Convolutional Neural Networks: Street View Housing Number Digit Recognition

Welcome to the project on classification using Convolutional Neural Networks. We will continue to work with the Street View Housing Numbers (SVHN) image dataset for this project.

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:

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

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.

In [ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

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

Importing the necessary libraries

In [ ]:
```
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, BatchNormaliza
from tensorflow.keras.utils import to_categorical
```

Let us check for the version of tensorflow.

In [ ]:
```
print(tf.__version__)
```

```
2.8.0
```

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.

In [ ]:
```
import h5py

# Open the file as read only
# User can make changes in the path as required
h5f = h5py.File('/content/drive/MyDrive/SVHN_single_grey1.h5', 'r')

# Load the training and the test set
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()
```

Let's check the number of images in the training and the testing dataset.

In [ ]:
```
len(X_train), len(X_test)
```

Out[ ]:
```
(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.

In [ ]:
```
# Visualizing the first 10 images in the dataset and 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")
    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.
- 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 set.
- One-hot encode the target variable.

In [ ]:
```
# Shape of the images and 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 [ ]:
```
# Reshaping the dataset to flatten them. Remember that we always have to giv
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 [ ]:
```
# Normalize inputs from 0-255 to 0-1
X_train = X_train / 255.0
X_test = X_test / 255.0
```

In [ ]:
```
# 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,)
```

In [ ]:
```
# One-hot encode output
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)

# Test labels
y_test
```

Out[ ]:
```
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, we have done data preprocessing, let's build a CNN model.

In [ ]:
```
# Fixing the seed for random number generators
np.random.seed(42)

import random
random.seed(42)

tf.random.set_seed(42)
```

Model Architecture

- Let's build a model with the following architecture,

- First Convolutional layer with **16 filters and a 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 a 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 a slope equal to 0.1**
- Add the final **output layer with nodes equal to the number of classes** and **softmax activation**
- Compile the model with the **categorical_crossentropy loss, adam optimizers (learning_rate = 0.001), and accuracy metric**.
- Print the summary of the model.
- Fit the model on the train data with a **validation split of 0.2, batch size = 32, verbose = 1, and 20 epochs**. Store the model building history to use later for visualization.

In [ ]:
```
# Define model

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


model1 = Sequential()
model1.add(Conv2D(filters=16, kernel_size=(3,3), padding="same", input_shape
model1.add(LeakyReLU(0.1))
model1.add(Conv2D(filters=32, kernel_size=(3,3), padding='same'))
model1.add(LeakyReLU(0.1))
model1.add(MaxPool2D(pool_size=(2,2)))
model1.add(Flatten())
model1.add(Dense(32))
model1.add(LeakyReLU(0.1))
model1.add(Dense(10, activation='softmax'))

adam = optimizers.Adam(learning_rate=0.001)
model1.compile(loss=losses.categorical_crossentropy, optimizer=adam, metrics
```

In [ ]:
```
# Model summary
model1.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
 )

 flatten (Flatten)           (None, 8192)              0

 dense (Dense)               (None, 32)                262176

 leaky_re_lu_2 (LeakyReLU)   (None, 32)                0

 dense_1 (Dense)             (None, 10)                330

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

- The model has 2,67,306 parameters. The majority of parameters belong to the single dense layer with 32 nodes.
- All the parameters are trainable.

