Artificial Neural Networks Project: Street View Housing Number Digit Recognition

---

Objective

To build a feed-forward neural network 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

start by mounting the Google drive. Run the below cell to mount the Google drive.

In [ ]:

```
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 [ ]:

```
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
```

check the version of tensorflow.

In [ ]:

```
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 [ ]:

```
import h5py
#/content/SVHN_single_grey1.h5
# Open the file as read only
# User can make changes in the path as required

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

```
# Load the training 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 [ ]:

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

Out[ ]:

(42000, 18000)

Observation:

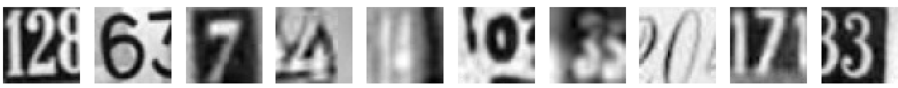
- There are 42,000 images in the training data and 18,000 images in the testing data.

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 printing their 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.
- 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 [ ]:

```
# 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 [ ]:

```
# Reshaping the dataset to flatten them. We are reshaping the 2D image into 1D array

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

In [ ]:

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

X_train = X_train/255

X_test = X_test/255
```

In [ ]:

```
# New shape

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,)
```

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

build an ANN 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

- 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

In [ ]:

```
# Define the model

def nn_model_1():

    model = Sequential()

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

    model.add(Dense(64, activation='relu', input_shape=(1024, )))

    model.add(Dense(32, activation='relu'))

    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 [ ]:

```
# Build the model

model_1 = nn_model_1()
```

In [ ]:

```
# Print the summary

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
_________________________________________________________________
```

In [ ]:

```
# Fit the model

history_model_1 = model_1.fit(X_train, y_train,

          validation_split = 0.2,

          batch_size = 128,

          epochs = 20,

          verbose = 1
)
```

