Convolutional Neural Networks Project: Street View Housing Number Digit Recognition

---

Objective

To build a CNN model that can recognize the digits in the images.

Dataset

Here, I 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

I started by mounting the Google drive.

In [26]:

```
from google.colab import drive

drive.mount('/content/drive')
```

Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).

Importing the necessary libraries

In [27]:

```
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, Flatten, LeakyReLU

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.

In [28]:

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

2.8.0

Load the dataset

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

In [30]:

```
import h5py

# Open the file as read only
# can make changes in the path as required

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

```
# Load the the train and the test dataset

X_train = h5f['X_train'][:]

y_train = h5f['y_train'][:]

X_test = h5f['X_test'][:]

y_test = h5f['y_test'][:]


# Close this file

h5f.close()
```

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

In [31]:

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

Out[31]:

(42000, 18000)

Observation:

- There are 42,000 images in the training data and 18,000 images in the testing data.

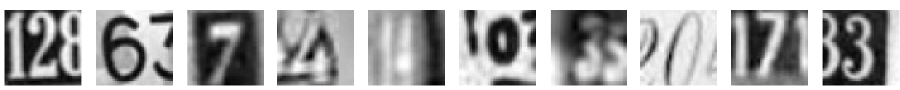
Visualizing images

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

visualizing the first 10 images in the dataset

In [32]:

```
# Visualizing the first 10 images in the dataset and printing their labels

%matplotlib inline

import matplotlib.pyplot as plt

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

for i in range(10):

    plt.subplot(1, 10, i+1)

    plt.imshow(X_train[i], cmap = "gray")  # Write the function to visualize images

    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]

For 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.

In [33]:

```
# Shape and the array of pixels for the first image

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]]
```

In [34]:

```
# Reshaping 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)
```

In [35]:

```
# Normalize inputs from 0-255 to 0-1

X_train = X_train / 255.0

X_test = X_test / 255.0
```

In [36]:

```
# New shape

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,)
```

encode the labels in the target variable y_train and y_test

In [37]:

```
# Write the function and appropriate variable name to one-hot encode the output

y_train = tf.keras.utils.to_categorical(y_train)

y_test = tf.keras.utils.to_categorical(y_test)

# test labels

y_test
```

Out[37]:

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

Observation:

- Notice that each entry of the target variable is a one-hot encoded vector instead of a single label.

Model Building

Now that I have done data preprocessing, let's build a CNN model.

In [38]:

```
# Fixing the seed for random number generators

np.random.seed(42)

import random

random.seed(42)

tf.random.set_seed(42)
```

Model Architecture

- **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

In [41]:

```
# Define the model

def cnn_model_1():

    model = Sequential()

    # Add layers as per the architecture mentioned above in the same sequence

    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'))

    # Compile the model

    model.compile(loss='categorical_crossentropy', optimizer=tf.keras.optimizers.Adam(learning_rate = 0.001), metrics=['accuracy'])

    return model
```

In [42]:

```
# Build the model

model_1 = cnn_model_1()
```

In [43]:

```
# Print the model summary

model_1.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_2 (Conv2D)           (None, 32, 32, 16)        160

 leaky_re_lu_3 (LeakyReLU)   (None, 32, 32, 16)        0

 conv2d_3 (Conv2D)           (None, 32, 32, 32)        4640

 leaky_re_lu_4 (LeakyReLU)   (None, 32, 32, 32)        0

 max_pooling2d_1 (MaxPooling  (None, 16, 16, 32)       0
 2D)