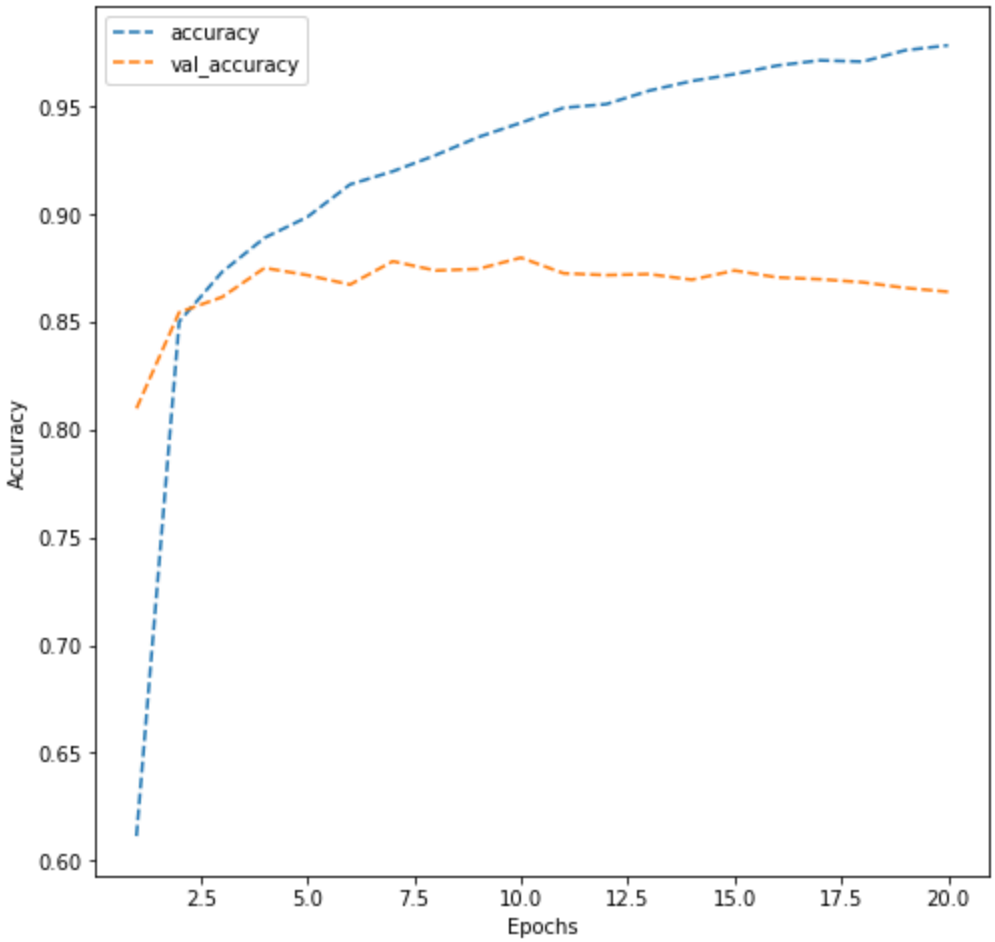
In [ ]:
```
# Fit the model
history_model_1 = model1.fit(X_train, y_train, validation_split=0.2, epochs=
```

```
Epoch 1/20
1050/1050 [==============================] - 16s 4ms/step - loss: 1.1700 - a
ccuracy: 0.6114 - val_loss: 0.6528 - val_accuracy: 0.8100
Epoch 2/20
1050/1050 [==============================] - 4s 4ms/step - loss: 0.5289 - ac
curacy: 0.8503 - val_loss: 0.5177 - val_accuracy: 0.8546
Epoch 3/20
1050/1050 [==============================] - 4s 4ms/step - loss: 0.4371 - ac
curacy: 0.8733 - val_loss: 0.4935 - val_accuracy: 0.8617
Epoch 4/20
1050/1050 [==============================] - 4s 4ms/step - loss: 0.3815 - ac
curacy: 0.8893 - val_loss: 0.4525 - val_accuracy: 0.8752
Epoch 5/20
1050/1050 [==============================] - 4s 4ms/step - loss: 0.3379 - ac
curacy: 0.8989 - val_loss: 0.4589 - val_accuracy: 0.8719
Epoch 6/20
1050/1050 [==============================] - 4s 4ms/step - loss: 0.2940 - ac
curacy: 0.9140 - val_loss: 0.4758 - val_accuracy: 0.8675
Epoch 7/20
1050/1050 [==============================] - 4s 4ms/step - loss: 0.2630 - ac
curacy: 0.9201 - val_loss: 0.4476 - val_accuracy: 0.8783
Epoch 8/20
1050/1050 [==============================] - 4s 4ms/step - loss: 0.2366 - ac
curacy: 0.9276 - val_loss: 0.4872 - val_accuracy: 0.8740
Epoch 9/20
1050/1050 [==============================] - 4s 4ms/step - loss: 0.2114 - ac
curacy: 0.9360 - val_loss: 0.4958 - val_accuracy: 0.8748
Epoch 10/20
1050/1050 [==============================] - 4s 4ms/step - loss: 0.1868 - ac
curacy: 0.9426 - val_loss: 0.4869 - val_accuracy: 0.8800
Epoch 11/20
1050/1050 [==============================] - 4s 4ms/step - loss: 0.1639 - ac
curacy: 0.9496 - val_loss: 0.5503 - val_accuracy: 0.8727
Epoch 12/20
1050/1050 [==============================] - 4s 4ms/step - loss: 0.1535 - ac
curacy: 0.9513 - val_loss: 0.5518 - val_accuracy: 0.8719
Epoch 13/20
1050/1050 [==============================] - 4s 4ms/step - loss: 0.1338 - ac
curacy: 0.9576 - val_loss: 0.5839 - val_accuracy: 0.8724
Epoch 14/20
1050/1050 [==============================] - 4s 4ms/step - loss: 0.1214 - ac
curacy: 0.9621 - val_loss: 0.6357 - val_accuracy: 0.8698
Epoch 15/20
1050/1050 [==============================] - 4s 4ms/step - loss: 0.1098 - ac
curacy: 0.9652 - val_loss: 0.6370 - val_accuracy: 0.8740
Epoch 16/20
1050/1050 [==============================] - 4s 4ms/step - loss: 0.0990 - ac
curacy: 0.9692 - val_loss: 0.6700 - val_accuracy: 0.8708
Epoch 17/20
1050/1050 [==============================] - 4s 4ms/step - loss: 0.0892 - ac
curacy: 0.9716 - val_loss: 0.7333 - val_accuracy: 0.8700
Epoch 18/20
1050/1050 [==============================] - 4s 4ms/step - loss: 0.0887 - ac
curacy: 0.9710 - val_loss: 0.7378 - val_accuracy: 0.8686
Epoch 19/20
1050/1050 [==============================] - 4s 4ms/step - loss: 0.0734 - ac
```

```
curacy: 0.9763 - val_loss: 0.7868 - val_accuracy: 0.8660
Epoch 20/20
1050/1050 [==============================] - 4s 4ms/step - loss: 0.0652 - ac
curacy: 0.9786 - val_loss: 0.8368 - val_accuracy: 0.8642
```