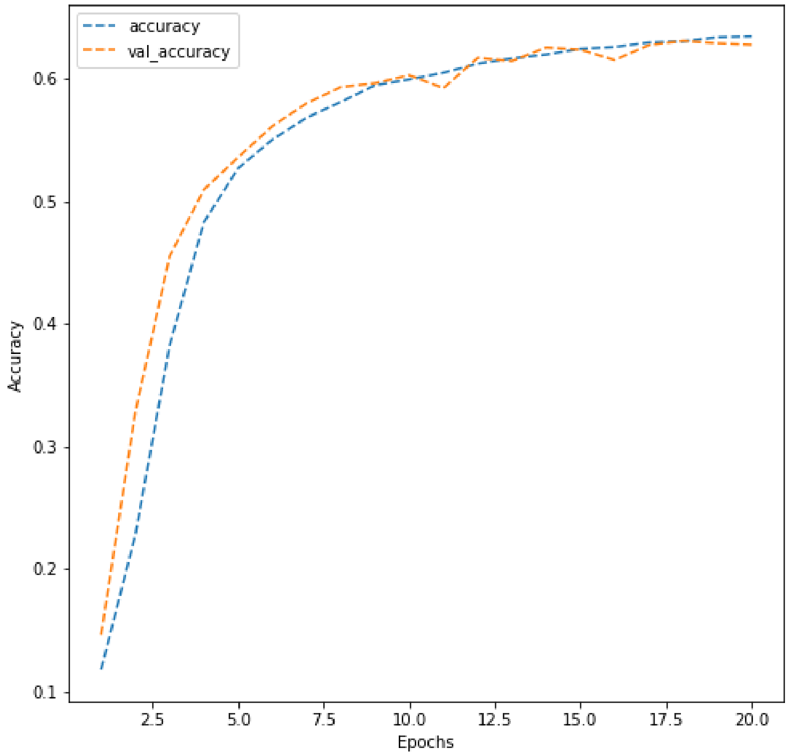
```
Epoch 1/20
263/263 [==============================] - 2s 5ms/step - loss: 2.2993 - accuracy: 0.1179 - val_loss: 2.2594 - val_accuracy: 0.1463
Epoch 2/20
263/263 [==============================] - 1s 5ms/step - loss: 2.1130 - accuracy: 0.2276 - val_loss: 1.9376 - val_accuracy: 0.3279
Epoch 3/20
263/263 [==============================] - 1s 5ms/step - loss: 1.7953 - accuracy: 0.3824 - val_loss: 1.6499 - val_accuracy: 0.4551
Epoch 4/20
263/263 [==============================] - 1s 5ms/step - loss: 1.5726 - accuracy: 0.4828 - val_loss: 1.4957 - val_accuracy: 0.5094
Epoch 5/20
263/263 [==============================] - 2s 6ms/step - loss: 1.4604 - accuracy: 0.5268 - val_loss: 1.4267 - val_accuracy: 0.5357
Epoch 6/20
263/263 [==============================] - 1s 5ms/step - loss: 1.4008 - accuracy: 0.5503 - val_loss: 1.3652 - val_accuracy: 0.5613
Epoch 7/20
263/263 [==============================] - 2s 7ms/step - loss: 1.3588 - accuracy: 0.5682 - val_loss: 1.3282 - val_accuracy: 0.5799
Epoch 8/20
263/263 [==============================] - 2s 6ms/step - loss: 1.3239 - accuracy: 0.5810 - val_loss: 1.2951 - val_accuracy: 0.5931
Epoch 9/20
263/263 [==============================] - 1s 5ms/step - loss: 1.2967 - accuracy: 0.5946 - val_loss: 1.2838 - val_accuracy: 0.5963
Epoch 10/20
263/263 [==============================] - 1s 5ms/step - loss: 1.2779 - accuracy: 0.5994 - val_loss: 1.2624 - val_accuracy: 0.6029
Epoch 11/20
263/263 [==============================] - 1s 5ms/step - loss: 1.2604 - accuracy: 0.6049 - val_loss: 1.2670 - val_accuracy: 0.5923
Epoch 12/20
263/263 [==============================] - 1s 4ms/step - loss: 1.2382 - accuracy: 0.6123 - val_loss: 1.2291 - val_accuracy: 0.6170
Epoch 13/20
263/263 [==============================] - 1s 6ms/step - loss: 1.2287 - accuracy: 0.6166 - val_loss: 1.2327 - val_accuracy: 0.6144
Epoch 14/20
263/263 [==============================] - 1s 5ms/step - loss: 1.2176 - accuracy: 0.6197 - val_loss: 1.1992 - val_accuracy: 0.6255
Epoch 15/20
263/263 [==============================] - 1s 5ms/step - loss: 1.2087 - accuracy: 0.6244 - val_loss: 1.2110 - val_accuracy: 0.6237
Epoch 16/20
263/263 [==============================] - 2s 6ms/step - loss: 1.2027 - accuracy: 0.6259 - val_loss: 1.2247 - val_accuracy: 0.6152
Epoch 17/20
263/263 [==============================] - 1s 5ms/step - loss: 1.1894 - accuracy: 0.6296 - val_loss: 1.1941 - val_accuracy: 0.6274
Epoch 18/20
263/263 [==============================] - 2s 7ms/step - loss: 1.1865 - accuracy: 0.6305 - val_loss: 1.1803 - val_accuracy: 0.6310
Epoch 19/20
263/263 [==============================] - 2s 7ms/step - loss: 1.1783 - accuracy: 0.6338 - val_loss: 1.1853 - val_accuracy: 0.6289
Epoch 20/20
263/263 [==============================] - 2s 6ms/step - loss: 1.1752 - accuracy: 0.6346 - val_loss: 1.1880 - val_accuracy: 0.6277
```

Plotting the validation and training accuracies

observations on the below plot

In [ ]:

```python
# 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 of both training and validation is almost symetric, the model is not overfitting the training dataset.

--Accuracy starts to increase steadily until epoch 6, then it increases in a lower rate until epoch 17 where we observe minimum increase afterward.

--the accuracy of validation dataset is slightly higher than the training dataset until epoch 9, where they start mixing up.

--Overall, the model is not overfit and is giving accuracy of over 0.6 on training dataset and roughly the same on validation dataset.

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.

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

- 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

In [ ]:

```
# Define the model

def nn_model_2():

    model = Sequential()

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

    model.add(Dense(256, activation = 'relu', input_shape = (1024, )))

    model.add(Dense(128, activation = 'relu'))

    model.add(Dropout(0.2))

    model.add(Dense(64, activation = 'relu'))

    model.add(Dense(64, activation = 'relu'))

    model.add(Dense(32, activation = 'relu'))

    model.add(BatchNormalization())

    model.add(Dense(10, activation = 'softmax'))

    # Compile the model

    model.compile(loss = 'categorical_crossentropy',optimizer = tf.keras.optimizers.Adam(learning_rate = 0.0005),metrics=['accuracy'])

    return model
```

In [ ]:

```
# Build the model

model_2 = nn_model_2()
```

In [ ]:

```
# Print the model summary
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
_________________________________________________________________
```

In [ ]:

```
# Fit the model

history_model_2 = model_2.fit(X_train, y_train,

            validation_split = 0.2,

            batch_size = 128,

            epochs = 30,

            verbose = 1
)
```

```
Epoch 1/30
263/263 [==============================] - 4s 13ms/step - loss: 2.3450 - accuracy: 0.1085 - val_loss: 2.2767 - val_accuracy: 0.1340
Epoch 2/30
263/263 [==============================] - 3s 12ms/step - loss: 2.0342 - accuracy: 0.2575 - val_loss: 1.8202 - val_accuracy: 0.3630
Epoch 3/30
263/263 [==============================] - 3s 12ms/step - loss: 1.6066 - accuracy: 0.4447 - val_loss: 1.5031 - val_accuracy: 0.4754
Epoch 4/30
263/263 [==============================] - 3s 12ms/step - loss: 1.4067 - accuracy: 0.5235 - val_loss: 1.2604 - val_accuracy: 0.5808
Epoch 5/30
263/263 [==============================] - 4s 13ms/step - loss: 1.2575 - accuracy: 0.5862 - val_loss: 1.1399 - val_accuracy: 0.6344
Epoch 6/30
263/263 [==============================] - 3s 12ms/step - loss: 1.1867 - accuracy: 0.6154 - val_loss: 1.0533 - val_accuracy: 0.6650
Epoch 7/30
263/263 [==============================] - 3s 12ms/step - loss: 1.1050 - accuracy: 0.6460 - val_loss: 1.0352 - val_accuracy: 0.6686
Epoch 8/30
263/263 [==============================] - 3s 11ms/step - loss: 1.0511 - accuracy: 0.6629 - val_loss: 1.0242 - val_accuracy: 0.6771
Epoch 9/30
263/263 [==============================] - 3s 12ms/step - loss: 1.0305 - accuracy: 0.6695 - val_loss: 0.9965 - val_accuracy: 0.6825
Epoch 10/30
263/263 [==============================] - 4s 14ms/step - loss: 0.9800 - accuracy: 0.6885 - val_loss: 0.9775 - val_accuracy: 0.6895
Epoch 11/30