 flatten_1 (Flatten)         (None, 8192)              0

 dense_2 (Dense)             (None, 32)                262176

 leaky_re_lu_5 (LeakyReLU)   (None, 32)                0

 dense_3 (Dense)             (None, 10)                330

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

In [44]:

```
# Fit the model

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

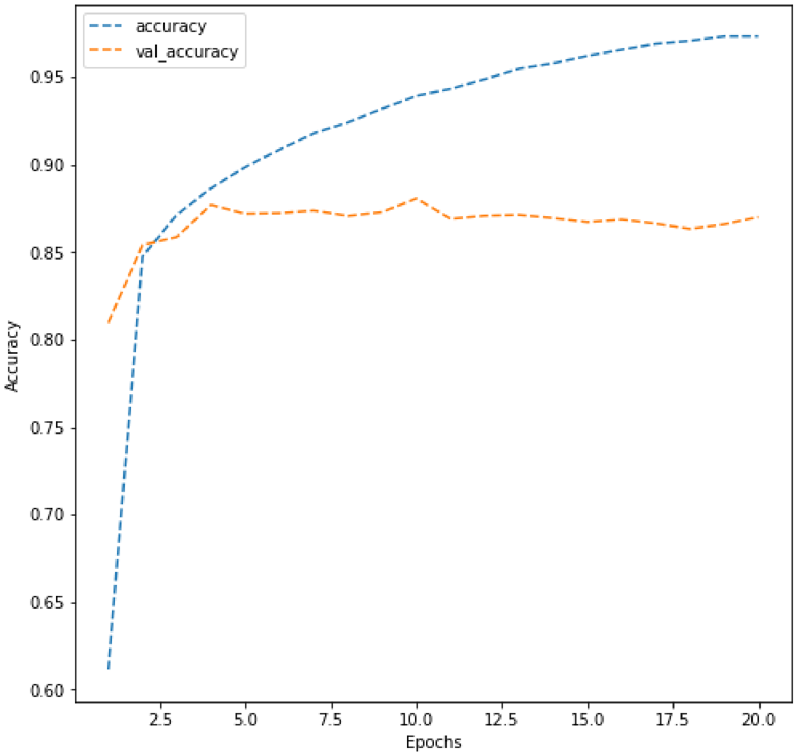
```
Epoch 1/20
1050/1050 [==============================] - 62s 58ms/step - loss: 1.1710 - accuracy: 0.6114 - val_loss: 0.6506 - val_accuracy: 0.8094
Epoch 2/20
1050/1050 [==============================] - 61s 58ms/step - loss: 0.5303 - accuracy: 0.8474 - val_loss: 0.5146 - val_accuracy: 0.8539
Epoch 3/20
1050/1050 [==============================] - 61s 58ms/step - loss: 0.4467 - accuracy: 0.8711 - val_loss: 0.4945 - val_accuracy: 0.8585
Epoch 4/20
1050/1050 [==============================] - 61s 58ms/step - loss: 0.3886 - accuracy: 0.8863 - val_loss: 0.4423 - val_accuracy: 0.8768
Epoch 5/20
1050/1050 [==============================] - 61s 58ms/step - loss: 0.3420 - accuracy: 0.8984 - val_loss: 0.4656 - val_accuracy: 0.8717
Epoch 6/20
1050/1050 [==============================] - 61s 58ms/step - loss: 0.3060 - accuracy: 0.9083 - val_loss: 0.4702 - val_accuracy: 0.8720
Epoch 7/20
1050/1050 [==============================] - 61s 58ms/step - loss: 0.2751 - accuracy: 0.9176 - val_loss: 0.4600 - val_accuracy: 0.8736
Epoch 8/20
1050/1050 [==============================] - 61s 58ms/step - loss: 0.2480 - accuracy: 0.9239 - val_loss: 0.4867 - val_accuracy: 0.8705
Epoch 9/20
1050/1050 [==============================] - 61s 58ms/step - loss: 0.2245 - accuracy: 0.9317 - val_loss: 0.4888 - val_accuracy: 0.8726
Epoch 10/20
1050/1050 [==============================] - 61s 58ms/step - loss: 0.1994 - accuracy: 0.9391 - val_loss: 0.4785 - val_accuracy: 0.8805
Epoch 11/20
1050/1050 [==============================] - 61s 58ms/step - loss: 0.1828 - accuracy: 0.9430 - val_loss: 0.5438 - val_accuracy: 0.8688
Epoch 12/20
1050/1050 [==============================] - 61s 58ms/step - loss: 0.1641 - accuracy: 0.9485 - val_loss: 0.5645 - val_accuracy: 0.8706
Epoch 13/20
1050/1050 [==============================] - 62s 59ms/step - loss: 0.1476 - accuracy: 0.9546 - val_loss: 0.5580 - val_accuracy: 0.8711
Epoch 14/20
1050/1050 [==============================] - 61s 59ms/step - loss: 0.1359 - accuracy: 0.9576 - val_loss: 0.5841 - val_accuracy: 0.8693
Epoch 15/20
1050/1050 [==============================] - 61s 58ms/step - loss: 0.1228 - accuracy: 0.9618 - val_loss: 0.6122 - val_accuracy: 0.8669
Epoch 16/20
1050/1050 [==============================] - 61s 58ms/step - loss: 0.1092 - accuracy: 0.9654 - val_loss: 0.6472 - val_accuracy: 0.8685
Epoch 17/20
1050/1050 [==============================] - 62s 59ms/step - loss: 0.0992 - accuracy: 0.9687 - val_loss: 0.6768 - val_accuracy: 0.8661
Epoch 18/20
1050/1050 [==============================] - 62s 59ms/step - loss: 0.0919 - accuracy: 0.9704 - val_loss: 0.7570 - val_accuracy: 0.8631
Epoch 19/20
1050/1050 [==============================] - 62s 59ms/step - loss: 0.0824 - accuracy: 0.9731 - val_loss: 0.7279 - val_accuracy: 0.8657
Epoch 20/20
1050/1050 [==============================] - 62s 59ms/step - loss: 0.0829 - accuracy: 0.9731 - val_loss: 0.7561 - val_accuracy: 0.8699
```

Plotting the validation and training accuracies

Write your observations on the below plot.

In [45]:

```
# Plotting the accuracies

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:__

--The model did poorly on the validation data; the model looks overfitting on the training data.

--The validation accuracy didn't really change after 2.5 epochs.

--The increasing rate of accuracy is lower after 2.5 epochs.
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.

In [46]:

```
# Clearing backend

from tensorflow.keras import backend

backend.clear_session()
```

In [47]:

```
# Fixing the seed for random number generators

np.random.seed(42)

import random

random.seed(42)

tf.random.set_seed(42)
```

Second Model Architecture

- 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

In [52]:

```
# Define the model

def cnn_model_2():

    model = Sequential()

    # Add layers as per the architecture mentioned above in the same sequence
    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'))

    # Compile the model

    model.compile(
      loss='categorical_crossentropy',
      optimizer=tf.keras.optimizers.Adam(learning_rate = 0.001),
      metrics=['accuracy'])
```

```
    return model
```

In [53]:

```
# Build the model

model_2 = cnn_model_2()
```

In [54]:

```
# Print the summary

model_2.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 conv2d_8 (Conv2D)           (None, 32, 32, 16)        160       
                                                                 
 leaky_re_lu_10 (LeakyReLU)  (None, 32, 32, 16)        0         
                                                                 
 conv2d_9 (Conv2D)           (None, 32, 32, 32)        4640      
                                                                 
 leaky_re_lu_11 (LeakyReLU)  (None, 32, 32, 32)        0         
                                                                 
 max_pooling2d_4 (MaxPooling  (None, 16, 16, 32)       0         
 2D)                                                             
                                                                 
 batch_normalization_4 (Batc  (None, 16, 16, 32)       128       
 hNormalization)                                                 
                                                                 
 conv2d_10 (Conv2D)          (None, 16, 16, 32)        9248      
                                                                 
 leaky_re_lu_12 (LeakyReLU)  (None, 16, 16, 32)        0         
                                                                 
 conv2d_11 (Conv2D)          (None, 16, 16, 64)        18496     
                                                                 
 leaky_re_lu_13 (LeakyReLU)  (None, 16, 16, 64)        0         
                                                                 
 max_pooling2d_5 (MaxPooling  (None, 8, 8, 64)         0         
 2D)                                                             
                                                                 
 batch_normalization_5 (Batc  (None, 8, 8, 64)         256       
 hNormalization)                                                 
                                                                 
 flatten_2 (Flatten)         (None, 4096)              0         
                                                                 
 dense_2 (Dense)             (None, 32)                131104    
                                                                 
 leaky_re_lu_14 (LeakyReLU)  (None, 32)                0         
                                                                 
 dropout (Dropout)           (None, 32)                0         
                                                                 
 dense_3 (Dense)             (None, 10)                330       
                                                                 
=================================================================
Total params: 164,362
Trainable params: 164,170
Non-trainable params: 192
_________________________________________________________________
```

In [56]:

```
# Fit the model

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

```
Epoch 1/30
263/263 [==============================] - 100s 379ms/step - loss: 1.3502 - accuracy: 0.5405 - val_loss: 1.7414 - val_accuracy: 0.4129
Epoch 2/30
263/263 [==============================] - 100s 380ms/step - loss: 0.6604 - accuracy: 0.7981 - val_loss: 0.5678 - val_accuracy: 0.8377
Epoch 3/30
263/263 [==============================] - 99s 378ms/step - loss: 0.5462 - accuracy: 0.8318 - val_loss: 0.4907 - val_accuracy: 0.8518
Epoch 4/30
263/263 [==============================] - 99s 378ms/step - loss: 0.4801 - accuracy: 0.8541 - val_loss: 0.4162 - val_accuracy: 0.8855
Epoch 5/30
263/263 [==============================] - 99s 378ms/step - loss: 0.4417 - accuracy: 0.8653 - val_loss: 0.4543 - val_accuracy: 0.8729
Epoch 6/30
263/263 [==============================] - 99s 378ms/step - loss: 0.3988 - accuracy: 0.8798 - val_loss: 0.5975 - val_accuracy: 0.8504
Epoch 7/30
263/263 [==============================] - 99s 378ms/step - loss: 0.3778 - accuracy: 0.8850 - val_loss: 0.3670 - val_accuracy: 0.8980
Epoch 8/30
263/263 [==============================] - 99s 378ms/step - loss: 0.3506 - accuracy: 0.8910 - val_loss: 0.4012 - val_accuracy: 0.8889
Epoch 9/30
263/263 [==============================] - 99s 378ms/step - loss: 0.3266 - accuracy: 0.8983 - val_loss: 0.3478 - val_accuracy: 0.9021
Epoch 10/30
263/263 [==============================] - 100s 379ms/step - loss: 0.3006 - accuracy: 0.9057 - val_loss: 0.3743 - val_accuracy: 0.8961
Epoch 11/30
263/263 [==============================] - 100s 379ms/step - loss: 0.2906 - accuracy: 0.9081 - val_loss: 0.3583 - val_accuracy: 0.9070
Epoch 12/30
263/263 [==============================] - 100s 380ms/step - loss: 0.2747 - accuracy: 0.9151 - val_loss: 0.4322 - val_accuracy: 0.8960
Epoch 13/30
263/263 [==============================] - 100s 381ms/step - loss: 0.2655 - accuracy: 0.9165 - val_loss: 0.3743 - val_accuracy: 0.9026
Epoch 14/30
263/263 [==============================] - 100s 380ms/step - loss: 0.2503 - accuracy: 0.9206 - val_loss: 0.4257 - val_accuracy: 0.8999
Epoch 15/30
263/263 [==============================] - 100s 382ms/step - loss: 0.2389 - accuracy: 0.9215 - val_loss: 0.4350 - val_accuracy: 0.9042
Epoch 16/30