Plotting the validation and training accuracies

In [ ]:
```
# 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 accuracy on the train set is much better (about 12%) than the validation set. We can say that model is overfitting the training data.

- The plot shows that training accuracy is increasing with epochs but the validation accuracy is more or less constant after 5 epochs.
- We can try adding dropout layers to the model's architecture to avoid overfitting. Also, we can add more convolutional layers for feature extraction.

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.

In [ ]:
```
# Clearing backend
from tensorflow.keras import backend
backend.clear_session()
```

In [ ]:
```
# Fixing the seed for random number generators
np.random.seed(42)

import random
random.seed(42)

tf.random.set_seed(42)
```

Second Model Architecture

- Let's build a second model with the following architecture,
- First Convolutional layer with **16 filters and a 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 a 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 a kernel size of 3x3 with 'same' padding**
- Add a **LeakyRelu layer with a slope equal to 0.1**
- Fourth Convolutional layer **64 filters and a kernel size of 3x3 with 'same' padding**
- Add a **LeakyRelu layer with a 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 a slope equal to 0.1**
- Add a **dropout layer with a rate equal to 0.5**

- Add the final **output layer with nodes equal to the number of classes** and **softmax activation**
- Compile the model with the **categorical_crossentropy loss, adam optimizers (learning_rate = 0.001), and accuracy metric**.
- 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 30 epochs**. Store the model building history to use later for visualization.

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

In [ ]:
```
model2 = Sequential()
model2.add(Conv2D(filters=16, kernel_size=(3,3), padding="same", input_shape
model2.add(LeakyReLU(0.1))
model2.add(Conv2D(filters=32, kernel_size=(3,3), padding='same'))
model2.add(LeakyReLU(0.1))
model2.add(MaxPool2D(pool_size=(2,2)))
model2.add(BatchNormalization())
model2.add(Conv2D(filters=32, kernel_size=(3,3), padding='same'))
model2.add(LeakyReLU(0.1))
model2.add(Conv2D(filters=64, kernel_size=(3,3), padding='same'))
model2.add(LeakyReLU(0.1))
model2.add(MaxPool2D(pool_size=(2,2)))
model2.add(BatchNormalization())
model2.add(Flatten())
model2.add(Dense(32))
model2.add(LeakyReLU(0.1))
model2.add(Dropout(0.5))
model2.add(Dense(10, activation='softmax'))