263/263 [==============================] - 3s 12ms/step - loss: 0.9585 - accuracy: 0.6932 - val_loss: 0.9910 - val_accuracy: 0.6764
Epoch 12/30
263/263 [==============================] - 3s 12ms/step - loss: 0.9316 - accuracy: 0.7015 - val_loss: 0.8793 - val_accuracy: 0.7231
Epoch 13/30
263/263 [==============================] - 3s 12ms/step - loss: 0.9036 - accuracy: 0.7131 - val_loss: 0.8365 - val_accuracy: 0.7401
Epoch 14/30
263/263 [==============================] - 3s 12ms/step - loss: 0.8904 - accuracy: 0.7165 - val_loss: 0.8627 - val_accuracy: 0.7236
Epoch 15/30
263/263 [==============================] - 3s 11ms/step - loss: 0.8834 - accuracy: 0.7207 - val_loss: 0.8662 - val_accuracy: 0.7251
Epoch 16/30
263/263 [==============================] - 3s 11ms/step - loss: 0.8676 - accuracy: 0.7254 - val_loss: 0.8450 - val_accuracy: 0.7302
Epoch 17/30
263/263 [==============================] - 3s 10ms/step - loss: 0.8379 - accuracy: 0.7358 - val_loss: 0.8372 - val_accuracy: 0.7331
Epoch 18/30
263/263 [==============================] - 3s 11ms/step - loss: 0.8385 - accuracy: 0.7326 - val_loss: 0.8039 - val_accuracy: 0.7463
Epoch 19/30
263/263 [==============================] - 3s 11ms/step - loss: 0.8272 - accuracy: 0.7370 - val_loss: 0.8011 - val_accuracy: 0.7479
Epoch 20/30
263/263 [==============================] - 3s 11ms/step - loss: 0.8099 - accuracy: 0.7423 - val_loss: 0.7790 - val_accuracy: 0.7565
Epoch 21/30
263/263 [==============================] - 3s 11ms/step - loss: 0.7917 - accuracy: 0.7484 - val_loss: 0.7915 - val_accuracy: 0.7470
Epoch 22/30
263/263 [==============================] - 3s 10ms/step - loss: 0.7988 - accuracy: 0.7477 - val_loss: 0.8588 - val_accuracy: 0.7273
Epoch 23/30
263/263 [==============================] - 3s 11ms/step - loss: 0.7755 - accuracy: 0.7541 - val_loss: 0.7969 - val_accuracy: 0.7508
Epoch 24/30
263/263 [==============================] - 3s 11ms/step - loss: 0.7655 - accuracy: 0.7546 - val_loss: 0.7608 - val_accuracy: 0.7623
Epoch 25/30
263/263 [==============================] - 3s 11ms/step - loss: 0.7594 - accuracy: 0.7567 - val_loss: 0.7546 - val_accuracy: 0.7596
Epoch 26/30
263/263 [==============================] - 3s 12ms/step - loss: 0.7590 - accuracy: 0.7590 - val_loss: 0.7774 - val_accuracy: 0.7554
Epoch 27/30
263/263 [==============================] - 3s 12ms/step - loss: 0.7366 - accuracy: 0.7651 - val_loss: 0.7772 - val_accuracy: 0.7546
Epoch 28/30
263/263 [==============================] - 3s 12ms/step - loss: 0.7340 - accuracy: 0.7656 - val_loss: 0.7828 - val_accuracy: 0.7510
Epoch 29/30
