Task 3

Character Recognition

Data Description

For recognizing handwritten forms, the very first step was to gather data in a considerable amount for training.

The dataset contains 26 folders (A-Z) containing handwritten images in size 28x28 pixels, each alphabet in the image is center fitted to 20x20 pixel box. Each image is stored as Gray-level.

The CSV file where each row represents an image, and columns represent pixel values and labels.

The first column could be the label (A-Z), and the remaining columns represent pixel values.

1.Preprocessing step

1.1 Handle Numeric Values to refer to alphabets

```
word_dictionary={0:'A',1:'B',2:'C',3:'D',4:'E',5:'F',6:'G',7:'H',8:'I',9:'J',10:'
K',11:'L',12:'M',13:'N',14:'O',15:'P',16:'Q',17:'R',18:'S',19:'T',20:'U',21:'V',2
2:'W',23:'X',24:'Y',25:'Z'}


train_data[0] = train_data[0].map(word_dictionary)

label_size = train_data.groupby(0).size()
label_size.plot.barh(figsize=(10,10))
plt.show()
```

Make a word dictionary , then convert the numeric values in the first column of train_data into their corresponding alphabetical letters based on the word_dictionary. It's a way to represent labels or categories with letters instead of numbers.

2. Split the data into 80% training and 20 % testing

3. Standard Scaling to features & Reshaping

```
x_train, x_test, y_train, y_test = train_test_split(X, Y, test_size=0.2,
random_state=42)
# scale data

standard_scaler = MinMaxScaler()
standard_scaler.fit(x_train)

x_train = standard_scaler.transform(x_train)
x_test = standard_scaler.transform(x_test)

x_train = x_train.reshape(x_train.shape[0], 28, 28, 1).astype('float32')
x_test = x_test.reshape(x_test.shape[0], 28, 28, 1).astype('float32')

y_train = to_categorical(y_train,num_classes=26,dtype='int')
y_test = to_categorical(y_test,num_classes=26,dtype='int')
```

MinMaxScaler is used to scale the input features to a specific range (usually between 0 and 1). It's important for neural networks to work with consistent and normalized input data.

The data is reshaped into a 4D array to match the input shape expected by convolutional neural networks (assuming it's image data). The shape (28, 28, 1) suggests grayscale images of size 28x28 pixels.

to_categorical is used to convert the class labels into one-hot encoded vectors. This is a common practice in classification tasks, especially when dealing with neural networks. The num_classes parameter specifies the number of classes (26 in this case, assuming there are 26 categories). The dtype parameter ensures that the labels are encoded as integers.

4. Building the model

Neural Network consists of the following:

1. Convolutional Layers (Conv2D):
    - Three sets of convolutional layers with increasing filter sizes (32, 64, 128), each followed by ReLU activation.
2. MaxPooling Layers (MaxPool2D):
    - MaxPooling layers with a pool size of (2, 2) and a stride of 2, which reduces the spatial dimensions of the feature maps.
3. Dropout Layers (Dropout):
    - Dropout layers with a dropout rate of 0.5, applied after each MaxPooling layer. Dropout helps prevent overfitting by randomly dropping a fraction of neurons during training.
4. Flatten Layer:
    - Flatten layer to convert the 2D feature maps to a 1D vector before passing to Dense layers.
5. Dense Layers (Dense):
    - A dense hidden layer with 64 neurons and ReLU activation.
6. Output Layer:
    - Output layer with 26 neurons (assuming it's a classification task with 26 classes) and softmax activation for multi-class classification.

```
from keras.models import Sequential
from keras.layers import Conv2D,Dense,Flatten,MaxPool2D,Dropout

model=Sequential()
model.add(Conv2D(filters=32,kernel_size=(3,3),activation='relu',input_shape=(28,2
8,1)))
model.add(MaxPool2D(pool_size=(2,2),strides=2))

model.add(Dropout(0.5))
model.add(Conv2D(filters=64,kernel_size=(3,3),activation='relu',input_shape=(28,2
8,1)))
model.add(MaxPool2D(pool_size=(2,2),strides=2))

model.add(Dropout(0.5))

model.add(Conv2D(filters=128,kernel_size=(3,3),activation='relu',input_shape=(28,
28,1)))
model.add(MaxPool2D(pool_size=(2,2),strides=2))

model.add(Flatten())
model.add(Dense(64,activation='relu'))
model.add(Dense(26,activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])
history = model.fit(x_train, y_train, validation_data=(x_test, y_test),
epochs=20,)

scores = model.evaluate(x_test,y_test, verbose=0)
print("CNN Score:",scores[1])
```

After building the model , it undergoes some steps :

1. Compilation:
    - The model is compiled with categorical crossentropy loss, the Adam optimizer, and accuracy as the evaluation metric.
2. Training:
    - The model is trained on the training data for 20 epochs, with validation data specified for monitoring the model's performance on unseen data.
3. Evaluation:
    - The model is evaluated on the test data, and the accuracy score is printed.