263/263 [==============================] - 101s 383ms/step - loss: 0.2320 - accuracy: 0.9261 - val_loss: 0.3679 - val_accuracy: 0.9105
Epoch 17/30
263/263 [==============================] - 101s 383ms/step - loss: 0.2238 - accuracy: 0.9282 - val_loss: 0.4124 - val_accuracy: 0.9055
Epoch 18/30
263/263 [==============================] - 100s 382ms/step - loss: 0.2167 - accuracy: 0.9297 - val_loss: 0.3827 - val_accuracy: 0.9061
Epoch 19/30
263/263 [==============================] - 101s 383ms/step - loss: 0.2074 - accuracy: 0.9322 - val_loss: 0.3811 - val_accuracy: 0.9024
Epoch 20/30
263/263 [==============================] - 100s 382ms/step - loss: 0.2013 - accuracy: 0.9333 - val_loss: 0.4015 - val_accuracy: 0.9067
Epoch 21/30
263/263 [==============================] - 100s 382ms/step - loss: 0.1950 - accuracy: 0.9369 - val_loss: 0.4312 - val_accuracy: 0.9079
Epoch 22/30
263/263 [==============================] - 101s 383ms/step - loss: 0.1877 - accuracy: 0.9394 - val_loss: 0.4362 - val_accuracy: 0.9111
Epoch 23/30
263/263 [==============================] - 101s 383ms/step - loss: 0.1829 - accuracy: 0.9410 - val_loss: 0.4330 - val_accuracy: 0.9075
Epoch 24/30
263/263 [==============================] - 101s 383ms/step - loss: 0.1773 - accuracy: 0.9415 - val_loss: 0.4011 - val_accuracy: 0.9118
Epoch 25/30
263/263 [==============================] - 101s 384ms/step - loss: 0.1664 - accuracy: 0.9449 - val_loss: 0.4763 - val_accuracy: 0.9087
Epoch 26/30
263/263 [==============================] - 101s 383ms/step - loss: 0.1670 - accuracy: 0.9445 - val_loss: 0.4698 - val_accuracy: 0.9090
Epoch 27/30
263/263 [==============================] - 101s 382ms/step - loss: 0.1640 - accuracy: 0.9452 - val_loss: 0.3981 - val_accuracy: 0.9115
Epoch 28/30
263/263 [==============================] - 101s 384ms/step - loss: 0.1526 - accuracy: 0.9487 - val_loss: 0.5173 - val_accuracy: 0.9032
Epoch 29/30
263/263 [==============================] - 101s 383ms/step - loss: 0.1509 - accuracy: 0.9493 - val_loss: 0.4352 - val_accuracy: 0.9112
Epoch 30/30
263/263 [==============================] - 101s 382ms/step - loss: 0.1483 - accuracy: 0.9506 - val_loss: 0.4499 - val_accuracy: 0.9125
```

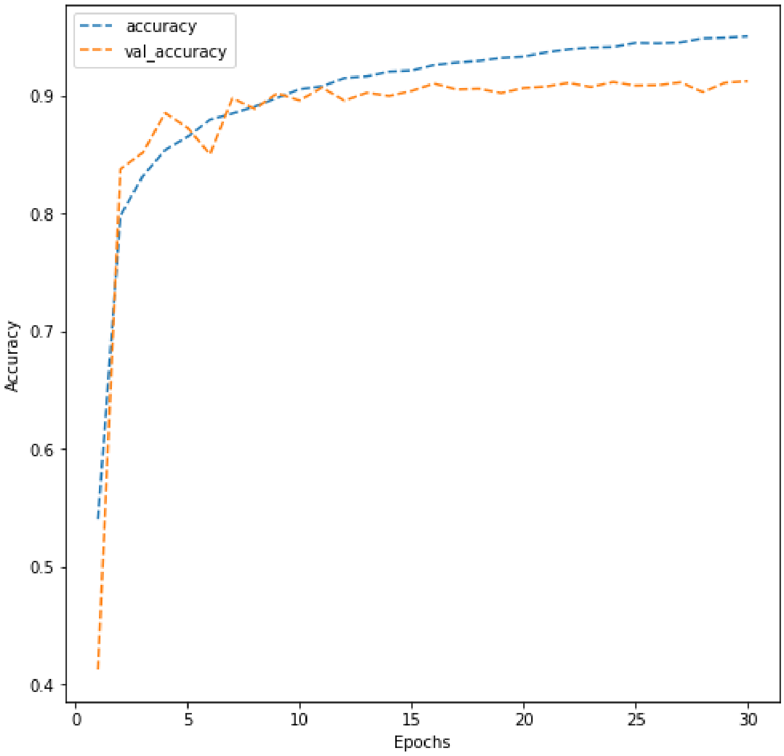
Plotting the validation and training accuracies

Write your observations on the below plot

In [58]:

```
# Plotting the accuracies

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 reduced the overfitting as compared to the previous model but still the validation data accuracy is a bit lower than train accuracy.

--There is a rapid increase up to around 5 epochs and then seems to have very lower increase after that.

--The validation accuracy is fluctuating but generally it is also increasing with the increase in epochs.

--The test model is giving close to 90% accuracy at 30 epochs while accuracy for training model for 30 epochs, is about 95%.

--Accuracy of this model is better than the 1st 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.

Make predictions on the test data using the second model

In [59]:

```
# Make prediction on the test data using model_2

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.

In [60]:

```
# Converting each entry to single label from one-hot encoded vector

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

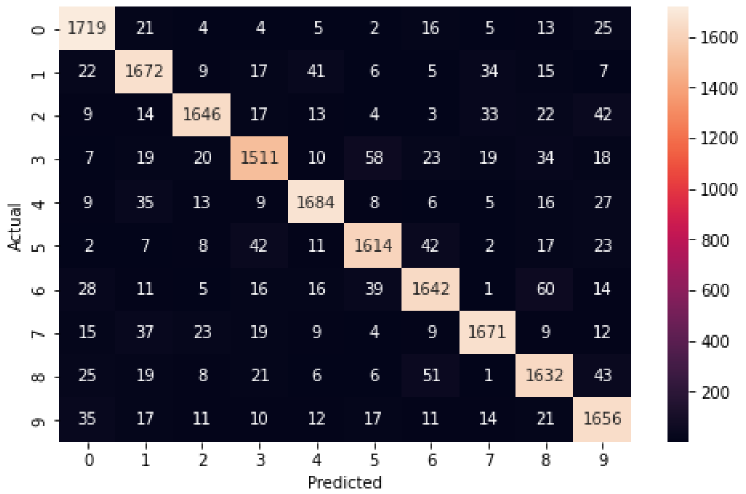
Write your final observations on the performance of the model on the test data.

In [61]:

```
# Importing required functions

from sklearn.metrics import classification_report

from sklearn.metrics import confusion_matrix

# Printing the classification report
```

```
print(classification_report(y_test, test_pred))

# Plotting the heatmap using confusion matrix

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       0.92      0.95      0.93      1814
           1       0.90      0.91      0.91      1828
           2       0.94      0.91      0.93      1803
           3       0.91      0.88      0.89      1719
           4       0.93      0.93      0.93      1812
           5       0.92      0.91      0.92      1768
           6       0.91      0.90      0.90      1832
           7       0.94      0.92      0.93      1808
           8       0.89      0.90      0.89      1812
           9       0.89      0.92      0.90      1804

    accuracy                           0.91     18000
   macro avg       0.91      0.91      0.91     18000
weighted avg       0.91      0.91      0.91     18000
```



Final Observations:_

--The model gives about 90% accuracy on the test data which is comparable to the accuracy of the validation data.

--This implies that the model is giving a generalized performance.

--The recall has a very high range (82-95)% which implies that the model is good at identifying most of the objects. Model is able to identify about 95% of image 0 but can only identify only ~82% of image 3.

--Generally the model could distiguish individual digits well.

-- CNN works better than ANN when identifying images.

In [ ]: