Deep Learning Project: Street View Housing Number Digit Recognition

Marks: 60

Context

One of the most interesting tasks in deep learning is to recognize objects in natural scenes. The ability to process visual information using machine learning algorithms can be very useful as demonstrated in various applications.

The SVHN dataset contains over 600,000 labeled digits cropped from street-level photos. It is one of the most popular image recognition datasets. It has been used in neural networks created by Google to improve the map quality by automatically transcribing the address numbers from a patch of pixels. The transcribed number with a known street address helps pinpoint the location of the building it represents.

Objective

Our objective is to predict the number depicted inside the image by using Artificial or Fully Connected Feed Forward Neural Networks and Convolutional Neural Networks. We will go through various models of each and finally select the one that is giving us the best performance.

Dataset

Here, we will use a subset of the original data to save some computation time. The dataset is provided as a .h5 file. The basic preprocessing steps have been applied on the dataset.

Mount the drive

Let us start by mounting the Google drive. You can run the below cell to mount the Google drive.

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, ca
```

Importing the necessary libraries

```
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import MinMaxScaler

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Dropout, Activation, BatchNormalization

from tensorflow.keras.losses import categorical_crossentropy

from tensorflow.keras.optimizers import Adam

from tensorflow.keras.utils import to_categorical
```

Let us check the version of tensorflow.

```
print(tf.__version__)
```

```
2.12.0
```

Load the dataset

- Let us now load the dataset that is available as a .h5 file.
- Split the data into the train and the test dataset.

```
import h5py
h5f = h5py.File('/content/drive/MyDrive/SVHN_single_grey1.h5', 'r')

X_train = h5f['X_train'][:]
y_train = h5f['y_train'][:]
X_test = h5f['X_test'][:]
y_test = h5f['y_test'][:]

h5f.close()
```

Check the number of images in the training and the testing dataset.

```
len(X_train), len(X_test),X_train.shape, X_test.shape
```

```
(42000, 18000, (42000, 32, 32), (18000, 32, 32))
```

Observation: 42K images in train set and 18k in test set.

Visualizing images

- Use X_train to visualize the first 10 images.
- Use Y_train to print the first 10 labels.

```
plt.figure(figsize=(10, 1))

for i in range(10):
    plt.subplot(1, 10, i+1)
    plt.imshow(X_train[i], cmap="gray")
    plt.axis('off')

plt.show()
print('label for each of the above image: %s' % (y_train[0:10]))
```

```
label for each of the above image: [2 6 7 4 4 0 3 0 7 3]
```

Data preparation

- Print the shape and the array of pixels for the first image in the training dataset.
- Normalize the train and the test dataset by dividing by 255.
- Print the new shapes of the train and the test dataset.
- One-hot encode the target variable.

```
print("Shape:", X_train[0].shape)
print()
print("First image:\n", X_train[0])
```

```
Shape: (32, 32)

First image:
 [[ 33.0704  30.2601  26.852  ...  71.4471  58.2204  42.9939]
 [ 25.2283  25.5533  29.9765 ... 113.0209 103.3639  84.2949]
 [ 26.2775  22.6137  40.4763 ... 113.3028 121.775  115.4228]
 ...
 [ 28.5502  36.212   45.0801 ...  24.1359  25.0927  26.0603]
 [ 38.4352  26.4733  23.2717 ...  28.1094  29.4683  30.0661]
 [ 50.2984  26.0773  24.0389 ...  49.6682  50.853   53.0377]]
```

```
X_train = X_train.reshape(X_train.shape[0], 1024)
X_test = X_test.reshape(X_test.shape[0], 1024)
```

Normalize the train and the test data

```
X_train = X_train/255
X_test = X_test/255
```

Print the shapes of Training and Test data

```
print('Training set:', X_train.shape, y_train.shape)
print('Test set:', X_test.shape, y_test.shape)
```

```
Training set: (42000, 1024) (42000,)
Test set: (18000, 1024) (18000,)
```

One-hot encode output

```
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
```

```
print(X_train.shape)
print(y_train.shape)
```

```
(42000, 1024)
(42000, 10)
```

Observation: The images were converted from 2 dimensional (32x32) to 1D array (1024).

Model Building

Now that we have done the data preprocessing, let's build an ANN model.

Fix the seed for random number generators

```
np.random.seed(42)
import random
random.seed(42)
tf.random.set_seed(42)
```

Model Architecture

- Write a function that returns a sequential model with the following architecture:
  - First hidden layer with **64 nodes and the relu activation** and the **input shape = (1024,)**
  - Second hidden layer with **32 nodes and the relu activation**
  - Output layer with **activation as 'softmax' and number of nodes equal to the number of classes, i.e., 10**
  - Compile the model with the **loss equal to categorical_crossentropy, optimizer equal to Adam(learning_rate = 0.001), and metric equal to 'accuracy'**. Do not fit the model here, just return the compiled model.
- Call the nn_model_1 function and store the model in a new variable.
- Print the summary of the model.
- Fit on the train data with a **validation split of 0.2, batch size = 128, verbose = 1, and epochs = 20**. Store the model building history to use later for visualization.

Build and train an ANN model as per the above mentioned architecture.

```
from tensorflow.keras import losses
from tensorflow.keras import optimizers


def nn_model_1():
    model = tf.keras.Sequential([
    tf.keras.layers.Dense(64, activation='relu', input_shape=(1024, )),
    tf.keras.layers.Dense(32, activation='relu'),
    tf.keras.layers.Dense(10, activation = 'softmax')
    ])


    adam = optimizers.Adam(learning_rate=0.001)


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

```
model_1 = nn_model_1()
```

```
model_1.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 64)                65600

 dense_1 (Dense)             (None, 32)                2080

 dense_2 (Dense)             (None, 10)                330

=================================================================
Total params: 68,010
Trainable params: 68,010
Non-trainable params: 0
_________________________________________________________________
```

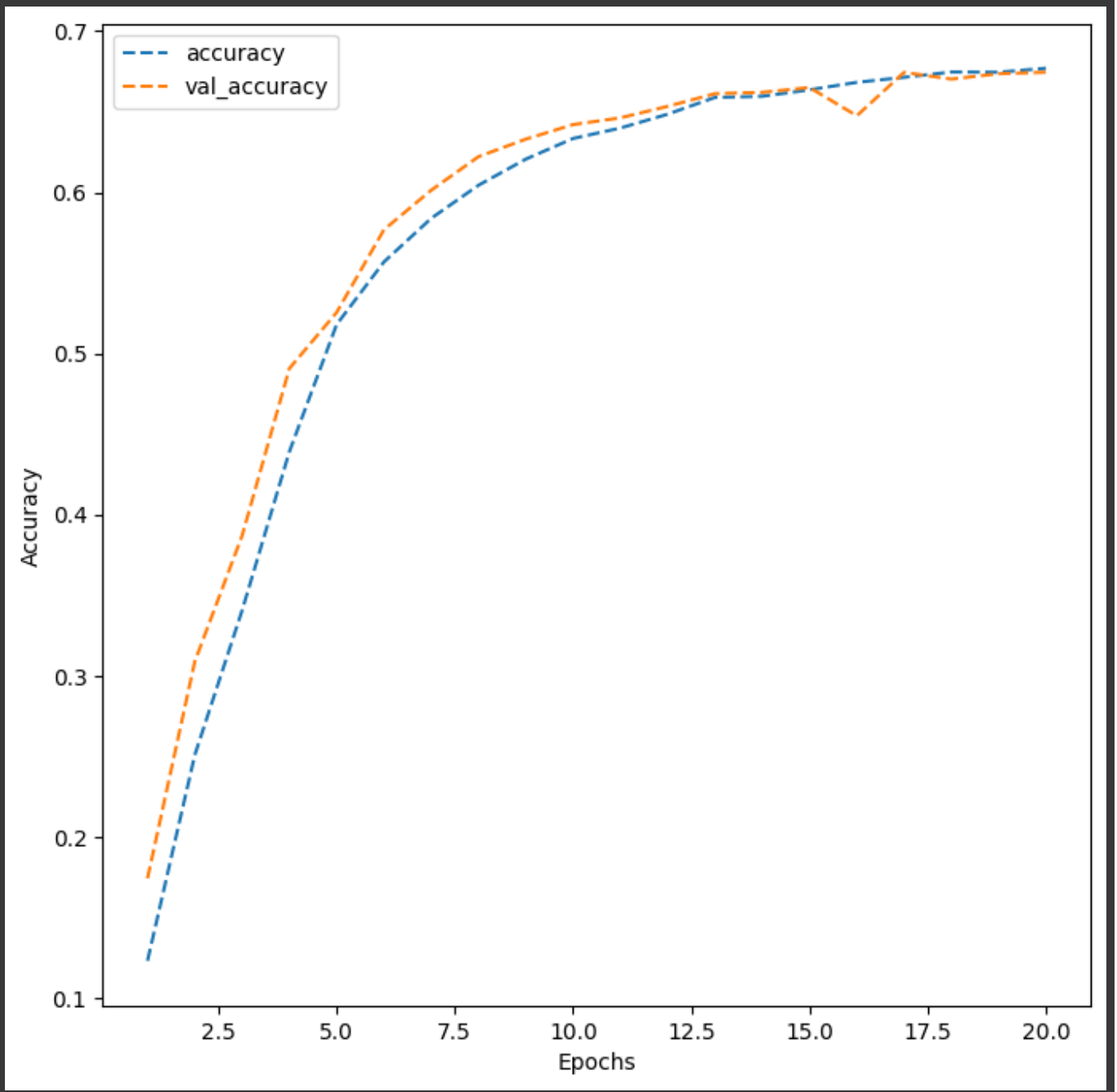
```
history_model_1 = model_1.fit(X_train, y_train, epochs=20, validation_split=0.2, batch_size=12
```

```
Epoch 1/20
263/263 [==============================] - 4s 9ms/step - loss: 2.2934 - acc
Epoch 2/20
263/263 [==============================] - 2s 6ms/step - loss: 2.0774 - acc
Epoch 3/20
263/263 [==============================] - 1s 5ms/step - loss: 1.8679 - acc
Epoch 4/20
263/263 [==============================] - 1s 5ms/step - loss: 1.6573 - acc
Epoch 5/20
263/263 [==============================] - 1s 5ms/step - loss: 1.4635 - acc
Epoch 6/20
263/263 [==============================] - 1s 5ms/step - loss: 1.3588 - acc
Epoch 7/20
263/263 [==============================] - 1s 5ms/step - loss: 1.2895 - acc
Epoch 8/20
263/263 [==============================] - 1s 5ms/step - loss: 1.2375 - acc
Epoch 9/20
263/263 [==============================] - 1s 5ms/step - loss: 1.1990 - acc
Epoch 10/20
263/263 [==============================] - 2s 7ms/step - loss: 1.1666 - acc
Epoch 11/20
263/263 [==============================] - 2s 8ms/step - loss: 1.1465 - acc
Epoch 12/20
263/263 [==============================] - 2s 8ms/step - loss: 1.1239 - acc
Epoch 13/20
263/263 [==============================] - 2s 6ms/step - loss: 1.1017 - acc
Epoch 14/20
263/263 [==============================] - 1s 5ms/step - loss: 1.0981 - acc
Epoch 15/20
263/263 [==============================] - 1s 5ms/step - loss: 1.0854 - acc
Epoch 16/20
263/263 [==============================] - 1s 5ms/step - loss: 1.0715 - acc
Epoch 17/20
263/263 [==============================] - 1s 5ms/step - loss: 1.0626 - acc
Epoch 18/20
263/263 [==============================] - 1s 5ms/step - loss: 1.0535 - acc
Epoch 19/20
263/263 [==============================] - 1s 5ms/step - loss: 1.0486 - acc
Epoch 20/20
263/263 [==============================] - 1s 5ms/step - loss: 1.0420 - acc
```

Plot the Training and Validation Accuracies and write down your Observations.

```
dict_hist = history_model_1.history
list_ep = [i for i in range(1,21)]

plt.figure(figsize = (8,8))
plt.plot(list_ep,dict_hist['accuracy'],ls = '--', label = 'accuracy')
plt.plot(list_ep,dict_hist['val_accuracy'],ls = '--', label = 'val_accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epochs')
plt.legend()
plt.show()
```



Observations:

- Epoch 17 is good a fit because the accuracy starts to plateau after it.
- Validatition set has better accuracy than the train set. We can make amends to train set in order to improve the accuracy.
- Both validation and train have accuracy around 0.7 and the model is not overfit.

Let's build one more model with higher complexity and see if we can improve the performance of the model.

First, we need to clear the previous model's history from the Keras backend. Also, let's fix the seed again after clearing the backend.

```
from tensorflow.keras import backend
backend.clear_session()
```

```
np.random.seed(42)
import random
random.seed(42)
tf.random.set_seed(42)
```

Second Model Architecture

- Write a function that returns a sequential model with the following architecture:
    - First hidden layer with **256 nodes and the relu activation** and the **input shape = (1024,)**
    - Second hidden layer with **128 nodes and the relu activation**
    - Add the **Dropout layer with the rate equal to 0.2**
    - Third hidden layer with **64 nodes and the relu activation**
    - Fourth hidden layer with **64 nodes and the relu activation**
    - Fifth hidden layer with **32 nodes and the relu activation**
    - Add the **BatchNormalization layer**
    - Output layer with **activation as 'softmax' and number of nodes equal to the number of classes, i.e., 10**
    - Compile the model with the **loss equal to categorical_crossentropy, optimizer equal to Adam(learning_rate = 0.0005), and metric equal to 'accuracy'**. Do not fit the model here, just return the compiled model.
- Call the nn_model_2 function and store the model in a new variable.
- Print the summary of the model.
- Fit on the train data with a **validation split of 0.2, batch size = 128, verbose = 1, and epochs = 30**. Store the model building history to use later for visualization.

Build and train the new ANN model as per the above mentioned architecture

```
from tensorflow.keras import losses
from tensorflow.keras import optimizers


def nn_model_2():
    model = tf.keras.Sequential([
      tf.keras.layers.Dense(256, activation='relu', input_shape=(1024, )),
      tf.keras.layers.Dense(128, activation='relu'),
      tf.keras.layers.Dropout(rate = 0.2),
      tf.keras.layers.Dense(64, activation='relu'),
      tf.keras.layers.Dense(64, activation='relu'),
      tf.keras.layers.Dense(32, activation='relu'),
      tf.keras.layers.BatchNormalization(),
      tf.keras.layers.Dense(10, activation = 'softmax')
    ])

    adam = optimizers.Adam(learning_rate=0.0005)

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

    return model
```

```
model_2 = nn_model_2()
```

```
model_2.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 256)               262400

 dense_1 (Dense)             (None, 128)               32896

 dropout (Dropout)           (None, 128)               0

 dense_2 (Dense)             (None, 64)                8256

 dense_3 (Dense)             (None, 64)                4160

 dense_4 (Dense)             (None, 32)                2080

 batch_normalization (BatchN  (None, 32)               128
 ormalization)

 dense_5 (Dense)             (None, 10)                330

=================================================================
Total params: 310,250
Trainable params: 310,186
Non-trainable params: 64
_________________________________________________________________
```

```
history_model_2 = model_2.fit(X_train,y_train, epochs=30, validation_split=0.2, batch_size=128
```

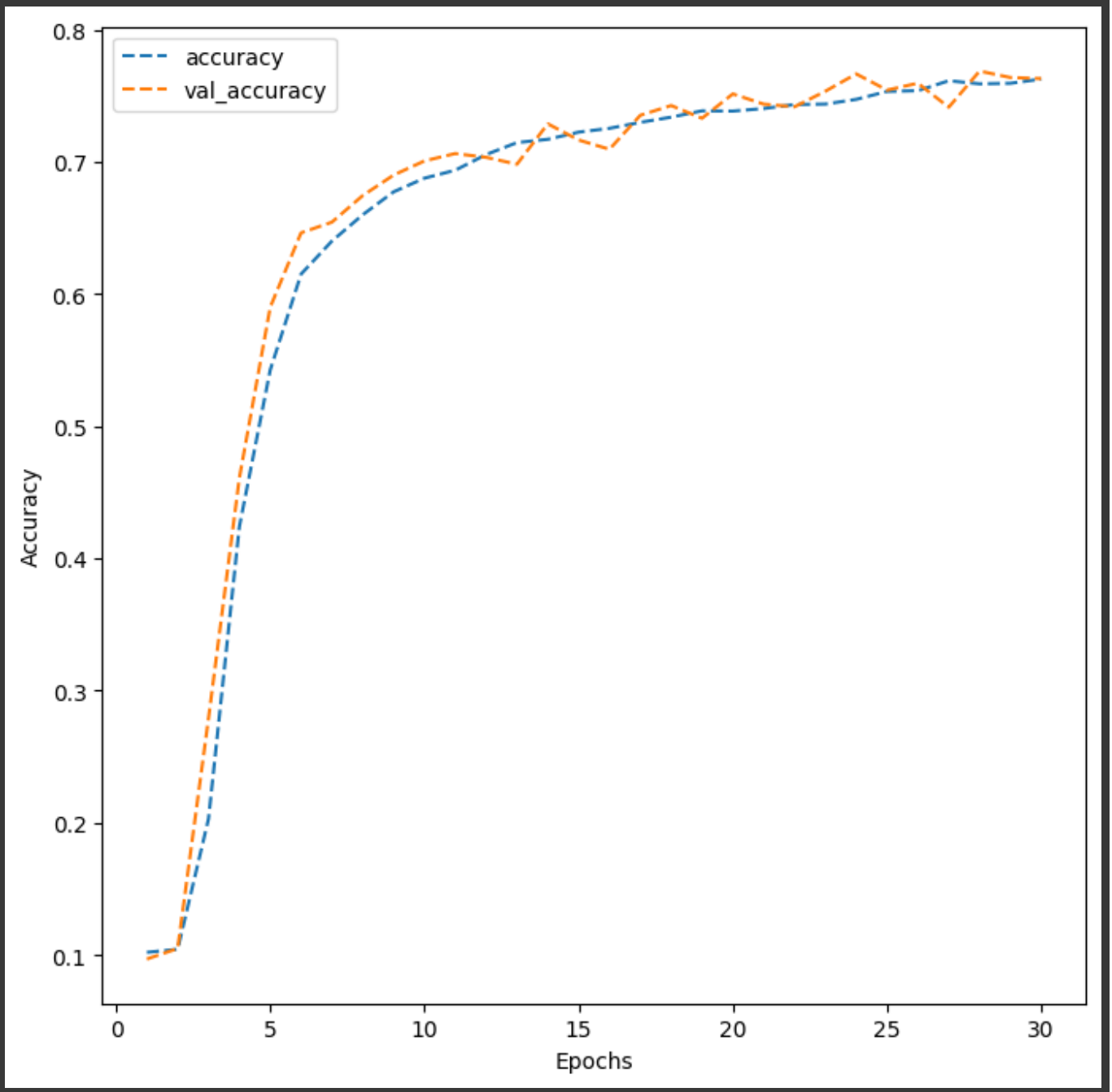
```
Epoch 1/30
263/263 [==============================] - 6s 12ms/step - loss: 2.3301 - ac
Epoch 2/30
263/263 [==============================] - 4s 14ms/step - loss: 2.3040 - ac
Epoch 3/30
263/263 [==============================] - 5s 17ms/step - loss: 2.1035 - ac
Epoch 4/30
263/263 [==============================] - 3s 12ms/step - loss: 1.6221 - ac
Epoch 5/30
263/263 [==============================] - 3s 11ms/step - loss: 1.3656 - ac
Epoch 6/30
263/263 [==============================] - 3s 12ms/step - loss: 1.1958 - ac
Epoch 7/30
263/263 [==============================] - 4s 17ms/step - loss: 1.1250 - ac
Epoch 8/30
263/263 [==============================] - 4s 16ms/step - loss: 1.0609 - ac
Epoch 9/30
263/263 [==============================] - 3s 11ms/step - loss: 1.0168 - ac
Epoch 10/30
263/263 [==============================] - 3s 11ms/step - loss: 0.9841 - ac
Epoch 11/30
263/263 [==============================] - 3s 11ms/step - loss: 0.9636 - ac
Epoch 12/30
263/263 [==============================] - 5s 19ms/step - loss: 0.9301 - ac
Epoch 13/30
263/263 [==============================] - 4s 14ms/step - loss: 0.9034 - ac
Epoch 14/30
263/263 [==============================] - 3s 12ms/step - loss: 0.8967 - ac
Epoch 15/30
263/263 [==============================] - 3s 12ms/step - loss: 0.8766 - ac
Epoch 16/30
263/263 [==============================] - 4s 15ms/step - loss: 0.8658 - ac
Epoch 17/30
263/263 [==============================] - 5s 18ms/step - loss: 0.8518 - ac
Epoch 18/30
263/263 [==============================] - 3s 12ms/step - loss: 0.8406 - ac
Epoch 19/30
263/263 [==============================] - 3s 12ms/step - loss: 0.8258 - ac
Epoch 20/30
263/263 [==============================] - 3s 11ms/step - loss: 0.8200 - ac
Epoch 21/30
263/263 [==============================] - 5s 17ms/step - loss: 0.8161 - ac
Epoch 22/30
263/263 [==============================] - 4s 16ms/step - loss: 0.8058 - ac
Epoch 23/30
263/263 [==============================] - 3s 12ms/step - loss: 0.8007 - ac
Epoch 24/30
263/263 [==============================] - 3s 11ms/step - loss: 0.7920 - ac
Epoch 25/30
263/263 [==============================] - 3s 12ms/step - loss: 0.7714 - ac
Epoch 26/30
263/263 [==============================] - 5s 19ms/step - loss: 0.7757 - ac
Epoch 27/30
263/263 [==============================] - 4s 14ms/step - loss: 0.7541 - ac
Epoch 28/30
263/263 [==============================] - 3s 11ms/step - loss: 0.7613 - ac
```

```
Epoch 29/30
263/263 [==============================] - 4s 14ms/step - loss: 0.7594 - ac
Epoch 30/30
```

Plot the Training and Validation Accuracies and write down your Observations.

```
dict_hist = history_model_2.history
list_ep = [i for i in range(1,31)]

plt.figure(figsize = (8,8))
plt.plot(list_ep,dict_hist['accuracy'],ls = '--', label = 'accuracy')
plt.plot(list_ep,dict_hist['val_accuracy'],ls = '--', label = 'val_accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epochs')
plt.legend()
plt.show()
```



Observations: This model has better accuracy than the previous. However, the validation might slightly be overfitted in this model.

Predictions on the test data

- Make predictions on the test set using the second model.
- Print the obtained results using the classification report and the confusion matrix.
- Final observations on the obtained results.

```
test_pred = model_2.predict(X_test)

test_pred = np.argmax(test_pred, axis=-1)
```

```
563/563 [==============================] - 3s 4ms/step
```

Note: Earlier, we noticed that each entry of the target variable is a one-hot encoded vector but to print the classification report and confusion matrix, we must convert each entry of y_test to a single label.

```
y_test = np.argmax(y_test, axis=-1)
```

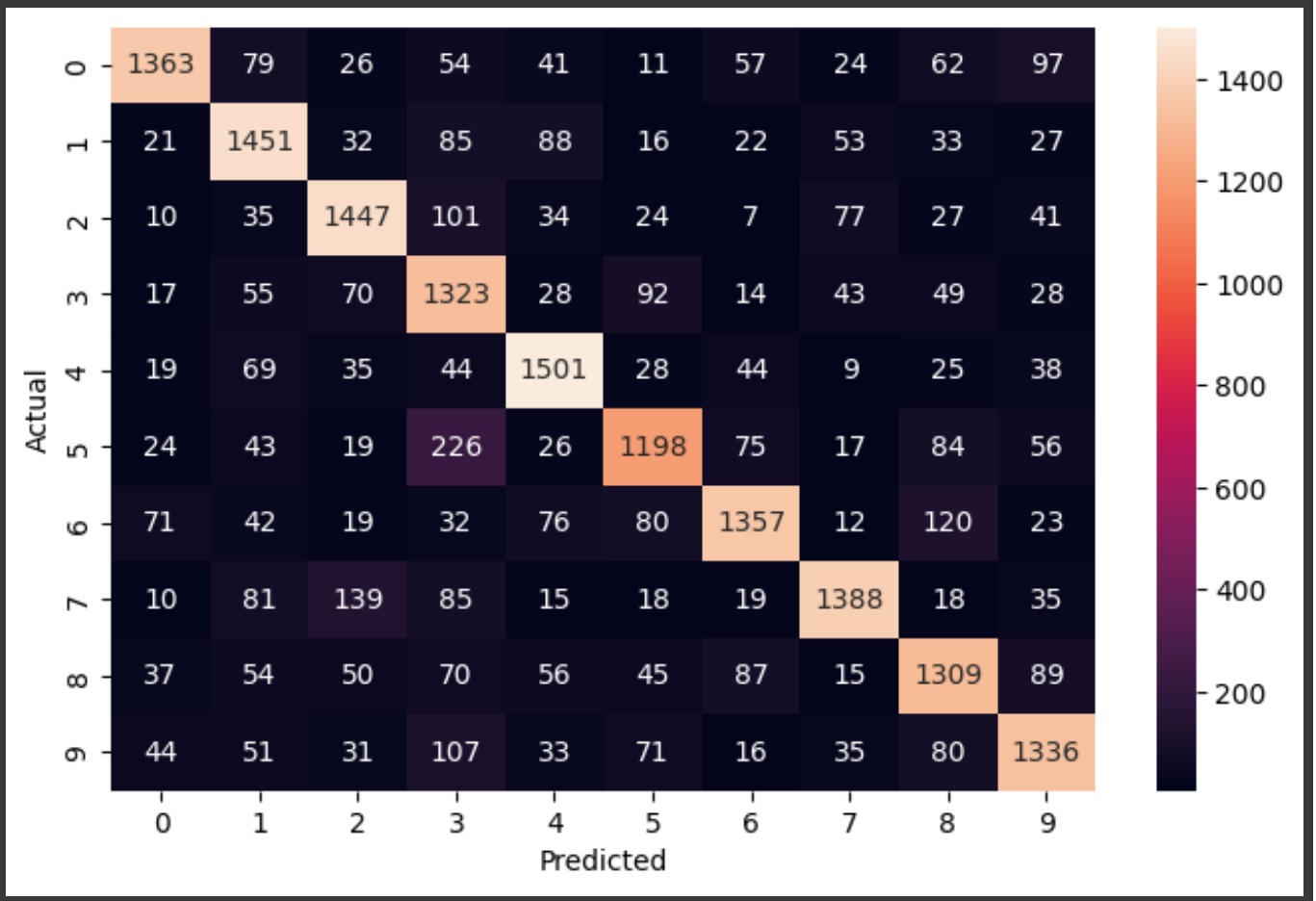
Print the classification report and the confusion matrix for the test predictions. Write your observations on the final results.

```
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix

print(classification_report(y_test, test_pred))


cm = confusion_matrix(y_test, test_pred)
plt.figure(figsize=(8,5))
sns.heatmap(cm, annot=True,  fmt='.0f')
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.show()
```

	precision	recall	f1-score	support
0	0.84	0.75	0.79	1814
1	0.74	0.79	0.77	1828
2	0.77	0.80	0.79	1803
3	0.62	0.77	0.69	1719
4	0.79	0.83	0.81	1812
5	0.76	0.68	0.72	1768
6	0.80	0.74	0.77	1832
7	0.83	0.77	0.80	1808
8	0.72	0.72	0.72	1812
9	0.75	0.74	0.75	1804
accuracy			0.76	18000
macro avg	0.76	0.76	0.76	18000
weighted avg	0.76	0.76	0.76	18000



Final Observations:

- Numbers 0 to 4 have the highest chance of recognition accuracy for being recognized based on the high f-1 score.
- Number 4 has the highest recall and 5 and 6 has the lowest recall. Thus, 5 and 6 are struggling to be recognized.
- Based on confusion matrix, 5-3 and 6-8 have issues being recognized together.

Using Convolutional Neural Networks

Load the dataset again and split the data into the train and the test dataset.

```
from google.colab import drive
drive.mount('/content/drive', force_remount=True)
```

```
Mounted at /content/drive
```

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler

import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, MaxPool2D, BatchNormalization, Dropout, Fla
from tensorflow.keras.utils import to_categorical
```

```
import h5py

h5f = h5py.File('/content/drive/MyDrive/SVHN_single_grey1.h5', 'r')

X_train = h5f['X_train'][:]
y_train = h5f['y_train'][:]
X_test = h5f['X_test'][:]
y_test = h5f['y_test'][:]


h5f.close()
```

Check the number of images in the training and the testing dataset.

```
len(X_train), len(X_test)
```

```
(42000, 18000)
```

Observation:

Data preparation

- Print the shape and the array of pixels for the first image in the training dataset.
- Reshape the train and the test dataset because we always have to give a 4D array as input to CNNs.
- Normalize the train and the test dataset by dividing by 255.
- Print the new shapes of the train and the test dataset.
- One-hot encode the target variable.

```
print("Shape:", X_train[0].shape)
print()
print("First image:\n", X_train[0])
```

```
Shape: (32, 32)

First image:
 [[ 33.0704  30.2601  26.852  ...  71.4471  58.2204  42.9939]
 [ 25.2283  25.5533  29.9765 ... 113.0209 103.3639  84.2949]
 [ 26.2775  22.6137  40.4763 ... 113.3028 121.775  115.4228]
 ...
 [ 28.5502  36.212   45.0801 ...  24.1359  25.0927  26.0603]
 [ 38.4352  26.4733  23.2717 ...  28.1094  29.4683  30.0661]
 [ 50.2984  26.0773  24.0389 ...  49.6682  50.853   53.0377]]
```

Reshape the dataset to be able to pass them to CNNs. Remember that we always have to give a 4D array as input to CNNs

```
X_train = X_train.reshape(X_train.shape[0], 32,32,1)
X_test = X_test.reshape(X_test.shape[0], 32,32,1)
```

Normalize inputs from 0-255 to 0-1

```
X_train = X_train / 255.0
X_test = X_test / 255.0
```

Print New shape of Training and Test

```
print('Training set:', X_train.shape, y_train.shape)
print('Test set:', X_test.shape, y_test.shape)
```

```
Training set: (42000, 32, 32, 1) (42000,)
Test set: (18000, 32, 32, 1) (18000,)
```

One-hot encode the labels in the target variable y_train and y_test.

```
y_train_encoded = tf.keras.utils.to_categorical(y_train)
y_test_encoded = tf.keras.utils.to_categorical(y_test)


y_test
```

```
array([1, 7, 2, ..., 7, 9, 2], dtype=uint8)
```

Observation:

Model Building

Now that we have done data preprocessing, let's build a CNN model. Fix the seed for random number generators

```
np.random.seed(42)
import random
random.seed(42)
tf.random.set_seed(42)
```

Model Architecture

- **Write a function** that returns a sequential model with the following architecture:
    - First Convolutional layer with **16 filters and the kernel size of 3x3**. Use the **'same' padding** and provide the **input shape = (32, 32, 1)**
    - Add a **LeakyRelu layer** with the **slope equal to 0.1**
    - Second Convolutional layer with **32 filters and the kernel size of 3x3 with 'same' padding**
    - Another **LeakyRelu** with the **slope equal to 0.1**
    - A **max-pooling layer** with a **pool size of 2x2**
    - **Flatten** the output from the previous layer
    - Add a **dense layer with 32 nodes**
    - Add a **LeakyRelu layer with the slope equal to 0.1**
    - Add the final **output layer with nodes equal to the number of classes, i.e., 10** and **'softmax' as the activation function**
    - Compile the model with the **loss equal to categorical_crossentropy, optimizer equal to Adam(learning_rate = 0.001), and metric equal to 'accuracy'**. Do not fit the model here, just return the compiled model.
- Call the function cnn_model_1 and store the output in a new variable.
- Print the summary of the model.
- Fit the model on the training data with a **validation split of 0.2, batch size = 32, verbose = 1, and epochs = 20**. Store the model building history to use later for visualization.

Build and train a CNN model as per the above mentioned architecture.

```
from tensorflow.keras import losses
from tensorflow.keras import optimizers


def cnn_model_1():
    model = Sequential()

    model.add(Conv2D(filters=16, kernel_size=(3, 3), padding="same", input_shape=(32, 32, 1)))
    model.add(LeakyReLU(0.1))
    model.add(Conv2D(filters=32, kernel_size=(3, 3), padding="same"))
    model.add(LeakyReLU(0.1))
    model.add(MaxPool2D(pool_size=(2, 2)))
    model.add(Flatten())
    model.add(Dense(32))
    model.add(LeakyReLU(0.1))
    model.add(Dense(10, activation='softmax'))

    adam = optimizers.Adam(learning_rate = 0.001)


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

    return model
```

```
model_1 = cnn_model_1()
```

```
model_1.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 32, 32, 16)        160

 leaky_re_lu (LeakyReLU)     (None, 32, 32, 16)        0

 conv2d_1 (Conv2D)           (None, 32, 32, 32)        4640

 leaky_re_lu_1 (LeakyReLU)   (None, 32, 32, 32)        0

 max_pooling2d (MaxPooling2D  (None, 16, 16, 32)       0
 )

 flatten (Flatten)           (None, 8192)              0

 dense_6 (Dense)             (None, 32)                262176

 leaky_re_lu_2 (LeakyReLU)   (None, 32)                0

 dense_7 (Dense)             (None, 10)                330

=================================================================
Total params: 267,306
Trainable params: 267,306
Non-trainable params: 0
_________________________________________________________________
```

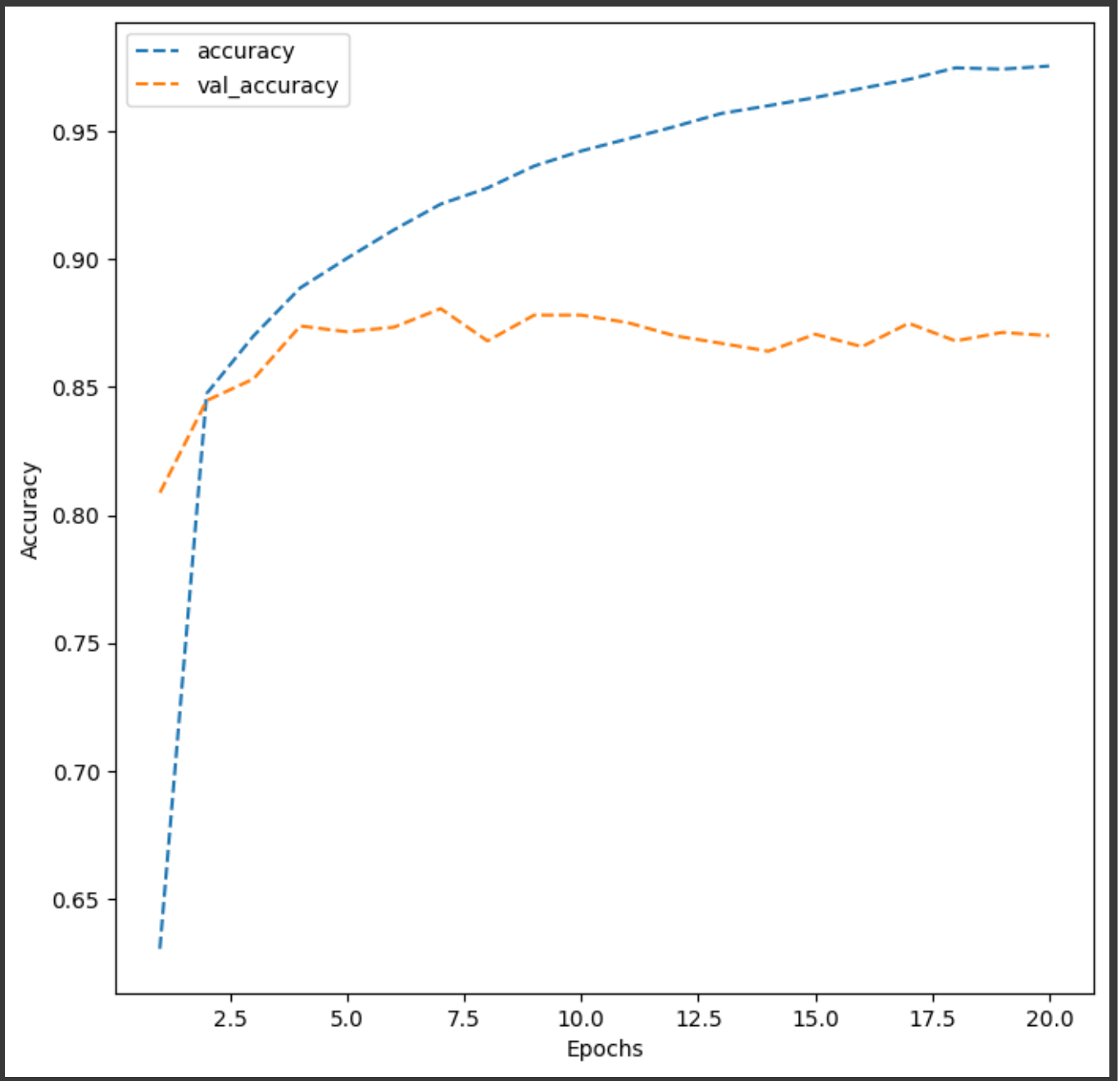
```
history_model_1 = model_1.fit(
            X_train, y_train_encoded,
            epochs=20,
            validation_split=0.2,
            batch_size = 32,
            verbose=1)
```

```
Epoch 1/20
1050/1050 [==============================] - 98s 93ms/step - loss: 1.1274 -
Epoch 2/20
1050/1050 [==============================] - 93s 88ms/step - loss: 0.5352 -
Epoch 3/20
1050/1050 [==============================] - 104s 99ms/step - loss: 0.4416
Epoch 4/20
1050/1050 [==============================] - 112s 107ms/step - loss: 0.3781
Epoch 5/20
1050/1050 [==============================] - 102s 97ms/step - loss: 0.3330
Epoch 6/20
1050/1050 [==============================] - 111s 105ms/step - loss: 0.2944
Epoch 7/20
1050/1050 [==============================] - 117s 111ms/step - loss: 0.2622
Epoch 8/20
1050/1050 [==============================] - 96s 92ms/step - loss: 0.2332 -
Epoch 9/20
1050/1050 [==============================] - 96s 91ms/step - loss: 0.2094 -
Epoch 10/20
1050/1050 [==============================] - 92s 87ms/step - loss: 0.1873 -
Epoch 11/20
1050/1050 [==============================] - 113s 108ms/step - loss: 0.1663
Epoch 12/20
1050/1050 [==============================] - 92s 87ms/step - loss: 0.1521 -
Epoch 13/20
1050/1050 [==============================] - 97s 92ms/step - loss: 0.1331 -
Epoch 14/20
1050/1050 [==============================] - 91s 87ms/step - loss: 0.1233 -
Epoch 15/20
1050/1050 [==============================] - 106s 101ms/step - loss: 0.1132
Epoch 16/20
1050/1050 [==============================] - 96s 92ms/step - loss: 0.1008 -
Epoch 17/20
1050/1050 [==============================] - 93s 89ms/step - loss: 0.0929 -
Epoch 18/20
1050/1050 [==============================] - 95s 90ms/step - loss: 0.0798 -
Epoch 19/20
1050/1050 [==============================] - 91s 87ms/step - loss: 0.0778 -
Epoch 20/20
1050/1050 [==============================] - 93s 89ms/step - loss: 0.0742 -
```

Plot the Training and Validation Accuracies and Write your observations.

```
dict_hist = history_model_1.history
list_ep = [i for i in range(1,21)]

plt.figure(figsize = (8,8))
plt.plot(list_ep,dict_hist['accuracy'],ls = '--', label = 'accuracy')
plt.plot(list_ep,dict_hist['val_accuracy'],ls = '--', label = 'val_accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epochs')
plt.legend()
plt.show()
```



Observations:

- Poor performance on the validation set. Model overfitting
- Less constant after epoch 2 but increases afterwards.

```
from tensorflow.keras import backend
backend.clear_session()
```

```
np.random.seed(42)
import random
random.seed(42)
tf.random.set_seed(42)
```

Let's build another model and see if we can get a better model with generalized performance.

First, we need to clear the previous model's history from the Keras backend. Also, let's fix the seed again after clearing the backend.

Second Model Architecture

- Write a function that returns a sequential model with the following architecture:
    - First Convolutional layer with **16 filters and the kernel size of 3x3**. Use the **'same' padding** and provide the **input shape = (32, 32, 1)**
    - Add a **LeakyRelu layer** with the **slope equal to 0.1**
    - Second Convolutional layer with **32 filters and the kernel size of 3x3 with 'same' padding**
    - Add **LeakyRelu** with the **slope equal to 0.1**
    - Add a **max-pooling layer** with a **pool size of 2x2**
    - Add a **BatchNormalization layer**
    - Third Convolutional layer with **32 filters and the kernel size of 3x3 with 'same' padding**
    - Add a **LeakyRelu layer with the slope equal to 0.1**
    - Fourth Convolutional layer **64 filters and the kernel size of 3x3 with 'same' padding**
    - Add a **LeakyRelu layer with the slope equal to 0.1**
    - Add a **max-pooling layer** with a **pool size of 2x2**
    - Add a **BatchNormalization layer**
    - **Flatten** the output from the previous layer
    - Add a **dense layer with 32 nodes**
    - Add a **LeakyRelu layer with the slope equal to 0.1**
    - Add a **dropout layer with the rate equal to 0.5**
    - Add the final **output layer with nodes equal to the number of classes, i.e., 10** and **'softmax' as the activation function**
    - Compile the model with the **categorical_crossentropy loss, adam optimizers (learning_rate = 0.001), and metric equal to 'accuracy'**. Do not fit the model here, just return the compiled model.
- Call the function cnn_model_2 and store the model in a new variable.
- Print the summary of the model.
- Fit the model on the train data with a **validation split of 0.2, batch size = 128, verbose = 1, and epochs = 30**. Store the model building history to use later for visualization.

Build and train the second CNN model as per the above mentioned architecture.

```
def cnn_model_2():
    model = Sequential()
    model.add(Conv2D(filters=16, kernel_size=(3, 3), padding="same", input_shape=(32, 32, 1)))
    model.add(LeakyReLU(0.1))
    model.add(Conv2D(filters=32, kernel_size=(3, 3), padding="same"))
    model.add(LeakyReLU(0.1))
    model.add(MaxPool2D(pool_size=(2, 2)))
    model.add(BatchNormalization())
    model.add(Conv2D(filters=32, kernel_size=(3, 3), padding="same"))
    model.add(LeakyReLU(0.1))
    model.add(Conv2D(filters=64, kernel_size=(3, 3), padding="same"))
    model.add(LeakyReLU(0.1))
    model.add(MaxPool2D(pool_size=(2, 2)))
    model.add(BatchNormalization())
    model.add(Flatten())
    model.add(Dense(32))
    model.add(LeakyReLU(0.1))
    model.add(Dropout(0.5))
    model.add(Dense(10, activation='softmax'))


    adam = optimizers.Adam(learning_rate = 0.001)


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

    return model
```

```
model_2 = cnn_model_2()
```

```
model_2.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 32, 32, 16)        160

 leaky_re_lu (LeakyReLU)     (None, 32, 32, 16)        0

 conv2d_1 (Conv2D)           (None, 32, 32, 32)        4640

 leaky_re_lu_1 (LeakyReLU)   (None, 32, 32, 32)        0

 max_pooling2d (MaxPooling2D  (None, 16, 16, 32)       0
 )

 batch_normalization (BatchN  (None, 16, 16, 32)       128
 ormalization)

 conv2d_2 (Conv2D)           (None, 16, 16, 32)        9248

 leaky_re_lu_2 (LeakyReLU)   (None, 16, 16, 32)        0

 conv2d_3 (Conv2D)           (None, 16, 16, 64)        18496

 leaky_re_lu_3 (LeakyReLU)   (None, 16, 16, 64)        0

 max_pooling2d_1 (MaxPooling  (None, 8, 8, 64)         0
 2D)

 batch_normalization_1 (Batc  (None, 8, 8, 64)         256
 hNormalization)

 flatten (Flatten)           (None, 4096)              0

 dense (Dense)               (None, 32)                131104

 leaky_re_lu_4 (LeakyReLU)   (None, 32)                0

 dropout (Dropout)           (None, 32)                0

 dense_1 (Dense)             (None, 10)                330

=================================================================
Total params: 164,362
Trainable params: 164,170
Non-trainable params: 192
_________________________________________________________________
```

```
history_model_2 = model_2.fit(
           X_train, y_train_encoded,
           epochs=30,
           validation_split=0.2,
           batch_size = 128,
           verbose=1)
```

```
Epoch 1/30
263/263 [==============================] - 158s 594ms/step - loss: 1.3271 -
Epoch 2/30
263/263 [==============================] - 157s 597ms/step - loss: 0.6722 -
Epoch 3/30
263/263 [==============================] - 154s 584ms/step - loss: 0.5530 -
Epoch 4/30
263/263 [==============================] - 155s 591ms/step - loss: 0.4921 -
Epoch 5/30
263/263 [==============================] - 153s 584ms/step - loss: 0.4399 -
Epoch 6/30
263/263 [==============================] - 156s 593ms/step - loss: 0.4033 -
Epoch 7/30
263/263 [==============================] - 154s 585ms/step - loss: 0.3876 -
Epoch 8/30
263/263 [==============================] - 154s 586ms/step - loss: 0.3505 -
Epoch 9/30
263/263 [==============================] - 196s 747ms/step - loss: 0.3317 -
Epoch 10/30
263/263 [==============================] - 182s 692ms/step - loss: 0.3098 -
Epoch 11/30
263/263 [==============================] - 168s 641ms/step - loss: 0.2903 -
Epoch 12/30
263/263 [==============================] - 159s 604ms/step - loss: 0.2815 -
Epoch 13/30
263/263 [==============================] - 154s 586ms/step - loss: 0.2723 -
Epoch 14/30
263/263 [==============================] - 157s 597ms/step - loss: 0.2562 -
Epoch 15/30
263/263 [==============================] - 157s 594ms/step - loss: 0.2440 -
Epoch 16/30
263/263 [==============================] - 154s 588ms/step - loss: 0.2309 -
Epoch 17/30
263/263 [==============================] - 153s 583ms/step - loss: 0.2201 -
Epoch 18/30
263/263 [==============================] - 154s 587ms/step - loss: 0.2141 -
Epoch 19/30
263/263 [==============================] - 151s 572ms/step - loss: 0.2078 -
Epoch 20/30
263/263 [==============================] - 151s 575ms/step - loss: 0.2058 -
Epoch 21/30
263/263 [==============================] - 152s 577ms/step - loss: 0.1957 -
Epoch 22/30
263/263 [==============================] - 151s 576ms/step - loss: 0.1852 -
Epoch 23/30
263/263 [==============================] - 152s 577ms/step - loss: 0.1844 -
Epoch 24/30
263/263 [==============================] - 153s 582ms/step - loss: 0.1797 -
Epoch 25/30
263/263 [==============================] - ETA: 0s - loss: 0.1721 - accurac
```

Plot the Training and Validation accuracies and write your observations.

```
dict_hist = history_model_2.history
list_ep = [i for i in range(1,31)]

plt.figure(figsize = (8,8))
plt.plot(list_ep,dict_hist['accuracy'],ls = '--', label = 'accuracy')
plt.plot(list_ep,dict_hist['val_accuracy'],ls = '--', label = 'val_accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epochs')
plt.legend()
plt.show()
```

Observations:

- The second model with dropout layers appears to have decreased the overfitting when contrasted with the past model yet at the same time the approval information accuracy is around 5% lower than train information precision.
- The overal precision of this model is superior to the first model.
- The Preparation precision is expanding with the expansion in ages.
- It has a quick increment up to around 5 ages and afterward appears to have exceptionally low

Predictions on the test data

- Make predictions on the test set using the second model.
- Print the obtained results using the classification report and the confusion matrix.
- Final observations on the obtained results.

Make predictions on the test data using the second model.

```
test_pred = model_2.predict(X_test)

test_pred = np.argmax(test_pred, axis=-1)
```

Note: Earlier, we noticed that each entry of the target variable is a one-hot encoded vector, but to print the classification report and confusion matrix, we must convert each entry of y_test to a single label.

```
y_test = np.argmax(y_test, axis=-1)
```

Write your final observations on the performance of the model on the test data.

```
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix


print(classification_report(y_test, test_pred))


cm = confusion_matrix(y_test, test_pred)
plt.figure(figsize=(8,5))
sns.heatmap(cm, annot=True,  fmt='.0f')
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.show()
```

Final Observations:

- The arrangement report lets us know that numbers 0 and 4 have the most elevated f1-score (0.81) meaning they have the best possibilities being precisely perceived.
- While, numbers 3 and 8 have the most reduced f1-score of (0.77).
- Number 3 has the most minimal accuracy and 0 has the most elevated, this implies that the model is classifing different numbers as 3 which not legitimate.
- Though 0 has the least possibilities being invalid

Colab paid products - Cancel contracts here