adam = optimizers.Adam(learning_rate=1e-3)
model2.compile(loss=losses.categorical_crossentropy, optimizer=adam, metrics
```

In [ ]:
```
# Model summary
model2.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
_________________________________________________________________
```

Observations:

- Although we have added layers to the model's architecture, the total number of parameters has decreased substantially (approx 40%) compared to the previous model.
- This is due to the additional max-pooling layer that has further reduced the size of images before passing them to the dense layer. Also, we have added a dropout layer

to the model.

- All the parameters are trainable.

In [ ]:
```
# Fit the model
history_model_2 = model2.fit(X_train, y_train, validation_split=0.2, epochs=
```

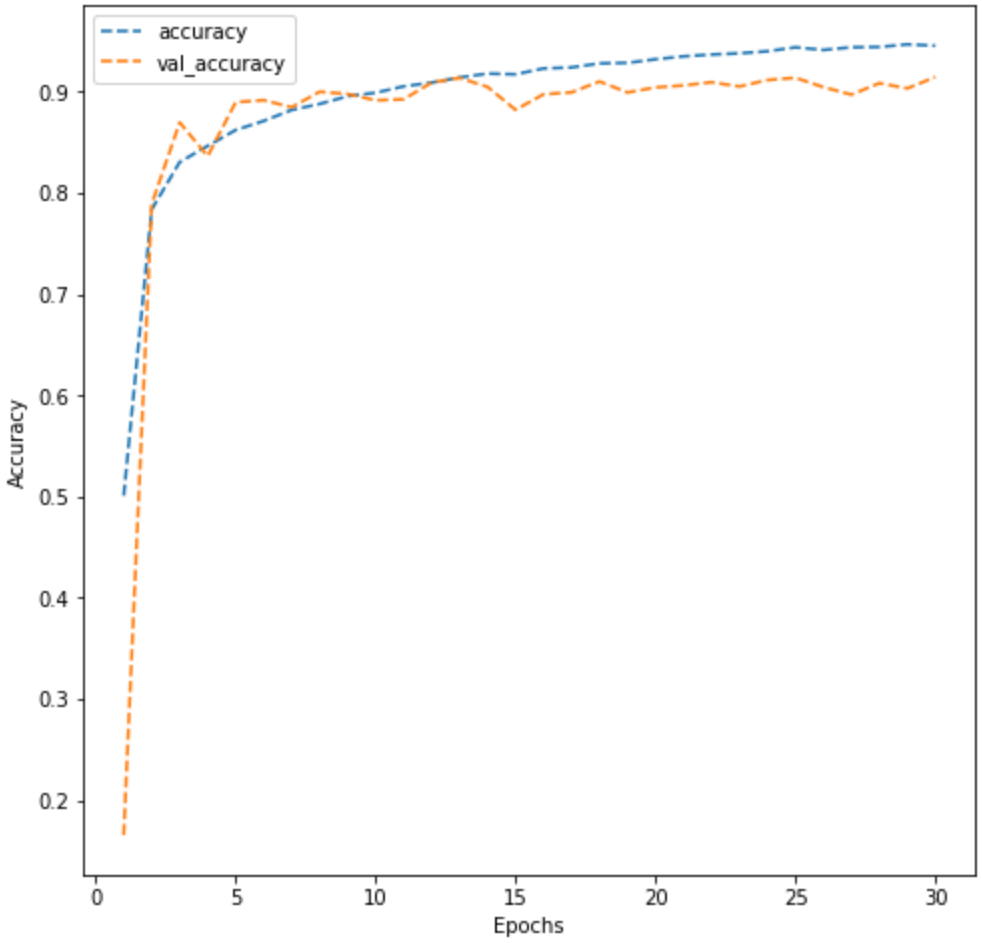
```
Epoch 1/30
263/263 [==============================] - 4s 11ms/step - loss: 1.4630 - acc
uracy: 0.5010 - val_loss: 2.8419 - val_accuracy: 0.1656
Epoch 2/30
263/263 [==============================] - 2s 9ms/step - loss: 0.6960 - accu
racy: 0.7839 - val_loss: 0.6896 - val_accuracy: 0.7892
Epoch 3/30
263/263 [==============================] - 3s 10ms/step - loss: 0.5587 - acc
uracy: 0.8307 - val_loss: 0.4496 - val_accuracy: 0.8699
Epoch 4/30
263/263 [==============================] - 2s 9ms/step - loss: 0.5012 - accu
racy: 0.8467 - val_loss: 0.5299 - val_accuracy: 0.8365
Epoch 5/30
263/263 [==============================] - 2s 9ms/step - loss: 0.4546 - accu
racy: 0.8624 - val_loss: 0.3936 - val_accuracy: 0.8899
Epoch 6/30
263/263 [==============================] - 2s 9ms/step - loss: 0.4163 - accu
racy: 0.8713 - val_loss: 0.3844 - val_accuracy: 0.8919
Epoch 7/30
263/263 [==============================] - 3s 10ms/step - loss: 0.3848 - acc
uracy: 0.8822 - val_loss: 0.4160 - val_accuracy: 0.8849
Epoch 8/30
263/263 [==============================] - 3s 10ms/step - loss: 0.3587 - acc
uracy: 0.8882 - val_loss: 0.3522 - val_accuracy: 0.9002
Epoch 9/30
263/263 [==============================] - 2s 9ms/step - loss: 0.3339 - accu
racy: 0.8958 - val_loss: 0.3742 - val_accuracy: 0.8980
Epoch 10/30
263/263 [==============================] - 3s 10ms/step - loss: 0.3221 - acc
uracy: 0.8993 - val_loss: 0.4067 - val_accuracy: 0.8918
Epoch 11/30
263/263 [==============================] - 2s 9ms/step - loss: 0.2965 - accu
racy: 0.9055 - val_loss: 0.4151 - val_accuracy: 0.8927
Epoch 12/30
263/263 [==============================] - 2s 9ms/step - loss: 0.2892 - accu
racy: 0.9093 - val_loss: 0.3647 - val_accuracy: 0.9093
Epoch 13/30
263/263 [==============================] - 2s 9ms/step - loss: 0.2704 - accu
racy: 0.9143 - val_loss: 0.3358 - val_accuracy: 0.9139
Epoch 14/30
263/263 [==============================] - 2s 9ms/step - loss: 0.2625 - accu
racy: 0.9182 - val_loss: 0.4111 - val_accuracy: 0.9050
Epoch 15/30
263/263 [==============================] - 3s 10ms/step - loss: 0.2581 - acc
uracy: 0.9175 - val_loss: 0.4845 - val_accuracy: 0.8824
Epoch 16/30
263/263 [==============================] - 2s 9ms/step - loss: 0.2434 - accu
racy: 0.9232 - val_loss: 0.4082 - val_accuracy: 0.8977
Epoch 17/30
263/263 [==============================] - 2s 9ms/step - loss: 0.2360 - accu
racy: 0.9243 - val_loss: 0.3794 - val_accuracy: 0.8996
Epoch 18/30
263/263 [==============================] - 2s 9ms/step - loss: 0.2249 - accu
racy: 0.9282 - val_loss: 0.3653 - val_accuracy: 0.9104
Epoch 19/30
263/263 [==============================] - 2s 9ms/step - loss: 0.2191 - accu
```

```
racy: 0.9288 - val_loss: 0.4608 - val_accuracy: 0.8995
Epoch 20/30
263/263 [==============================] - 2s 9ms/step - loss: 0.2074 - accu
racy: 0.9323 - val_loss: 0.4097 - val_accuracy: 0.9045
Epoch 21/30
263/263 [==============================] - 2s 9ms/step - loss: 0.2026 - accu
racy: 0.9352 - val_loss: 0.3751 - val_accuracy: 0.9064
Epoch 22/30
263/263 [==============================] - 2s 9ms/step - loss: 0.1884 - accu
racy: 0.9370 - val_loss: 0.4227 - val_accuracy: 0.9096
Epoch 23/30
263/263 [==============================] - 2s 9ms/step - loss: 0.1873 - accu
racy: 0.9382 - val_loss: 0.4186 - val_accuracy: 0.9055
Epoch 24/30
263/263 [==============================] - 3s 10ms/step - loss: 0.1798 - acc
uracy: 0.9402 - val_loss: 0.3876 - val_accuracy: 0.9118
Epoch 25/30
263/263 [==============================] - 2s 9ms/step - loss: 0.1694 - accu
racy: 0.9441 - val_loss: 0.4175 - val_accuracy: 0.9140
Epoch 26/30
263/263 [==============================] - 2s 9ms/step - loss: 0.1784 - accu
racy: 0.9415 - val_loss: 0.4477 - val_accuracy: 0.9048
Epoch 27/30
263/263 [==============================] - 2s 9ms/step - loss: 0.1684 - accu
racy: 0.9442 - val_loss: 0.4627 - val_accuracy: 0.8974
Epoch 28/30
263/263 [==============================] - 3s 10ms/step - loss: 0.1690 - acc
uracy: 0.9445 - val_loss: 0.4774 - val_accuracy: 0.9087
Epoch 29/30
263/263 [==============================] - 2s 9ms/step - loss: 0.1571 - accu
racy: 0.9471 - val_loss: 0.4835 - val_accuracy: 0.9036
Epoch 30/30
263/263 [==============================] - 3s 10ms/step - loss: 0.1601 - acc
uracy: 0.9457 - val_loss: 0.4030 - val_accuracy: 0.9148
```

Plotting the validation and training accuracies

In [ ]:

```python
# 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:

- The second model which has more convolutional layers and less complex in terms of the number of parameters is performing significantly better than the first model.
- The overfitting has reduced significantly. The model is giving a generalized performance.
- As you can see in the graph, validation is almost constant after the epoch 15 which might be due to the plateau region (saddle point or local extremum). We can try adjusting the learning rate during the plateau region.

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.

In [ ]:
```
test_pred = model2.predict(X_test)
test_pred = np.argmax(test_pred, axis=-1)
```

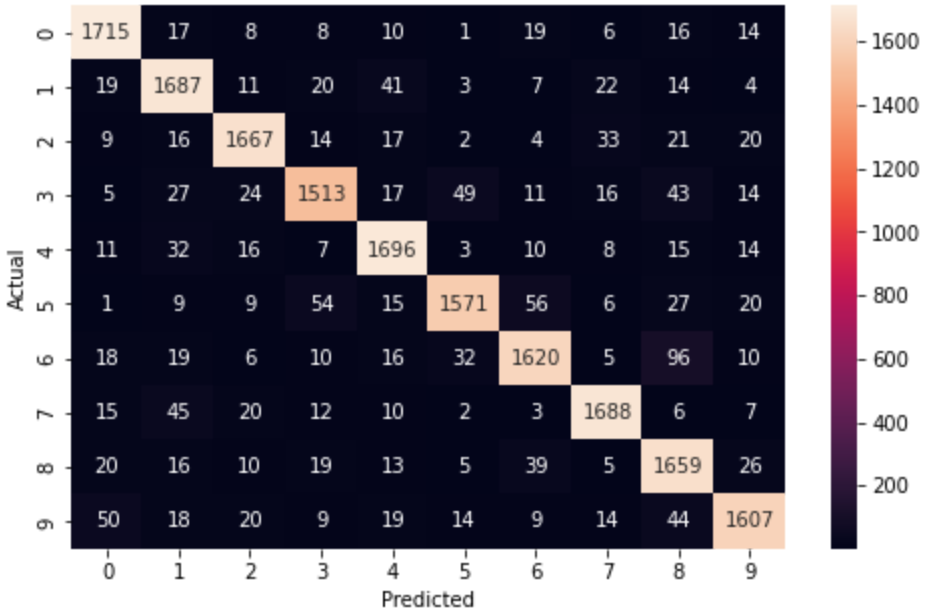
Note: Earlier, we noticed that each entry of the test data is a one-hot encoded vector, but to print the classification report and confusion matrix, we must convert each entry of

y_test to a single label.

In [ ]:
```
# Converting each entry to single label from one-hot encoded vector
y_test = np.argmax(y_test, axis=-1)
```

In [ ]:
```
# 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       0.89      0.92      0.91      1828
           2       0.93      0.92      0.93      1803
           3       0.91      0.88      0.89      1719
           4       0.91      0.94      0.93      1812
           5       0.93      0.89      0.91      1768
           6       0.91      0.88      0.90      1832
           7       0.94      0.93      0.93      1808
           8       0.85      0.92      0.88      1812
           9       0.93      0.89      0.91      1804

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



Observations:

- The accuracy is **91% on the test set.** This is comparable with the results on the validation set which implies that the model is giving a generalized performance.
- This performance is significantly better than the performance of the final model using simple feed-forward neural networks. This suggests that **CNNs are a better choice for this particular dataset.**
- The recall values for all the digits are higher than or equal to 88% with 3 having the least recall. The confusion matrix shows that the model has confused 3 with digits 5 and 8 the most number of times.
- The same can be observed for digits 1 and 8 as well. The model has confused digits 1 and 8 with 4 and 6 respectively.
- The highest recall of about 95% is for digit 0 i.e. the model can identify 95% of images with digit 0.
- **The precision values show that the model is also precise in its prediction, and not just sacrificing precision for recall.** All digits have precision greater than or equal to 85% with 7 having the highest precision of 94%. This indicates that it is easier for the model to identify images with digits 7.
- The class with the least precision is digit 8 and 1.

Note:

- We can try hyperparameter tuning to get an even better performance.
- Data Augmentation might help to make the model more robust and invariant toward different orientations.
- We can also try techniques like transfer learning and see if we can get better results.