The accuracy score is 0.98 which is a very outstanding result.

4. Extending the model to recognize words

4.1 Change Dataset

Here we can change the dataset to hold word images
like iam_handwriting_word_database , it would very efficient in word recognition

Description: The database contains forms of unconstrained handwritten text, which were scanned at a resolution of 300dpi and saved as PNG images with 256 gray levels. The figure below provides samples of a complete form, a text line and some extracted words.

Characteristics

The IAM Handwriting Database 3.0 is structured as follows:

657 writers contributed samples of their handwriting
1'539 pages of scanned text
5'685 isolated and labeled sentences
13'353 isolated and labeled text lines
115'320 isolated and labeled words

The words have been extracted from pages of scanned text using an automatic segmentation scheme and were verified manually. The segmentation scheme has been developed at our institute

Preprocessing steps

1. Loop through lines in the dataset: Iterate through each line in the dataset, keeping track of the index.
2. Split the line: Split the current line into a list of substrings based on spaces to extract different pieces of information.
3. Extract status information: Extract the status information (whether the word is labeled as 'ok' or not) from the second element of the split line.
4. Process 'ok' status: Check if the status is 'ok'. If true, proceed to process the word.
5. Extract word information: Extract the word ID and reconstruct the word by joining elements from the split line, starting from the 8th element.
6. Construct filepath: Construct the file path for the image using the word ID and its components.
7. Process image: Read the image from the constructed file path, process it (possibly resizing), and store it.
8. Process label: Encode the word label into a list of numerical values.
9. Split data for training and validation: Split the data into training and validation sets, storing images, labels, input lengths, label lengths, and original text separately.
10. Track maximum label length: Keep track of the maximum label length encountered during processing.
11. Limit processing to a specified number of records: Break the loop if the specified number of records has been processed.
12. Using pad_sequences adds padding (len(char_list)) to the end ('post' padding) of each sequence to make them uniform in length (max_label_len).It ensures that all sequences have the same length as the longest one in the dataset. value=len(char_list) is used as the padding value, assuming it represents a special token for padding in your label encoding scheme

```
train_padded_label = pad_sequences(train_labels,
                                   maxlen=max_label_len,
                                   padding='post',
                                   value=len(char_list))

valid_padded_label = pad_sequences(valid_labels,
                                   maxlen=max_label_len,
                                   padding='post',
                                   value=len(char_list))
```

13. The NumPy arrays are used as inputs (x) and labels (y) for training a sequence-to-sequence model. The model is likely designed to handle variable-length sequences, where $train\_input\_length$ and $valid\_input\_length$ specify the lengths of input sequences, and $train\_label\_length$ and $valid\_label\_length$ specify the lengths of corresponding label.

```
train_images = np.asarray(train_images)
train_input_length = np.asarray(train_input_length)
train_label_length = np.asarray(train_label_length)

valid_images = np.asarray(valid_images)
valid_input_length = np.asarray(valid_input_length)
valid_label_length = np.asarray(valid_label_length)
```

Overall, this process involves reading and organizing data from the IAM Handwriting Database for word recognition. It checks the status of each word, processes valid words, and splits the data into training and validation sets while tracking important information such as image paths, labels, and lengths.

4.1 The neural network can be like this

```
from keras.layers import Input

# input with shape of height=32 and width=128
inputs = Input(shape=(32,128,1))

# convolution layer with kernel size (3,3)
conv_1 = Conv2D(64, (3,3), activation = 'relu', padding='same')(inputs)
# poolig layer with kernel size (2,2)
pool_1 = MaxPool2D(pool_size=(2, 2), strides=2)(conv_1)

conv_2 = Conv2D(128, (3,3), activation = 'relu', padding='same')(pool_1)
pool_2 = MaxPool2D(pool_size=(2, 2), strides=2)(conv_2)

conv_3 = Conv2D(256, (3,3), activation = 'relu', padding='same')(pool_2)

conv_4 = Conv2D(256, (3,3), activation = 'relu', padding='same')(conv_3)
# poolig layer with kernel size (2,1)
pool_4 = MaxPool2D(pool_size=(2, 1))(conv_4)

conv_5 = Conv2D(512, (3,3), activation = 'relu', padding='same')(pool_4)
# Batch normalization layer
batch_norm_5 = BatchNormalization()(conv_5)

conv_6 = Conv2D(512, (3,3), activation = 'relu', padding='same')(batch_norm_5)
batch_norm_6 = BatchNormalization()(conv_6)
pool_6 = MaxPool2D(pool_size=(2, 1))(batch_norm_6)

conv_7 = Conv2D(512, (2,2), activation = 'relu')(pool_6)

squeezed = Lambda(lambda x: K.squeeze(x, 1))(conv_7)

# bidirectional LSTM layers with units=128
blstm_1 = Bidirectional(LSTM(256, return_sequences=True, dropout =
0.2))(squeezed)
blstm_2 = Bidirectional(LSTM(256, return_sequences=True, dropout = 0.2))(blstm_1)

outputs = Dense(len(char_list)+1, activation = 'softmax')(blstm_2)
# model to be used at test time
act_model = Model(inputs, outputs)
```

In the context of detecting words in an image:

- The convolutional layers can learn to recognize visual patterns and structures in the image that might correspond to characters or parts of words.
- The LSTM layers are used to capture sequential information, which can be helpful when the task involves recognizing words or sequences of characters. In the case of word detection, the LSTM layers could learn to understand the spatial arrangement of characters that form words.

```
the_labels = Input(name='the_labels', shape=[max_label_len], dtype='float32')
input_length = Input(name='input_length', shape=[1], dtype='int64')
label_length = Input(name='label_length', shape=[1], dtype='int64')

def ctc_lambda_func(args):
    y_pred, labels, input_length, label_length = args

    return K.ctc_batch_cost(labels, y_pred, input_length, label_length)

loss_out = Lambda(ctc_lambda_func, output_shape=(1,), name='ctc')([outputs,
the_labels, input_length, label_length])

#model to be used at training time
model = Model(inputs=[inputs, the_labels, input_length, label_length],
outputs=loss_out)
```

Input Layers:

- the_labels: Represents the true labels for training, with a shape of [max_label_len].
- input_length: Represents the length of the input sequence (e.g., the length of the spectrogram). It has a shape of [1].
- label_length: Represents the length of the true label sequence. It also has a shape of [1].

CTC Lambda Function:

- ctc_lambda_func: Defines a lambda function that takes four arguments: y_pred (predicted output from the model), labels (true labels), input_length, and label_length. Inside the lambda function, it uses Keras backend functions to calculate the CTC batch cost, which measures the difference between predicted and true labels given the input sequence lengths.
- Lambda: Wraps the lambda function and creates a Keras layer. This layer will be used to compute the CTC loss during training.

CNN-LSTM Score: 0.628

4.3 Second Approach Using OCR techniques like Tesseract library

```
from PIL import Image
import cv2
import pytesseract

# Configure the path to the Tesseract executable (replace
'your_path_to_tesseract' with the actual path)
pytesseract.pytesseract.tesseract_cmd = r'C:\Program Files\Tesseract-
OCR\tesseract.exe'

def preprocess_test_data(image_path):
    # Load the image
    img = Image.open(image_path).convert('L')  # Convert to grayscale
    img = img.resize((28, 28))  # Resize to match model input size

    # Convert to numpy array and normalize pixel values
    img_array = np.array(img) / 255.0

    # Reshape the data to match the model input shape
    img_array = img_array.reshape(1, 28, 28, 1)

    return img_array

# Load the word image
word_image_path =
'D:\\DeepLearning\\CodeAlpha_Project_Name\\HandWrittenCharacterRecognition\\Datas
et\\test2.png'
word_image = cv2.imread(word_image_path)

# Convert the image to grayscale
gray = cv2.cvtColor(word_image, cv2.COLOR_BGR2GRAY)

# Use pytesseract to perform OCR and extract text from the image
extracted_text = pytesseract.image_to_string(gray)

# Preprocess the extracted text (adjust as needed)
preprocessed_text = extracted_text.upper().strip()

# Now, preprocessed_text contains the recognized text from the image
print("Recognized Text:", preprocessed_text)
```

Explanation:

1. Tesseract OCR Configuration:
    - The Tesseract OCR engine is configured with the path to the Tesseract executable (tesseract_cmd). This is necessary for pytesseract to interact with the Tesseract OCR engine.
2. Image Preprocessing Function (preprocess_test_data):
    - The function takes an image path as input, loads the image, converts it to grayscale, resizes it to match the expected input size of a model (28x28 pixels), and then converts it to a numpy array.
    - The pixel values are normalized to the range [0, 1].
    - The data is reshaped to match the input shape expected by a model (assuming it's a neural network for image classification).
3. Loading and Preprocessing the Word Image:
    - The word image is loaded using OpenCV (cv2.imread) and converted to grayscale.
    - The preprocessed data is then passed to the Tesseract OCR engine using pytesseract.image_to_string.
4. Post-processing the Recognized Text:
    - The extracted text from the OCR engine is stored in the variable extracted_text.
    - The upper() method is used to convert the text to uppercase, and strip() is used to remove leading and trailing whitespaces.
    - The preprocessed text is stored in the variable preprocessed_text.
5. Printing the Recognized Text:
    - The final recognized and preprocessed text is printed.