263/263 [==============================] - 3s 11ms/step - loss: 0.7204 - accuracy: 0.7709 - val_loss: 0.7290 - val_accuracy: 0.7771
Epoch 30/30
263/263 [==============================] - 3s 11ms/step - loss: 0.7138 - accuracy: 0.7734 - val_loss: 0.7256 - val_accuracy: 0.7773
```

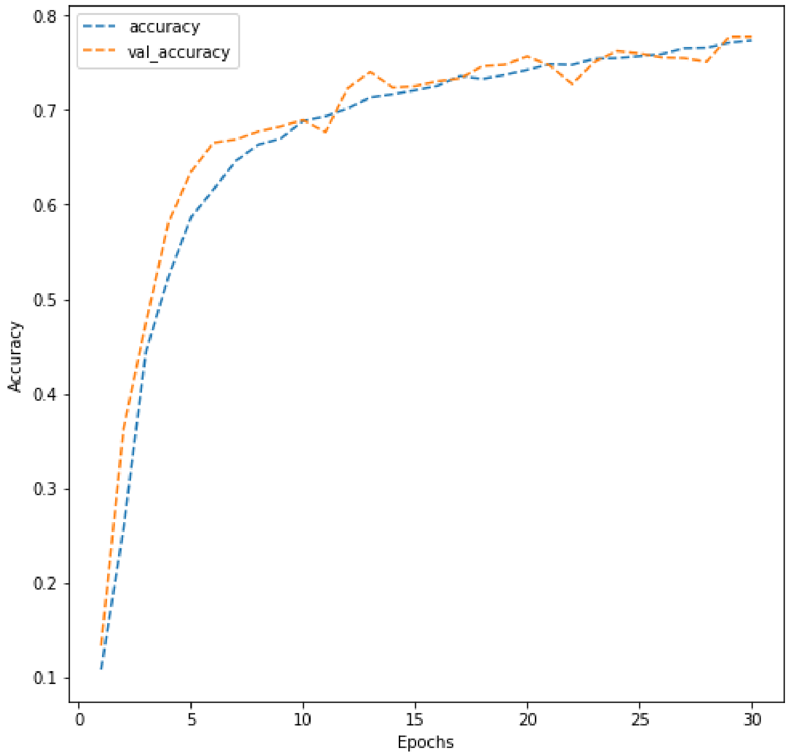
Plotting the validation and training accuracies

observations on the below plot

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:___

--both training dataset and validation dataset have symetric accuracy to some level and there is no overfitting.

--Accuracy is growing rapidly as the model increases with epochs to 8, then grows slowly.

--After epoch 11 we some mixup between training and validation accuracy

--this model is better than the previous with accuracy of 0.77 compared to 0.70.

Predictions on the test data

- 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 = model_2.predict(X_test)

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

Note: 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 [ ]:

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

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

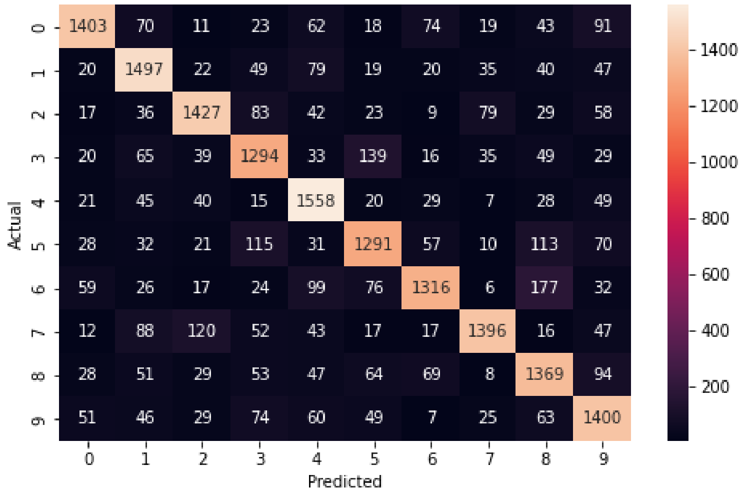
Print the classification report and the confusion matrix for the test predictions.

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       0.85      0.77      0.81      1814
       1       0.77      0.82      0.79      1828
       2       0.81      0.79      0.80      1803
       3       0.73      0.75      0.74      1719
       4       0.76      0.86      0.81      1812
       5       0.75      0.73      0.74      1768
       6       0.82      0.72      0.76      1832
       7       0.86      0.77      0.81      1808
       8       0.71      0.76      0.73      1812
       9       0.73      0.78      0.75      1804

accuracy                           0.78     18000
macro avg       0.78      0.77      0.78     18000
weighted avg    0.78      0.78      0.78     18000
```



Final Observations:__

---The report shows numbers 0,4 and 7 have the highest f1-score (0.81) meaning they have the best chances of being accurately recognized. Whereas, number 8 have the lowest f1-score of (0.73).

---Number 8 has the lowest precision and 7 has the highest.

---Number 4 has the highest recall, whereas 6 has the lowest. It indicates that the model is struggling to identify all 6's as what they are. Opposed to 4, which the model idenifies in high rates of completion.

---The confusion matrix shows that the model confused 5 with 3 and 6 with 8.

In [ ]: