, 240,  50,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0, 187, 252, 218,  34,
          0,   0,   0,   0,   0,   0,   0, 116, 253, 252,  69,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0, 116, 248, 252, 253,  92,
          0,   0,   0,   0,   0,   0,  95, 230, 253, 157,   6,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0, 116, 249, 253, 189,  42,
          0,   0,   0,   0,  36, 170, 253, 243, 158,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0, 133, 252, 245, 140,
         34,   0,   0,  57, 219, 252, 235,  60,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,  25, 205, 253, 252,
        234, 184, 184, 253, 240, 100,  44,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  21, 161, 219,
        252, 252, 252, 234,  37,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  11, 203,
        252, 252, 252, 251, 135,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   9,  76, 255, 253,
        205, 168, 220, 255, 253, 137,   5,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0, 114, 252, 249, 132,
         25,   0,   0, 180, 252, 252,  45,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,  51, 220, 252, 199,   0,
          0,   0,   0,  38, 186, 252, 154,   7,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0, 184, 252, 252,  21,   0,
          0,   0,   0,   0,  67, 252, 252,  22,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0, 184, 252, 200,   0,   0,
```

```
# What is the corresponding label in the training set?
y_train[333]
```

```
8
```

```
# Let's see what this image actually looks like

plt.imshow(x_train[333], cmap='Greys_r')
```

```
<matplotlib.image.AxesImage at 0x7fac821dfc50>
```




```
# this is the shape of the np.array x_train
# it is 3 dimensional.
print(x_train.shape, 'train samples')
print(x_test.shape, 'test samples')
```

```
(60000, 28, 28) train samples
(10000, 28, 28) test samples
```

```
## For our purposes, these images are just a vector of 784 inputs, so let's convert
x_train = x_train.reshape(len(x_train), 28*28)
x_test = x_test.reshape(len(x_test), 28*28)

## Keras works with floats, so we must cast the numbers to floats
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')

## Normalize the inputs so they are between 0 and 1
x_train /= 255
x_test /= 255
```

```
# convert class vectors to binary class matrices
num_classes = 10
y_train = tf.keras.utils.to_categorical(y_train, num_classes)
y_test = tf.keras.utils.to_categorical(y_test, num_classes)
```

```
y_train[333]  # now the digit k is represented by a 1 in the kth entry (0-indexed) of the length 10 vector
```

```
array([0., 0., 0., 0., 0., 0., 0., 0., 1., 0.], dtype=float32)
```

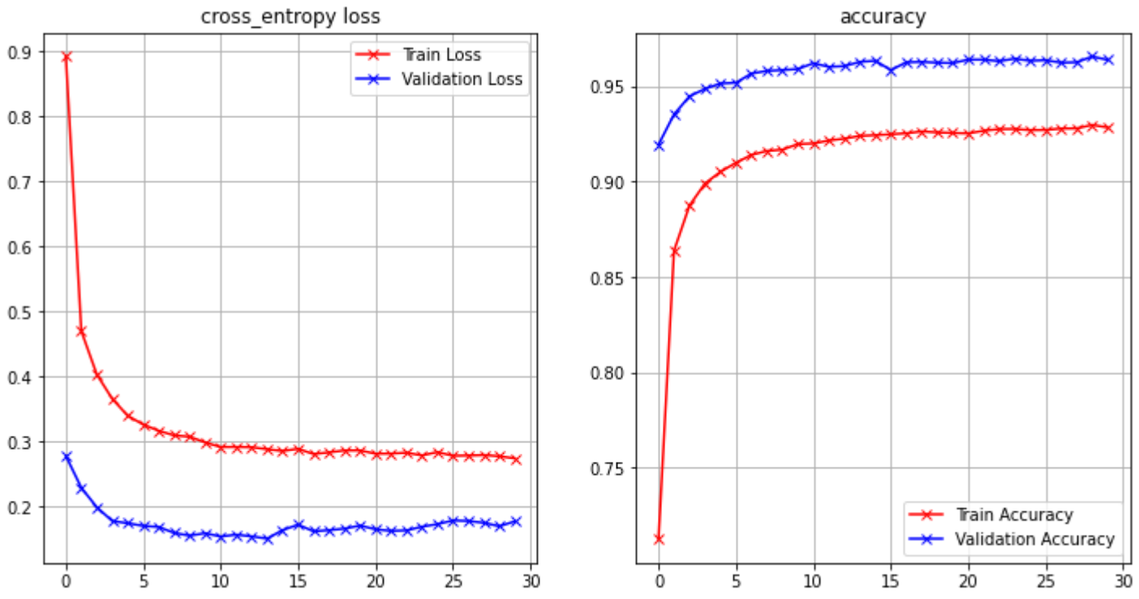
```
# We will build a model with two hidden layers of size 512
# Fully connected inputs at each layer
# We will use dropout of .5 to help regularize
model_1 = Sequential()
model_1.add(Dense(64, activation='relu', input_shape=(784,)))
model_1.add(Dropout(0.5))
model_1.add(Dense(64, activation='relu'))
model_1.add(Dropout(0.5))
model_1.add(Dense(10, activation='softmax'))


## Note that this model has a LOT of parameters
model_1.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 64)                50240
_________________________________________________________________
dropout (Dropout)            (None, 64)                0
_________________________________________________________________
dense_1 (Dense)              (None, 64)                4160
_________________________________________________________________
dropout_1 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_2 (Dense)              (None, 10)                650
=================================================================
Total params: 55,050
Trainable params: 55,050
Non-trainable params: 0
_________________________________________________________________
```

```
# Let's compile the model
learning_rate = .001
model_1.compile(loss='categorical_crossentropy',
              optimizer=RMSprop(learning_rate=learning_rate),
              metrics=['accuracy'])
# note that `categorical cross entropy` is the natural generalization
# of the loss function we had in binary classification case, to multi class case


# And now let's fit.

batch_size = 128  # mini-batch with 128 examples
epochs = 30
history = model_1.fit(
    x_train, y_train,
    batch_size=batch_size,
    epochs=epochs,
    verbose=1,
    validation_data=(x_test, y_test))
```

```
Epoch 1/30
469/469 [==============================] - 3s 4ms/step - loss: 0.8925 - accuracy: 0.7130 - val_loss: 0.2784 - val_accuracy: 0.9194
Epoch 2/30
469/469 [==============================] - 2s 4ms/step - loss: 0.4703 - accuracy: 0.8638 - val_loss: 0.2294 - val_accuracy: 0.9352
Epoch 3/30
469/469 [==============================] - 2s 3ms/step - loss: 0.4036 - accuracy: 0.8875 - val_loss: 0.1994 - val_accuracy: 0.9447
Epoch 4/30
469/469 [==============================] - 2s 3ms/step - loss: 0.3661 - accuracy: 0.8990 - val_loss: 0.1782 - val_accuracy: 0.9486
Epoch 5/30
469/469 [==============================] - 2s 4ms/step - loss: 0.3401 - accuracy: 0.9051 - val_loss: 0.1749 - val_accuracy: 0.9514
Epoch 6/30
469/469 [==============================] - 2s 4ms/step - loss: 0.3266 - accuracy: 0.9099 - val_loss: 0.1708 - val_accuracy: 0.9518
Epoch 7/30
469/469 [==============================] - 2s 4ms/step - loss: 0.3170 - accuracy: 0.9140 - val_loss: 0.1691 - val_accuracy: 0.9566
Epoch 8/30
469/469 [==============================] - 2s 4ms/step - loss: 0.3100 - accuracy: 0.9161 - val_loss: 0.1599 - val_accuracy: 0.9580
Epoch 9/30
469/469 [==============================] - 2s 4ms/step - loss: 0.3080 - accuracy: 0.9167 - val_loss: 0.1558 - val_accuracy: 0.9584
Epoch 10/30
469/469 [==============================] - 2s 4ms/step - loss: 0.2990 - accuracy: 0.9197 - val_loss: 0.1595 - val_accuracy: 0.9590
Epoch 11/30
469/469 [==============================] - 2s 4ms/step - loss: 0.2921 - accuracy: 0.9200 - val_loss: 0.1544 - val_accuracy: 0.9620
Epoch 12/30
469/469 [==============================] - 2s 4ms/step - loss: 0.2922 - accuracy: 0.9215 - val_loss: 0.1572 - val_accuracy: 0.9601
Epoch 13/30
469/469 [==============================] - 2s 4ms/step - loss: 0.2918 - accuracy: 0.9225 - val_loss: 0.1546 - val_accuracy: 0.9605
Epoch 14/30
469/469 [==============================] - 2s 4ms/step - loss: 0.2887 - accuracy: 0.9239 - val_loss: 0.1512 - val_accuracy: 0.9627
Epoch 15/30
469/469 [==============================] - 2s 3ms/step - loss: 0.2860 - accuracy: 0.9243 - val_loss: 0.1643 - val_accuracy: 0.9633
Epoch 16/30
469/469 [==============================] - 2s 3ms/step - loss: 0.2893 - accuracy: 0.9248 - val_loss: 0.1726 - val_accuracy: 0.9584
Epoch 17/30
469/469 [==============================] - 2s 4ms/step - loss: 0.2812 - accuracy: 0.9253 - val_loss: 0.1624 - val_accuracy: 0.9625
Epoch 18/30
469/469 [==============================] - 2s 4ms/step - loss: 0.2838 - accuracy: 0.9263 - val_loss: 0.1642 - val_accuracy: 0.9627
Epoch 19/30
469/469 [==============================] - 2s 4ms/step - loss: 0.2868 - accuracy: 0.9258 - val_loss: 0.1665 - val_accuracy: 0.9623
Epoch 20/30
469/469 [==============================] - 2s 4ms/step - loss: 0.2868 - accuracy: 0.9255 - val_loss: 0.1713 - val_accuracy: 0.9621
Epoch 21/30
469/469 [==============================] - 2s 4ms/step - loss: 0.2818 - accuracy: 0.9252 - val_loss: 0.1655 - val_accuracy: 0.9639
Epoch 22/30
469/469 [==============================] - 2s 4ms/step - loss: 0.2820 - accuracy: 0.9266 - val_loss: 0.1631 - val_accuracy: 0.9639
Epoch 23/30
469/469 [==============================] - 2s 3ms/step - loss: 0.2833 - accuracy: 0.9275 - val_loss: 0.1638 - val_accuracy: 0.9631
Epoch 24/30
469/469 [==============================] - 2s 3ms/step - loss: 0.2792 - accuracy: 0.9276 - val_loss: 0.1695 - val_accuracy: 0.9644
Epoch 25/30
469/469 [==============================] - 2s 3ms/step - loss: 0.2843 - accuracy: 0.9270 - val_loss: 0.1734 - val_accuracy: 0.9633
Epoch 26/30
469/469 [==============================] - 2s 4ms/step - loss: 0.2786 - accuracy: 0.9270 - val_loss: 0.1791 - val_accuracy: 0.9636
Epoch 27/30
469/469 [==============================] - 2s 4ms/step - loss: 0.2790 - accuracy: 0.9278 - val_loss: 0.1783 - val_accuracy: 0.9623
Epoch 28/30
469/469 [==============================] - 2s 4ms/step - loss: 0.2797 - accuracy: 0.9279 - val_loss: 0.1755 - val_accuracy: 0.9625
Epoch 29/30
469/469 [==============================] - 2s 4ms/step - loss: 0.2779 - accuracy: 0.9297 - val_loss: 0.1702 - val_accuracy: 0.9655
```

```
## We will use Keras evaluate function to evaluate performance on the test set

score = model_1.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])

     Test loss: 0.17780174314975739
     Test accuracy: 0.9638000130653381


def plot_loss_accuracy(history):
    fig = plt.figure(figsize=(12, 6))
    ax = fig.add_subplot(1, 2, 1)
    ax.plot(history.history["loss"],'r-x', label="Train Loss")
    ax.plot(history.history["val_loss"],'b-x', label="Validation Loss")
    ax.legend()
    ax.set_title('cross_entropy loss')
    ax.grid(True)

    ax = fig.add_subplot(1, 2, 2)
    ax.plot(history.history["accuracy"],'r-x', label="Train Accuracy")
    ax.plot(history.history["val_accuracy"],'b-x', label="Validation Accuracy")
    ax.legend()
    ax.set_title('accuracy')
    ax.grid(True)

plot_loss_accuracy(history)
```



This is reasonably good performance, but we can do even better! Next you will build an even bigger network and compare the performance.

Keras Layers for CNNs

- Previously we built Neural Networks using primarily the Dense, Activation and Dropout Layers.
- Here we will describe how to use some of the CNN-specific layers provided by Keras

Conv2D

```
keras.layers.convolutional.Conv2D(filters, kernel_size, strides=(1, 1), padding='valid', data_format=None, dilation_rate=(1, 1), activation=None,
```

A few parameters explained:

- `filters` : the number of filter used per location. In other words, the depth of the output.
- `kernel_size` : an (x,y) tuple giving the height and width of the kernel to be used
- `strides` : and (x,y) tuple giving the stride in each dimension. Default is `(1,1)`
- `input_shape` : required only for the first layer

Note, the size of the output will be determined by the kernel_size, strides

MaxPooling2D

```
keras.layers.pooling.MaxPooling2D(pool_size=(2, 2), strides=None, padding='valid', data_format=None)
```

- `pool_size` : the (x,y) size of the grid to be pooled.
- `strides` : Assumed to be the `pool_size` unless otherwise specified

Flatten

Turns its input into a one-dimensional vector (per instance). Usually used when transitioning between convolutional layers and fully connected layers.

Exercise

Build your own CNN model

Use the Keras "Sequential" functionality to build a convolutional neural network `model_2` with the following specifications:

Model Architecture:
We will build the famous LeNet-5 architecutre and measure its performance.
Convolution -> Relu -> Max pooling -> Convolution -> Relu -> Max pooling -> FC1 -> Relu -> FC2 -> Relu -> Output(SoftMax)

1. Convolution1 kernel size: 5(H) x 5(W) x 6(filters), stride = 1, no padding
2. Max pooling1 kernel size: 2(H) x 2(W), stride = 2
3. Convolution2 kernel size: 5(H) x 5(W) x 16(filters), stride = 1, no padding
4. Max pooling2 kernel size: 2(H) x 2(W), stride = 2
5. Fully Connected1 size: 120
6. Fully Connected2 size: 84
7. Train this model for 20 epochs with RMSProp at a learning rate of .001 and a batch size of 128

8. Plot the loss and accuracy graph for training the new model
9. Evaluate the model on test data

Hints:

- You can match the model summary to the LeNet-5 diagram in your slides to verify your implmentation.
  Slide: 07_Transfer_Learning -> Slide 27.
- You will not be graded on the accuracy of your model but it should have a decent accuracy of at least 70%. Performance below 70% means there is something wrong with the implementation.
- Check how to connect the output of Convolution layer with first Fully connected layer.
- Do not use any padding or dropout in LeNet model, it can be used for Project Bonus / Fashion MNIST.

To use the LeNet model, we need to do some preprocessing on the data first.

```
# Data is currently flattened i.e. m X 784, we need to reshape it back to 28 * 28. To do that we reshape the data.

x_train = np.reshape(x_train, [-1, 28, 28])
x_test = np.reshape(x_test, [-1, 28, 28])
x_train.shape, x_test.shape
```

```
# LeNet requires input of 32 X 32. So, we will pad the train and test images with zeros to increase the size to 32 X 32.

x_train=np.pad(x_train, ((0,0), (2,2), (2, 2)), 'constant')
x_test=np.pad(x_test, ((0,0), (2,2), (2, 2)), 'constant')
x_train.shape, x_test.shape
```

```
# Convolutional model requires input to be of 3 dimensions. We will add a channel dimension to it.

x_train = np.reshape(x_train, [-1, 32, 32, 1])
x_test = np.reshape(x_test, [-1, 32, 32, 1])
x_train.shape, x_test.shape
```

Write your code below

```
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import Flatten
from keras.layers.convolutional import Conv2D
from keras.layers.convolutional import MaxPooling2D
from keras.utils import np_utils
import matplotlib.pyplot as plt

from __future__ import print_function

import keras
import tensorflow as tf
```

```
from tensorflow.keras.optimizers import RMSprop
from keras.datasets import fashion_mnist

import numpy as np
%matplotlib inline


# normalize inputs from 0-255 to 0-1
x_train = x_train / 255
x_test = x_test / 255
# one hot encode outputs
y_train = np_utils.to_categorical(y_train)
y_test = np_utils.to_categorical(y_test)
num_classes = y_test.shape[1]


def baseline_model():
# create model
  model_2 = Sequential()

  model_2.add(Conv2D(filters=6, kernel_size=(5, 5), activation='relu', input_shape=(28,28,1)))
  model_2.add(MaxPooling2D(pool_size=(2, 2)))

  model_2.add(Conv2D(filters=16, kernel_size=(5, 5), activation='relu'))
  model_2.add(MaxPooling2D(pool_size=(2, 2)))

  model_2.add(Flatten())

  model_2.add(Dense(units=120, activation='relu'))

  model_2.add(Dense(units=84, activation='relu'))

  model_2.add(Dense(units=10, activation = 'softmax'))
#Compile model
  model_2.compile(loss='categorical_crossentropy', optimizer='adam',
  metrics=['accuracy'])
  return model_2


# build the model
model_2 = baseline_model()
# Fit the model
history_1 = model.fit(x_train, y_train, validation_data=(x_test, y_test),
                      epochs=20, batch_size=128, verbose=2)
# Final evaluation of the model
scores = model.evaluate(x_test, y_test, verbose=0)
print("CNN Error: %.2f%%" % (100-scores[1]*100))
```

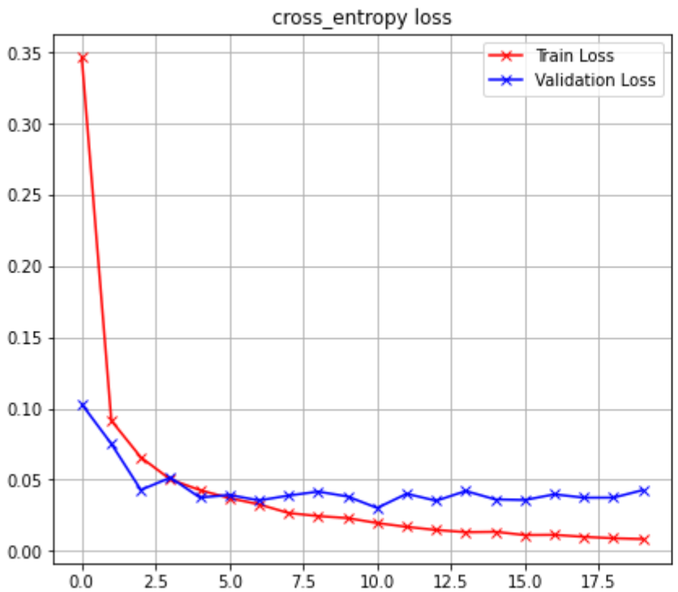
```
Epoch 1/20
469/469 - 23s - loss: 0.3462 - accuracy: 0.8984 - val_loss: 0.1030 - val_accuracy: 0.9684 - 23s/epoch - 48ms/step
Epoch 2/20
469/469 - 21s - loss: 0.0915 - accuracy: 0.9718 - val_loss: 0.0753 - val_accuracy: 0.9762 - 21s/epoch - 45ms/step
Epoch 3/20
469/469 - 21s - loss: 0.0657 - accuracy: 0.9790 - val_loss: 0.0429 - val_accuracy: 0.9854 - 21s/epoch - 45ms/step
Epoch 4/20
469/469 - 21s - loss: 0.0503 - accuracy: 0.9847 - val_loss: 0.0515 - val_accuracy: 0.9822 - 21s/epoch - 45ms/step
```

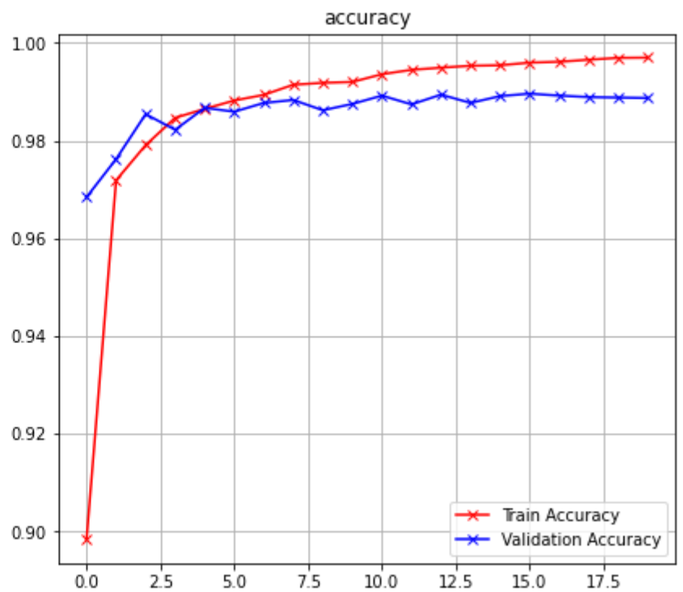
```
Epoch 5/20
469/469 - 21s - loss: 0.0428 - accuracy: 0.9865 - val_loss: 0.0375 - val_accuracy: 0.9867 - 21s/epoch - 45ms/step
Epoch 6/20
469/469 - 22s - loss: 0.0371 - accuracy: 0.9882 - val_loss: 0.0394 - val_accuracy: 0.9859 - 22s/epoch - 46ms/step
Epoch 7/20
469/469 - 21s - loss: 0.0328 - accuracy: 0.9894 - val_loss: 0.0355 - val_accuracy: 0.9877 - 21s/epoch - 46ms/step
Epoch 8/20
469/469 - 22s - loss: 0.0267 - accuracy: 0.9914 - val_loss: 0.0389 - val_accuracy: 0.9883 - 22s/epoch - 46ms/step
Epoch 9/20
469/469 - 21s - loss: 0.0245 - accuracy: 0.9918 - val_loss: 0.0417 - val_accuracy: 0.9862 - 21s/epoch - 45ms/step
Epoch 10/20
469/469 - 21s - loss: 0.0231 - accuracy: 0.9920 - val_loss: 0.0383 - val_accuracy: 0.9875 - 21s/epoch - 46ms/step
Epoch 11/20
469/469 - 21s - loss: 0.0197 - accuracy: 0.9936 - val_loss: 0.0302 - val_accuracy: 0.9892 - 21s/epoch - 46ms/step
Epoch 12/20
469/469 - 21s - loss: 0.0170 - accuracy: 0.9945 - val_loss: 0.0402 - val_accuracy: 0.9874 - 21s/epoch - 46ms/step
Epoch 13/20
469/469 - 22s - loss: 0.0148 - accuracy: 0.9949 - val_loss: 0.0353 - val_accuracy: 0.9894 - 22s/epoch - 46ms/step
Epoch 14/20
469/469 - 22s - loss: 0.0133 - accuracy: 0.9953 - val_loss: 0.0421 - val_accuracy: 0.9877 - 22s/epoch - 46ms/step
Epoch 15/20
469/469 - 21s - loss: 0.0135 - accuracy: 0.9954 - val_loss: 0.0362 - val_accuracy: 0.9891 - 21s/epoch - 46ms/step
Epoch 16/20
469/469 - 21s - loss: 0.0112 - accuracy: 0.9959 - val_loss: 0.0358 - val_accuracy: 0.9896 - 21s/epoch - 46ms/step
Epoch 17/20
469/469 - 22s - loss: 0.0114 - accuracy: 0.9961 - val_loss: 0.0398 - val_accuracy: 0.9892 - 22s/epoch - 46ms/step
Epoch 18/20
469/469 - 22s - loss: 0.0100 - accuracy: 0.9966 - val_loss: 0.0374 - val_accuracy: 0.9889 - 22s/epoch - 46ms/step
Epoch 19/20
469/469 - 22s - loss: 0.0090 - accuracy: 0.9969 - val_loss: 0.0375 - val_accuracy: 0.9888 - 22s/epoch - 46ms/step
Epoch 20/20
469/469 - 22s - loss: 0.0084 - accuracy: 0.9970 - val_loss: 0.0428 - val_accuracy: 0.9887 - 22s/epoch - 46ms/step
CNN Error: 1.13%
```

```
print("Accuracy:",scores[1]*100)
```

```
Accuracy: 98.86999726295471
```

```
def plot_loss_accuracy(history):
    fig = plt.figure(figsize=(15, 6))
    bx = fig.add_subplot(1, 2, 1)
    bx.plot(history.history["loss"],'r-x', label="Train Loss")
    bx.plot(history.history["val_loss"],'b-x', label="Validation Loss")
    bx.legend()
    bx.set_title('cross_entropy loss')
    bx.grid(True)

    bx = fig.add_subplot(1, 2, 2)
    bx.plot(history.history["accuracy"],'r-x', label="Train Accuracy")
    bx.plot(history.history["val_accuracy"],'b-x', label="Validation Accuracy")
    bx.legend()
    bx.set_title('accuracy')
    bx.grid(True)

plot_loss_accuracy(history_1)
```





Answer following questions

1) How do model_1 and model_2 compare? Which do you prefer? If you were going to put one into production, which would you choose and why?

2) Compare the trajectories of the loss function on the training set and test set for each model? How do they compare? What does that suggest about each model? Do the same for accuracy? Which do you think is more meaningful, the loss or the accuracy?

```
Answer 1: Model_2 is compared to Model_1 on the basis of 'Accuracy','Loss',.
 Model_2 is preferred over Model_1. The reason being the accuracy of Model_2 is
 greater than the accuracy of Model_1. The accuracy of model_2 is 98.86. The loss of
 model_2 is less than model_1.
```

Answer 2:The accuracy of model_1 is less than that of model_2. The loss of model_2 is less than model_1. However, model_2 is 'Overfit'. Since, the accuracy of model_2 is high than model_1 and the loss of model_2 is less than that of model_1, we prefer model_2 over model_1. Based on my observation, I think accuracy is more meaningful than loss.

Fashion MNIST

We will do the similar things for Fashion MNIST dataset. Fashion MNIST has 10 categories of clothing items:

Label	Description
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat

Label	Description
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

```
# Load the data, shuffled and split between train and test sets
(x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-labels-idx1-ubyte.gz
32768/29515 [================================] - 0s 0us/step
40960/29515 [========================================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-images-idx3-ubyte.gz
26427392/26421880 [==============================] - 0s 0us/step
26435584/26421880 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-labels-idx1-ubyte.gz
16384/5148 [===============================================================================================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-images-idx3-ubyte.gz
4423680/4422102 [==============================] - 0s 0us/step
4431872/4422102 [==============================] - 0s 0us/step
```

```
x_train[0].shape
```

```
(28, 28)
```

```
#Let's just look at a particular example to see what is inside

x_train[333]  ## Just a 28 x 28 numpy array of ints from 0 to 255
```

```
array([[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   3,   0,
         42,  92,  71, 107,  33,   0,   3,   0,   1,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   2,   0,  23,
        145,   0,   0,   0, 113,  20,   0,   3,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  89,
          8,   0,  10,   0,  36, 104,   0,   2,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   3,   0,  37,  97,
          0,   5,   1,   0,   0, 141,  23,   0,   2,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   5,   0,  70,  75,
          0,   6,   1,   0,   0, 141,  56,   0,   5,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   6,   0,  85,  57,
          0,   5,   1,   1,   0, 126,  62,   0,   6,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   6,   0,  85,  44,
          0,   3,   0,   4,   0, 109,  66,   0,   6,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   6,   0,  75,  41,
          0,   3,   0,   5,   0,  89,  68,   0,   6,   0,   0,   0,   0,
          0,   0],
```

```
[  0,   0,   0,   0,   0,   0,   0,   0,   0,   5,   0,  71,  44,
   0,   3,   0,   6,   0,  74,  69,   0,   6,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,   0,   4,   0,  80,  68,
   0,   4,   0,   5,   0,  89,  85,   0,   6,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,   0,   2,   3,   8,   0,  74,  89,
   0,  10,   3,  10,   0,  98,  84,   0,  10,   4,   3,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  30,  80,
   0,   0,   0,   0,   0,  69,  37,   0,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   2,   0,  80, 117, 118,  92, 125, 144,
 109, 141, 126, 132, 115, 151, 141, 130, 142, 150,  57,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,   0,  81, 162, 158, 153, 165, 192,
 182, 196, 187, 172, 183, 203, 206, 195, 188, 200, 140,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,   0, 114, 158, 169, 164, 164, 196,
 183, 177, 168, 163, 174, 195, 198, 195, 174, 170, 142,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,   0, 186, 166, 161, 180, 183, 214,
 181, 158, 163, 181, 186, 197, 186, 184, 193, 197,  86,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,   0, 162, 195, 185, 182, 186, 186,
 186, 193, 194, 194, 196, 195, 195, 192, 184, 215, 161,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,   0, 165, 233, 219, 230, 235, 230,
 228, 229, 229, 221, 218, 217, 215, 212, 204, 221, 160,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0, 134, 202, 221, 218, 220, 224, 222,
 220, 219, 221, 219, 213, 213, 208, 208, 199, 220, 142,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,  92, 213, 205, 223, 217, 226, 225, 226,
```

```
# What is the corresponding label in the training set?
y_train[333]
```

```
8
```

```
# this is the shape of the np.array x_train
# it is 3 dimensional.
print(x_train.shape, 'train samples')
print(x_test.shape, 'test samples')
```

```
(60000, 28, 28) train samples
(10000, 28, 28) test samples
```

```
## For our purposes, these images are just a vector of 784 inputs, so let's convert
x_train = x_train.reshape(len(x_train), 28*28)
x_test = x_test.reshape(len(x_test), 28*28)

## Keras works with floats, so we must cast the numbers to floats
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')

## Normalize the inputs so they are between 0 and 1
```

```
x_train /= 255
x_test /= 255

# convert class vectors to binary class matrices
num_classes = 10
y_train = tf.keras.utils.to_categorical(y_train, num_classes)
y_test = tf.keras.utils.to_categorical(y_test, num_classes)

y_train[333]
```

```
array([0., 0., 0., 0., 0., 0., 0., 0., 1., 0.], dtype=float32)
```

Reperforming the earlier preprocessing methods

```
# Data is currently flattened i.e. m X 784, we need to reshape it back to 28 * 28. To do that we reshape the data.

x_train = np.reshape(x_train, [-1, 28, 28])
x_test = np.reshape(x_test, [-1, 28, 28])
x_train.shape, x_test.shape
```

```
((60000, 28, 28), (10000, 28, 28))
```

```
# LeNet requires input of 32 X 32. So, we will pad the train and test images with zeros to increase the size to 32 X 32.

x_train=np.pad(x_train, ((0,0), (2,2), (2, 2)), 'constant')
x_test=np.pad(x_test, ((0,0), (2,2), (2, 2)), 'constant')
x_train.shape, x_test.shape
```

```
((60000, 32, 32), (10000, 32, 32))
```

```
# Convolutional model requires input to be of 3 dimensions. We will add a channel dimension to it.

x_train = np.reshape(x_train, [-1, 32, 32, 1])
x_test = np.reshape(x_test, [-1, 32, 32, 1])
x_train.shape, x_test.shape
```

```
((60000, 32, 32, 1), (10000, 32, 32, 1))
```

Build a similar convolutional model with a differnet structure, learning rate or number of epochs, etc. that you think will result in a good model for this dataset. Report the accuracy on test dataset.

```
#Download the fashion_mnist data

import tensorflow as tf

import numpy as np
import matplotlib.pyplot as plt

# Load the fashion-mnist pre-shuffled train data and test data
```

```
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.fashion_mnist.load_data()



#Visualize the data


# Print the number of training and test datasets
print(x_train.shape[0], 'train set')
print(x_test.shape[0], 'test set')

# Define the text labels
fashion_mnist_labels = ["T-shirt/top",  # index 0
                        "Trouser",      # index 1
                        "Pullover",     # index 2
                        "Dress",        # index 3
                        "Coat",         # index 4
                        "Sandal",       # index 5
                        "Shirt",        # index 6
                        "Sneaker",      # index 7
                        "Bag",          # index 8
                        "Ankle boot"]   # index 9

# Image index, you can pick any number between 0 and 59,999
img_index = 5
# y_train contains the lables, ranging from 0 to 9
label_index = y_train[img_index]
# Print the label, for example 2 Pullover
print ("y = " + str(label_index) + " " +(fashion_mnist_labels[label_index]))
# # Show one of the images from the training dataset
#plt.imshow(x_train[img_index],cmap='Greys_r')

     60000 train set
     10000 test set
     y = 2 Pullover


#Data Normalization
x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255


print("Number of train data - " + str(len(x_train)))
print("Number of test data - " + str(len(x_test)))

     Number of train data - 60000
     Number of test data - 10000


#Split the data into train/validation/test data sets

# Further break training data into train / validation sets
# put 5000 into validation set and keep remaining 55,000 for train)
(x_train, x_valid) = x_train[5000:], x_train[:5000]
(y_train, y_valid) = y_train[5000:], y_train[:5000]
```

```
w, h = 28, 28
x_train = x_train.reshape(x_train.shape[0], w, h, 1)
x_valid = x_valid.reshape(x_valid.shape[0], w, h, 1)
x_test = x_test.reshape(x_test.shape[0], w, h, 1)

# One-hot encode the labels
y_train = tf.keras.utils.to_categorical(y_train, 10)
y_valid = tf.keras.utils.to_categorical(y_valid, 10)
y_test = tf.keras.utils.to_categorical(y_test, 10)

# Print the number of training, validation, and test datasets
print(x_train.shape[0], 'train set')
print(x_valid.shape[0], 'validation set')
print(x_test.shape[0], 'test set')
```

```
55000 train set
5000 validation set
10000 test set
```

```
#Create the model architecture
model = tf.keras.Sequential()

# Must define the input shape in the first layer of the neural network
model.add(tf.keras.layers.Conv2D(filters=64, kernel_size=2,
                                 padding='same', activation='relu',
                                 input_shape=(28,28,1)))
model.add(tf.keras.layers.MaxPooling2D(pool_size=2))
model.add(tf.keras.layers.Dropout(0.3))

model.add(tf.keras.layers.Conv2D(filters=32, kernel_size=2,
                                 padding='same', activation='relu'))
model.add(tf.keras.layers.MaxPooling2D(pool_size=2))
model.add(tf.keras.layers.Dropout(0.3))

model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(256, activation='relu'))
model.add(tf.keras.layers.Dropout(0.5))
model.add(tf.keras.layers.Dense(10, activation='softmax'))

# Take a look at the model summary
model.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_4 (Conv2D)           (None, 28, 28, 64)        320

 max_pooling2d_4 (MaxPooling  (None, 14, 14, 64)       0
 2D)

 dropout (Dropout)           (None, 14, 14, 64)        0

 conv2d_5 (Conv2D)           (None, 14, 14, 32)        8224
```

```
 max_pooling2d_5 (MaxPooling  (None, 7, 7, 32)         0
 2D)

 dropout_1 (Dropout)         (None, 7, 7, 32)          0

 flatten_2 (Flatten)         (None, 1568)              0

 dense_6 (Dense)             (None, 256)               401664

 dropout_2 (Dropout)         (None, 256)               0

 dense_7 (Dense)             (None, 10)                2570

=================================================================
Total params: 412,778
Trainable params: 412,778
Non-trainable params: 0
_________________________________________________________________
```

```
#Complie the model
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])


#Train the model
from keras.callbacks import ModelCheckpoint

checkpointer = ModelCheckpoint(filepath='model.weights.best.hdf5',
                               verbose = 1, save_best_only=True)
model.fit(x_train,
          y_train,
          batch_size=128,
          epochs=20,
          validation_data=(x_valid, y_valid),
          callbacks=[checkpointer])
```

```
Epoch 1/20
430/430 [==============================] - ETA: 0s - loss: 0.6534 - accuracy: 0.7618
Epoch 00001: val_loss improved from inf to 0.40799, saving model to model.weights.best.hdf5
430/430 [==============================] - 71s 164ms/step - loss: 0.6534 - accuracy: 0.7618 - val_loss: 0.4080 - val_accuracy: 0.8492
Epoch 2/20
430/430 [==============================] - ETA: 0s - loss: 0.4467 - accuracy: 0.8398
Epoch 00002: val_loss improved from 0.40799 to 0.33959, saving model to model.weights.best.hdf5
430/430 [==============================] - 70s 164ms/step - loss: 0.4467 - accuracy: 0.8398 - val_loss: 0.3396 - val_accuracy: 0.8780
Epoch 3/20
430/430 [==============================] - ETA: 0s - loss: 0.3920 - accuracy: 0.8581
Epoch 00003: val_loss improved from 0.33959 to 0.31721, saving model to model.weights.best.hdf5
430/430 [==============================] - 70s 163ms/step - loss: 0.3920 - accuracy: 0.8581 - val_loss: 0.3172 - val_accuracy: 0.8890
Epoch 4/20
430/430 [==============================] - ETA: 0s - loss: 0.3663 - accuracy: 0.8662
Epoch 00004: val_loss improved from 0.31721 to 0.29608, saving model to model.weights.best.hdf5
430/430 [==============================] - 70s 163ms/step - loss: 0.3663 - accuracy: 0.8662 - val_loss: 0.2961 - val_accuracy: 0.8958
Epoch 5/20
430/430 [==============================] - ETA: 0s - loss: 0.3509 - accuracy: 0.8700
Epoch 00005: val_loss improved from 0.29608 to 0.28460, saving model to model.weights.best.hdf5
430/430 [==============================] - 70s 164ms/step - loss: 0.3509 - accuracy: 0.8700 - val_loss: 0.2846 - val_accuracy: 0.8990
```

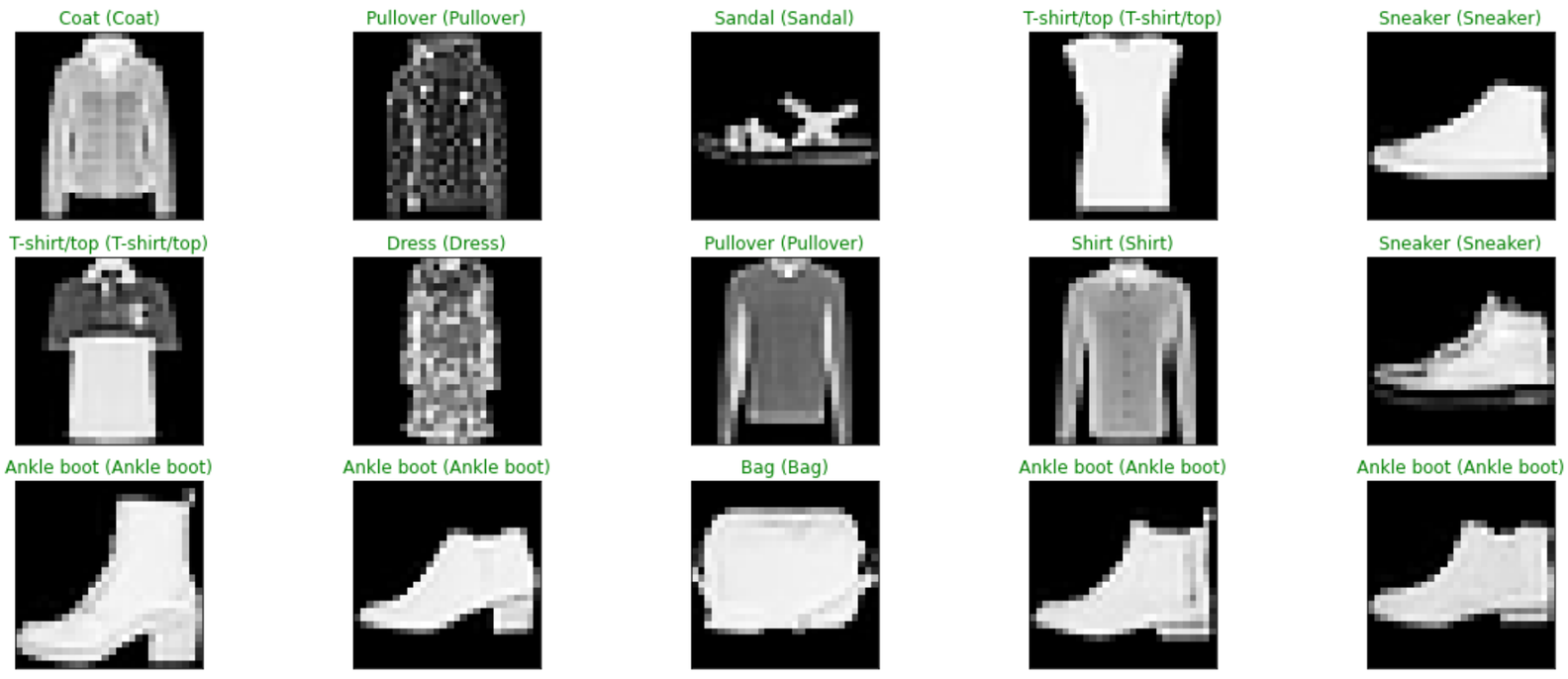
```
Epoch 6/20
430/430 [==============================] - ETA: 0s - loss: 0.3314 - accuracy: 0.8796
Epoch 00006: val_loss improved from 0.28460 to 0.26840, saving model to model.weights.best.hdf5
430/430 [==============================] - 70s 163ms/step - loss: 0.3314 - accuracy: 0.8796 - val_loss: 0.2684 - val_accuracy: 0.9002
Epoch 7/20
430/430 [==============================] - ETA: 0s - loss: 0.3186 - accuracy: 0.8833
Epoch 00007: val_loss improved from 0.26840 to 0.26084, saving model to model.weights.best.hdf5
430/430 [==============================] - 70s 163ms/step - loss: 0.3186 - accuracy: 0.8833 - val_loss: 0.2608 - val_accuracy: 0.9046
Epoch 8/20
430/430 [==============================] - ETA: 0s - loss: 0.3074 - accuracy: 0.8871
Epoch 00008: val_loss did not improve from 0.26084
430/430 [==============================] - 70s 163ms/step - loss: 0.3074 - accuracy: 0.8871 - val_loss: 0.2615 - val_accuracy: 0.9038
Epoch 9/20
430/430 [==============================] - ETA: 0s - loss: 0.2964 - accuracy: 0.8903
Epoch 00009: val_loss improved from 0.26084 to 0.24475, saving model to model.weights.best.hdf5
430/430 [==============================] - 70s 163ms/step - loss: 0.2964 - accuracy: 0.8903 - val_loss: 0.2448 - val_accuracy: 0.9104
Epoch 10/20
430/430 [==============================] - ETA: 0s - loss: 0.2919 - accuracy: 0.8915
Epoch 00010: val_loss did not improve from 0.24475
430/430 [==============================] - 70s 162ms/step - loss: 0.2919 - accuracy: 0.8915 - val_loss: 0.2462 - val_accuracy: 0.9074
Epoch 11/20
430/430 [==============================] - ETA: 0s - loss: 0.2814 - accuracy: 0.8961
Epoch 00011: val_loss improved from 0.24475 to 0.23628, saving model to model.weights.best.hdf5
430/430 [==============================] - 70s 162ms/step - loss: 0.2814 - accuracy: 0.8961 - val_loss: 0.2363 - val_accuracy: 0.9114
Epoch 12/20
430/430 [==============================] - ETA: 0s - loss: 0.2750 - accuracy: 0.8986
Epoch 00012: val_loss improved from 0.23628 to 0.23166, saving model to model.weights.best.hdf5
430/430 [==============================] - 70s 162ms/step - loss: 0.2750 - accuracy: 0.8986 - val_loss: 0.2317 - val_accuracy: 0.9134
Epoch 13/20
430/430 [==============================] - ETA: 0s - loss: 0.2686 - accuracy: 0.9012
Epoch 00013: val_loss improved from 0.23166 to 0.22904, saving model to model.weights.best.hdf5
430/430 [==============================] - 70s 163ms/step - loss: 0.2686 - accuracy: 0.9012 - val_loss: 0.2290 - val_accuracy: 0.9134
Epoch 14/20
430/430 [==============================] - ETA: 0s - loss: 0.2626 - accuracy: 0.9020
Epoch 00014: val_loss improved from 0.22904 to 0.22881, saving model to model.weights.best.hdf5
430/430 [==============================] - 70s 163ms/step - loss: 0.2626 - accuracy: 0.9020 - val_loss: 0.2288 - val_accuracy: 0.9164
Epoch 15/20
```

```
#Load Model with the best validation accuracy
model.load_weights('model.weights.best.hdf5')


#Test Accuracy

# Evaluate the model on test set
score = model.evaluate(x_test, y_test, verbose=0)

# Print test accuracy
print('\n', 'Test accuracy:', score[1])
print('\n' 'The accuracy to classify fashion-MNIST is:',score[1]*100 )
```

```
 Test accuracy: 0.9168000221252441

The accuracy to classify fashion-MNIST is: 91.68000221252441
```

```
y_hat = model.predict(x_test)
```

```
# Plot a random sample of 10 test images, their predicted labels and
# ground truth
figure = plt.figure(figsize=(20, 8))
for i, index in enumerate(np.random.choice(x_test.shape[0], size=15,
                                           replace=False)):
    ax = figure.add_subplot(3, 5, i + 1, xticks=[], yticks=[])
    # Display each image
    ax.imshow(np.squeeze(x_test[index]),cmap='Greys_r')
    predict_index = np.argmax(y_hat[index])
    true_index = np.argmax(y_test[index])
    # Set the title for each image
    ax.set_title("{} ({})".format(fashion_mnist_labels[predict_index],
                                  fashion_mnist_labels[true_index]),
                                  color=("green" if predict_index == true_index
                                         else "red"))
```



✓ 0s completed at 9:45 PM