Handwriting Recognition

March 10, 2023

1 IMPORT THE ESSENTIAL LIBRARIES

[1]:
```
# Import the essential libraries
import numpy as np
import cv2
import os
import pandas as pd
import string
import matplotlib.pyplot as plt
import sys, tarfile
```

This is a Python code that imports essential libraries for working with image data, file handling, data manipulation, and visualization.

numpy is a Python package for scientific computing that provides support for array and matrix operations.

cv2 is a Python package for computer vision tasks and provides image processing functionalities.

os is a Python module that provides a way of using operating system dependent functionality. It is used here for file handling.

pandas is a Python package used for data manipulation and analysis. It provides data structures for efficient handling of large datasets.

string is a Python module that contains various string constants and templates. It is used here for string manipulation.

matplotlib is a Python package used for data visualization. It provides functions for creating various types of plots, graphs, and charts.

sys is a Python module that provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter.

tarfile is a Python module that provides an interface for reading and writing tar archives. It is used here for file handling.

Overall, this code imports the libraries necessary for performing various image processing tasks, file handling, and data manipulation, and visualization.

[2]:
```
import tensorflow as tf
```

```
#ignore warnings in the output
tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.ERROR)
import tensorflow.keras.backend as K

from tensorflow import keras
from tensorflow.keras import layers

from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.layers import Dense, Reshape, BatchNormalization, Input,␣
 ↪Conv2D, MaxPool2D, Lambda, Bidirectional
from tensorflow.compat.v1.keras.layers import LSTM
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import *
from tensorflow.keras.utils import to_categorical, Sequence
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping

from tqdm import tqdm
from collections import Counter
from PIL import Image
from itertools import groupby
from keras.activations import relu, sigmoid, softmax

from sklearn.model_selection import train_test_split
```

import tensorflow as tf imports the TensorFlow library and assigns it the alias "tf".

tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.ERROR) is used to ignore any warning messages that may be generated during the model training process.

import tensorflow.keras.backend as K imports the backend module of the Keras API from TensorFlow, which is used for building deep learning models.

from tensorflow import keras imports the high-level Keras API from TensorFlow.

from tensorflow.keras import layers imports the layers module from the Keras API, which provides various types of neural network layers.

from tensorflow.keras.preprocessing.sequence import pad_sequences imports the pad_sequences function from the preprocessing module, which is used to pad sequences to a fixed length.

from tensorflow.keras.layers import Dense, Reshape, BatchNormalization, Input, Conv2D, MaxPool2D, Lambda, Bidirectional imports various types of Keras layers that will be used in the model.

from tensorflow.compat.v1.keras.layers import LSTM imports the LSTM layer from the Keras API, which is used for building recurrent neural networks.

from tensorflow.keras.models import Model imports the Model class from the Keras API, which is used for defining and training deep learning models.

from tensorflow.keras.optimizers import * imports various types of optimization algorithms that can be used to train the model.

from tensorflow.keras.utils import to_categorical, Sequence imports utility functions from the Keras API that can be used for data preparation and processing.

from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping imports callback functions from the Keras API that can be used to monitor the training process and save the best model weights.

from tqdm import tqdm imports the tqdm library, which is used to display progress bars during the training process.

from collections import Counter imports the Counter class from the collections module, which is used to count the frequency of elements in a list.

from PIL import Image imports the Image module from the Pillow library, which is used for image processing.

from itertools import groupby imports the groupby function from the itertools module, which is used for grouping elements in a list.

from keras.activations import relu, sigmoid, softmax imports various activation functions that can be used in the model.

from sklearn.model_selection import train_test_split imports the train_test_split function from the scikit-learn library, which is used for splitting the dataset into training and validation sets.

DOWNLOAD THE IAM DATASET

Url: https://fki.tic.heia-fr.ch/databases/iam-handwriting-database

IAM dataset category: data/words.tgz

Version: 3

2 CONFIGURING THE ENVIRONMENT FOR DATASET

[5]:
```
# To check if a folder is already existed or not. If not, then create one.
def make_folder(path):
    if not os.path.exists(path):
        os.mkdir(path)
```

This code defines a function called make_folder that takes a single argument called path. The function is used to check if a folder already exists at the specified path. If the folder does not exist, the function creates a new folder at the specified path.

The function uses the os module, which is a part of the Python standard library, to check if the folder exists. The os.path.exists function is used to check if a path exists or not. If the path exists, the function returns True, otherwise it returns False.

The if statement in the code checks the return value of os.path.exists(path). If the path does not exist (not os.path.exists(path)), the os.mkdir(path) function is called to create a new folder at the specified path. The os.mkdir function creates a new directory with the specified path.

Overall, this code can be used to create a folder at a specified path if it does not already exist.

[6]:
```
# Create a folder with the name "datasets" where the image dataset will be␣
↪saved.
make_folder("datasets")
```

This code calls the make_folder function that was defined previously to create a new folder called "datasets" in the current working directory where the image dataset will be saved.

The function make_folder checks if a folder already exists at the specified path, which in this case is "datasets". If the folder does not exist, it creates a new folder with the specified name.

Overall, this code ensures that a folder with the name "datasets" exists in the current working directory where the image dataset will be saved. If the folder already exists, the make_folder function will not create a new one.

[7]:
```
# Extract the images from the zip file and save the images into the dataset␣
↪folder.
def extract(tar_url, extract_path='/content/datasets/'):
    print (tar_url)
    tar = tarfile.open(tar_url, 'r')
    for item in tar:
        tar.extract(item, extract_path)
        if item.name.find(".tgz") != -1 or item.name.find(".tar") != -1:
            extract(item.name, "./" + item.name[:item.name.rfind('/')])
try:
    extract('/content/words.tgz')
    print ('Done.')
except:
    name = os.path.basename('/content/words.tgz')
    print (name[:name.rfind('.')], '<filename>')
```

```
/content/words.tgz
Done.
```

This code defines a function called extract that takes two arguments, tar_url and extract_path. The function is used to extract images from a tar file and save them into the specified extract_path directory.

The function first prints the tar_url to the console for debugging purposes.

The tarfile module is imported and used to open the tar file specified in tar_url using the tarfile.open function. The r parameter is used to indicate that the tar file is being opened for reading.

A for loop is used to iterate through all items (files and directories) in the tar file using the tar object. The tar.extract method is called for each item to extract and save it to the extract_path directory.

The if statement inside the for loop checks if the item is a .tgz or .tar file by checking if its name contains the substring ".tgz" or ".tar". If the item is a .tgz or .tar file, the extract function is called recursively to extract its contents into a subdirectory with the same name as the file.

The try and except blocks are used to handle any errors that may occur during the extraction process. If an error occurs, the basename function from the os.path module is used to extract the

name of the tar file, and the error message includes the filename and an indication that an error occurred.

Overall, this code is used to extract images from a tar file and save them into the specified directory. It uses the tarfile module to extract the contents of the tar file and calls itself recursively to extract any subdirectories that contain additional .tgz or .tar files.

3 PREPROCESSING

[8]:
```
# Open and read the parser.txt file
with open('/content/parser.txt') as f:
    contents = f.readlines()

lines = [line.strip() for line in contents]

# display the first element of parser.txt
lines[0]
```

[8]:
```
'a01-000u-00-00 ok 154 408 768 27 51 AT A'
```

This code reads the contents of a file called parser.txt located at the specified path (/content/parser.txt).

This file contains annotations for each image. The **AT A'** is the label of the image.

The with statement is used to open the file and create a file object f. The with statement automatically closes the file object f when the block is exited.

The readlines() method is used to read the contents of the file object f. The readlines() method returns a list of all the lines in the file.

The for loop and list comprehension are used to remove any leading or trailing whitespace from each line in the contents list. The resulting list is assigned to a new list called lines.

Finally, the first element of the lines list is accessed and printed to the console.

Overall, this code reads the contents of a file called parser.txt, removes any leading or trailing whitespace from each line, and prints the first line to the console.

[9]:
```
# max label length, that is, the length of the word string. if string is "two",␣
 ↪the
# max_label_length will be 3
max_label_len = 0

char_list = "!\"#&'()*+,-./0123456789:;?
 ↪ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"

print(char_list, len(char_list))


# encoding each output word into the digits
```

```
def encode_to_labels(txt):
    dig_lst = []
    for index, char in enumerate(txt):
        dig_lst.append(char_list.index(char))

    return dig_lst
```

```
!"#&'()*+,-./0123456789:;?ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
78
```

This code defines a variable called max_label_len and initializes it to zero. This variable will be used to store the length of the longest label (word string).

The code also defines a string called char_list that contains all possible characters that may appear in the labels. The len() function is used to determine the length of the char_list string, which is printed to the console.

The code also defines a function called encode_to_labels that takes a string (txt) as input and returns a list of digits representing the encoded label.

Inside the function, a new empty list called dig_lst is defined. The enumerate() function is used to iterate through each character in the txt string along with its index. For each character, its index in the char_list string is determined using the char_list.index(char) method and appended to the dig_lst list.

Finally, the dig_lst list is returned by the function, containing the encoded label in the form of a list of digits.

Overall, this code defines a character list and a function that encodes a label (word string) into a list of digits using the index of each character in the character list. It also initializes a variable to store the length of the longest label.

[10]:
```
# Initilizing the images and lables list
images = []
labels = []

# Number of images to work with.
RECORDS_COUNT = 10000
```

This code initializes two empty lists: images and labels.

The images list will be used to store the image data, while the labels list will be used to store the corresponding labels (word strings) for each image.

The code also defines a constant called RECORDS_COUNT and sets its value to 10000. This constant will be used to specify the number of images to work with.

[11]:
```
# Initilizing the required train images lists
train_images = []
train_labels = []
train_input_length = []
train_label_length = []
```

```
train_original_text = []

# Initilizing the required train images lists
valid_images = []
valid_labels = []
valid_input_length = []
valid_label_length = []
valid_original_text = []

# length of inputs and the labels words
inputs_length = []
labels_length = []
```

This code initializes several empty lists that will be used to store data during training and validation of a machine learning model.

For the training set, the code initializes five lists:

train_images: will store the training set images data.

train_labels: will store the corresponding training set labels (word strings) for each image.

train_input_length: will store the length of each input image (in terms of time steps).

train_label_length: will store the length of each label (word string) for each image.

train_original_text: will store the original text of each label (word string) for each image.

Similarly, for the validation set, the code initializes another five empty lists:

valid_images: will store the validation set images data.

valid_labels: will store the corresponding validation set labels (word strings) for each image.

valid_input_length: will store the length of each input image (in terms of time steps).

valid_label_length: will store the length of each label (word string) for each image.

valid_original_text: will store the original text of each label (word string) for each image.

Finally, the code initializes two more empty lists:

inputs_length: will store the length of each input image for both training and validation sets.

labels_length: will store the length of each label (word string) for both training and validation sets.

[12]:
```
def process_image(img):
    """
    Converts image to shape (32, 128, 1) & normalize.
    W refers to the width and h refers to the height
    """
    w, h = img.shape

    new_w = 32
```

```
    new_h = int(h * (new_w / w))

    # resize the image
    img = cv2.resize(img, (new_h, new_w))
    w, h = img.shape

    img = img.astype('float32')

    # Converts each to (32, 128, 1)
    if w < 32:
        add_zeros = np.full((32-w, h), 255)
        img = np.concatenate((img, add_zeros))
        w, h = img.shape

    if h < 128:
        add_zeros = np.full((w, 128-h), 255)
        img = np.concatenate((img, add_zeros), axis=1)
        w, h = img.shape

    if h > 128 or w > 32:
        dim = (128,32)
        img = cv2.resize(img, dim)

    img = cv2.subtract(255, img)

    img = np.expand_dims(img, axis=2)

    # Normalize the image between 0 to 1
    img = img / 255

    return img
```

This is a Python function named process_image that takes an image as input and returns a processed image. The function performs the following operations:

The dimensions (width and height) of the input image are obtained using img.shape.

The image is resized to have a width of 32 pixels while maintaining its aspect ratio by calculating the new height using new_h = int(h * (new_w / w)) and calling cv2.resize(img, (new_h, new_w)).

If the resized image has dimensions smaller than (32,128), zeros are added to the image to make it (32,128) using np.full() and np.concatenate().

If the resized image has dimensions larger than (32,128), the image is resized to (32,128) using cv2.resize().

The image is then inverted (subtracting the pixel values from 255) using cv2.subtract(255, img).

The image is then expanded to have an additional dimension using np.expand_dims(img, axis=2).

Finally, the image is normalized by dividing all pixel values by 255 to scale them between 0 and 1.

The returned image has dimensions (32,128,1) and has been normalized for use in deep learning models.

[13]:
```
# Read the image dataset
# lines is the list of all the information derived from the parser.txt file.

# images having ok text, instead of err, in the parser.txt file
ok_image_counter = 0

for index, line in enumerate(lines):
    splits = line.split(' ')

    # Get the ok/error text mentioned in the parserfile that is assigned to␣
 ↪each image
    status = splits[1]

    # if the text is ok then,
    if status == 'ok':

        # get the word
        word_id = splits[0]

        # Get the word
        word = "".join(splits[8:])

        splits_id = word_id.split('-')

        # location of image datasets
        filepath = '/content/datasets/{}/{}-{}/{}.png'.format(splits_id[0],
                                                splits_id[0],
                                                splits_id[1],
                                                word_id)

        # Convert the image into grayscale
        img = cv2.imread(filepath, cv2.IMREAD_GRAYSCALE)

        # process the image and check if the image is corrupted or not.
        try:
            img = process_image(img)
        except:
            # if corrupted the try block will show the error and the next loop␣
 ↪will iterate,
            # and the present corrupted image will not be saved into the list.
            continue

        # process label
        try:
```

```
            # Get the labels
            label = encode_to_labels(word)
        except:
            continue

        # if the images are not corrupted and have ok test in the parser.txt␣
↪file
        ok_image_counter += 1

        # append the uncorrupted images and the labels into the train_images␣
↪list.
        train_images.append(img)
        train_labels.append(label)
        train_original_text.append(word)

        # checking if the label is maximum or not so that we can use it while␣
↪training the model.
        if len(word) > max_label_len:
            max_label_len = len(word)

    # checking if we have surpassed the required number of training instances,␣
↪that is 10000, or not.
    if ok_image_counter >= RECORDS_COUNT:
        break
```

This code reads image datasets from the file system and processes them for use in training a machine learning model. Here's what it does:

The variable lines contains a list of strings, each of which represents information about an image that was parsed from a file. The for loop iterates over each line in lines, and for each line, extracts the status (either "ok" or "err") of the image. If the status is "ok", the code extracts the word ID, word text, and file path for the corresponding image.

The image is loaded using OpenCV's cv2.imread() function, and then passed through a process_image() function for preprocessing. If an exception occurs during preprocessing (i.e. the image is corrupted), the loop continues to the next image.

The word text is then converted to a label using an encode_to_labels() function. If an exception occurs during this process, the loop continues to the next image.

If the image is not corrupted and has an "ok" status, the image, label, and original word text are appended to train_images, train_labels, and train_original_text lists, respectively. The variable ok_image_counter is incremented by 1.

If the number of images with "ok" status reaches the value of RECORDS_COUNT, which is set to 10000, the loop is broken. The code keeps track of the maximum length of the labels in the max_label_len variable. This will be used later during the training of the machine learning model.

Overall, this code reads and processes image data, converts the word text to a label, and stores the preprocessed data for training a machine learning model.

[14]:
```
# split the dataset into training and testing.
train_images, valid_images, train_labels, valid_labels=␣
 ↪train_test_split(train_images, train_labels, test_size = 0.1, random_state =␣
 ↪42)
```

This code splits a dataset into training and testing sets using the train_test_split() function from the Scikit-Learn library. Here's what each line does:

train_images and train_labels are the lists containing the preprocessed image data and labels, respectively, that were created in the previous code.

train_test_split() takes four arguments: the data to be split, the corresponding labels, the proportion of the data to use for testing (in this case, 10%), and a random seed value for reproducibility.

The function returns four sets of data: train_images and train_labels, which contain 90% of the data and will be used for training the machine learning model, and valid_images and valid_labels, which contain the remaining 10% and will be used for testing the model's accuracy.

The code assigns the output of train_test_split() to the four variables on the left-hand side of the equals sign, which updates their values accordingly.

Overall, this code splits the preprocessed image data and labels into separate training and testing sets, which is a common step in machine learning workflows to evaluate the performance of the trained model on new, unseen data.

[15]:
```
# Get the train_label_length and train_input_length of the train data.
# Here we have seleted the train_input_length to be 31
for i in range(len(train_labels)):
  train_label_length.append(len(train_labels[i]))
  train_input_length.append(31)
```

This code computes the length of each label in the training set and sets the input length to a fixed value of 31. Here's what each line does:

train_labels is the list containing the labels for each image in the training set, which was created in the previous code.

The len() function is called on each label in the list to get its length.

The length of each label is appended to the train_label_length list.

The train_input_length list is populated with the fixed value of 31, which is the length of the longest label in the dataset.

The for loop iterates over all of the labels in the training set, adding their lengths to the train_label_length list and setting each corresponding element in the train_input_length list to 31.

Overall, this code prepares the training data for use with a connectionist temporal classification (CTC) loss function, which requires that the length of the input sequences and output labels be specified.

By setting the input length to a fixed value, the model can be trained on inputs of a consistent length, and the CTC loss function can handle variable-length output sequences.

[16]:
```
for i in range(len(valid_labels)):
  valid_label_length.append(len(valid_labels[i]))
  valid_input_length.append(31)
```

[17]:
```
# Padding so that all the words will have similar length

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))
```

This code is performing padding on the train and validation label sequences so that all the labels have the same length. The pad_sequences() function is used from the Keras library to perform the padding.

The train_labels and valid_labels are the label sequences for the training and validation datasets respectively, which are lists of integers representing character indices. max_label_len is the maximum length of any label sequence in the dataset.

The pad_sequences() function takes three required arguments: the sequences to be padded, the maximum length to pad to (maxlen), and the padding position (padding). Here, the padding is done after the sequence values (i.e., padding='post').

The fourth argument value is the value to use for padding. Here, the length of the character list (len(char_list)) is used as the padding value, indicating that this value does not correspond to any actual character in the dataset.

The resulting train_padded_label and valid_padded_label are arrays of shape (num_sequences, max_label_len), where num_sequences is the number of sequences in the respective datasets.

The padded sequences have a length of max_label_len, with any remaining elements beyond the length of the original sequences filled with the padding value.

[18]:
```
train_padded_label.shape, valid_padded_label.shape
```

[18]:
```
((9000, 16), (1000, 16))
```

This code returns the shapes of two numpy arrays train_padded_label and valid_padded_label.

train_padded_label and valid_padded_label are obtained after padding the train_labels and valid_labels respectively using the pad_sequences() function. The maxlen parameter is used to set the maximum length for the padded sequence and padding parameter is used to determine whether to pad sequences at the beginning or end of the sequence.

value parameter is used to set the value to use for padding. Here, the value is set to the length of char_list (which is the total number of unique characters in the dataset plus 1 for the blank character).

The returned values are the shapes of the two padded label arrays. The first value represents the number of samples and the second value represents the maximum length of the sequence after padding.

[19]:
```
# convert the train images into numpy array
train_images = np.asarray(train_images)
train_input_length = np.asarray(train_input_length)
train_label_length = np.asarray(train_label_length)
```

This code converts the train_images, train_input_length, and train_label_length lists into numpy arrays using the np.asarray() function.

Numpy is a Python library used for numerical operations in Python. By converting these lists into numpy arrays, they can be efficiently operated upon using the optimized algorithms provided by the numpy library.

The resulting numpy arrays can be accessed and manipulated in the same way as other numpy arrays, and can be used as inputs for further processing, such as training a machine learning model.

[20]:
```
valid_images = np.asarray(valid_images)
valid_input_length = np.asarray(valid_input_length)
valid_label_length = np.asarray(valid_label_length)
```

4 BUILD THE MODEL

[21]:
```
# 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)
```

This code defines a deep learning model architecture for image-based text recognition. The architecture is as follows:

Input layer: the input image is 32 pixels in height and 128 pixels in width with 1 channel (grayscale image).

Convolutional layer 1: 64 filters with a kernel size of 3x3, activation function ReLU, and padding same.

Pooling layer 1: max pooling with a pool size of 2x2 and stride of 2.

Convolutional layer 2: 128 filters with a kernel size of 3x3, activation function ReLU, and padding same.

Pooling layer 2: max pooling with a pool size of 2x2 and stride of 2.

Convolutional layer 3: 256 filters with a kernel size of 3x3, activation function ReLU, and padding same.

Convolutional layer 4: 256 filters with a kernel size of 3x3, activation function ReLU, and padding same.

Pooling layer 3: max pooling with a pool size of 2x1.

Convolutional layer 5: 512 filters with a kernel size of 3x3, activation function ReLU, and padding same.

Batch normalization layer: to normalize the activations of the previous layer.

Convolutional layer 6: 512 filters with a kernel size of 3x3, activation function ReLU, and padding same.

Batch normalization layer: to normalize the activations of the previous layer.

Pooling layer 4: max pooling with a pool size of 2x1.

Convolutional layer 7: 512 filters with a kernel size of 2x2, activation function ReLU.

Lambda layer: to squeeze the output to remove the dimension of size 1.

Bidirectional LSTM layer 1: 256 units with a dropout rate of 0.2 and returns the sequence.

Bidirectional LSTM layer 2: 256 units with a dropout rate of 0.2 and returns the sequence.

Output layer: a dense layer with a softmax activation function to classify the input image into one of the possible characters. The number of output units is the number of possible characters plus 1 (for the blank character).

Model layer: a model layer with input layer and output layer that can be used at test time.

[22]:
```
act_model.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 32, 128, 1)]      0

 conv2d (Conv2D)             (None, 32, 128, 64)       640

 max_pooling2d (MaxPooling2D  (None, 16, 64, 64)       0
 )

 conv2d_1 (Conv2D)           (None, 16, 64, 128)       73856

 max_pooling2d_1 (MaxPooling  (None, 8, 32, 128)       0
 2D)

 conv2d_2 (Conv2D)           (None, 8, 32, 256)        295168

 conv2d_3 (Conv2D)           (None, 8, 32, 256)        590080

 max_pooling2d_2 (MaxPooling  (None, 4, 32, 256)       0
 2D)

 conv2d_4 (Conv2D)           (None, 4, 32, 512)        1180160

 batch_normalization (BatchN  (None, 4, 32, 512)       2048
 ormalization)

 conv2d_5 (Conv2D)           (None, 4, 32, 512)        2359808

 batch_normalization_1 (Batc  (None, 4, 32, 512)       2048
 hNormalization)

 max_pooling2d_3 (MaxPooling  (None, 2, 32, 512)       0
 2D)
```

```
conv2d_6 (Conv2D)           (None, 1, 31, 512)        1049088

lambda (Lambda)             (None, 31, 512)           0

bidirectional (Bidirectiona  (None, 31, 512)          1574912
l)

bidirectional_1 (Bidirectio  (None, 31, 512)          1574912
nal)

dense (Dense)               (None, 31, 79)            40527

=================================================================
Total params: 8,743,247
Trainable params: 8,741,199
Non-trainable params: 2,048
_________________________________________________________________
```

act_model.summary() is a method used to print out a summary of the neural network model defined in the code. It displays a table with the following columns:

Layer (Name): Name of the layer in the model Output Shape: Shape of the output tensor produced by the layer.

Param # (Number of parameters): Number of trainable parameters in the layer.

Connected to: List of layers that this layer is connected to The summary provides a detailed overview of the layers, their output shapes, and number of parameters in the model. It is a useful tool for debugging and optimizing the neural network architecture.

[23]:
```
# configuring the variables for the CTC lambda function
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')


# defining the CTC lambda function
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)
```

This code is defining a CTC (Connectionist Temporal Classification) lambda function and using it

to define a model for training.

The input of the model consists of four inputs:

inputs: an input tensor of shape (32, 128, 1), representing the image input with height=32 and width=128.

the_labels: a tensor representing the ground truth labels. input_length: a tensor representing the length of the input sequence.

label_length: a tensor representing the length of the label sequence.

The CTC lambda function takes in the outputs from the previous model, outputs, as well as the_labels, input_length, and label_length, and returns the CTC loss.

The model for training is defined by specifying the input and output tensors and the CTC lambda function. The input tensors are specified as a list [inputs, the_labels, input_length, label_length], and the output tensor is the loss_out tensor produced by the CTC lambda function. The resulting model is used for training.

[24]:
```
batch_size = 8
epochs = 20
# epochs = 30
e = str(epochs)
optimizer_name = 'sgd'
```

In this code, three variables are defined: batch_size, epochs, and optimizer_name.

batch_size: It refers to the number of samples per batch of training. Here, it is set to 8, which means that the model will update the weights after every batch of 8 samples.

epochs: It refers to the number of times the entire dataset will be passed through the model during training. Here, it is set to 20.

optimizer_name: It refers to the name of the optimizer that will be used to update the weights of the model during training. Here, it is set to 'sgd', which refers to Stochastic Gradient Descent optimizer.

[25]:
```
model.compile(loss={'ctc': lambda y_true, y_pred: y_pred}, optimizer =␣
 ↪optimizer_name, metrics=['accuracy'])


# For creating the checkpoint for out model so that we can save the optimal␣
 ↪model.
filepath="{}o-{}r-{}e-{}t-{}v.hdf5".format(optimizer_name,
                                           str(RECORDS_COUNT),
                                           str(epochs),
                                           str(train_images.shape[0]),
                                           str(valid_images.shape[0]))

checkpoint = ModelCheckpoint(filepath=filepath, monitor='val_loss', verbose=1,␣
 ↪save_best_only=True, mode='auto')
```

```
callbacks_list = [checkpoint]
```

In this code, the model is compiled using CTC loss function and an optimizer. The CTC loss function is defined as a lambda function.

The optimizer used is specified by the optimizer_name variable. Additionally, the model is configured to compute and report accuracy as a metric during training.

A checkpoint is created using ModelCheckpoint callback to save the optimal model during training.

The filepath argument specifies the name of the file in which to save the model.

The callback monitors the validation loss and saves the model whenever the validation loss decreases, which is determined by save_best_only=True argument.

The callbacks_list variable is a list of all callbacks used during training, which in this case only contains the ModelCheckpoint callback.

[ ]:
```
# Train the model
history = model.fit(x=[train_images, train_padded_label, train_input_length,␣
 ↪train_label_length],
                    y=np.zeros(len(train_images)),
                    batch_size=batch_size,
                    epochs=epochs,
                    validation_data=([valid_images, valid_padded_label,␣
 ↪valid_input_length, valid_label_length], [np.zeros(len(valid_images))]),
                    verbose=1,
                    callbacks=callbacks_list)
```

```
Epoch 1/20
1125/1125 [==============================] - ETA: 0s - loss: 15.0253 - accuracy:
0.0000e+00
Epoch 1: val_loss improved from inf to 16.58457, saving model to
sgdo-10000r-20e-9000t-1000v.hdf5
1125/1125 [==============================] - 264s 229ms/step - loss: 15.0253 -
accuracy: 0.0000e+00 - val_loss: 16.5846 - val_accuracy: 0.0270
Epoch 2/20
1125/1125 [==============================] - ETA: 0s - loss: 11.9752 - accuracy:
0.0427
Epoch 2: val_loss improved from 16.58457 to 10.92107, saving model to
sgdo-10000r-20e-9000t-1000v.hdf5
1125/1125 [==============================] - 259s 230ms/step - loss: 11.9752 -
accuracy: 0.0427 - val_loss: 10.9211 - val_accuracy: 0.0690
Epoch 3/20
1125/1125 [==============================] - ETA: 0s - loss: 9.6128 - accuracy:
0.0851
Epoch 3: val_loss improved from 10.92107 to 9.30470, saving model to
sgdo-10000r-20e-9000t-1000v.hdf5
1125/1125 [==============================] - 281s 250ms/step - loss: 9.6128 -
accuracy: 0.0851 - val_loss: 9.3047 - val_accuracy: 0.1640
Epoch 4/20
```

```
1125/1125 [==============================] - ETA: 0s - loss: 7.0946 - accuracy:
0.1501
Epoch 4: val_loss improved from 9.30470 to 6.49634, saving model to
sgdo-10000r-20e-9000t-1000v.hdf5
1125/1125 [==============================] - 259s 230ms/step - loss: 7.0946 -
accuracy: 0.1501 - val_loss: 6.4963 - val_accuracy: 0.1910
Epoch 5/20
1125/1125 [==============================] - ETA: 0s - loss: 5.1924 - accuracy:
0.2172
Epoch 5: val_loss improved from 6.49634 to 5.26142, saving model to
sgdo-10000r-20e-9000t-1000v.hdf5
1125/1125 [==============================] - 259s 230ms/step - loss: 5.1924 -
accuracy: 0.2172 - val_loss: 5.2614 - val_accuracy: 0.2390
Epoch 6/20
1125/1125 [==============================] - ETA: 0s - loss: 3.9749 - accuracy:
0.2866
Epoch 6: val_loss improved from 5.26142 to 4.40791, saving model to
sgdo-10000r-20e-9000t-1000v.hdf5
1125/1125 [==============================] - 258s 229ms/step - loss: 3.9749 -
accuracy: 0.2866 - val_loss: 4.4079 - val_accuracy: 0.3140
Epoch 7/20
1125/1125 [==============================] - ETA: 0s - loss: 3.1341 - accuracy:
0.3416
Epoch 7: val_loss improved from 4.40791 to 3.77183, saving model to
sgdo-10000r-20e-9000t-1000v.hdf5
1125/1125 [==============================] - 258s 229ms/step - loss: 3.1341 -
accuracy: 0.3416 - val_loss: 3.7718 - val_accuracy: 0.3520
Epoch 8/20
1125/1125 [==============================] - ETA: 0s - loss: 2.5141 - accuracy:
0.4007
Epoch 8: val_loss improved from 3.77183 to 3.54684, saving model to
sgdo-10000r-20e-9000t-1000v.hdf5
1125/1125 [==============================] - 258s 229ms/step - loss: 2.5141 -
accuracy: 0.4007 - val_loss: 3.5468 - val_accuracy: 0.4020
Epoch 9/20
1125/1125 [==============================] - ETA: 0s - loss: 1.9676 - accuracy:
0.4674
Epoch 9: val_loss improved from 3.54684 to 3.35881, saving model to
sgdo-10000r-20e-9000t-1000v.hdf5
1125/1125 [==============================] - 257s 228ms/step - loss: 1.9676 -
accuracy: 0.4674 - val_loss: 3.3588 - val_accuracy: 0.4130
Epoch 10/20
1125/1125 [==============================] - ETA: 0s - loss: 1.5288 - accuracy:
0.5350
Epoch 10: val_loss did not improve from 3.35881
1125/1125 [==============================] - 257s 228ms/step - loss: 1.5288 -
accuracy: 0.5350 - val_loss: 3.3627 - val_accuracy: 0.4630
Epoch 11/20
```

```
1125/1125 [==============================] - ETA: 0s - loss: 1.1683 - accuracy:
0.5988
Epoch 11: val_loss did not improve from 3.35881
1125/1125 [==============================] - 256s 228ms/step - loss: 1.1683 -
accuracy: 0.5988 - val_loss: 3.3861 - val_accuracy: 0.4710
Epoch 12/20
1125/1125 [==============================] - ETA: 0s - loss: 0.8831 - accuracy:
0.6664
Epoch 12: val_loss did not improve from 3.35881
1125/1125 [==============================] - 256s 227ms/step - loss: 0.8831 -
accuracy: 0.6664 - val_loss: 3.6844 - val_accuracy: 0.4730
Epoch 13/20
1125/1125 [==============================] - ETA: 0s - loss: 0.6619 - accuracy:
0.7238
Epoch 13: val_loss did not improve from 3.35881
1125/1125 [==============================] - 257s 228ms/step - loss: 0.6619 -
accuracy: 0.7238 - val_loss: 3.6204 - val_accuracy: 0.5020
Epoch 14/20
1125/1125 [==============================] - ETA: 0s - loss: 0.4804 - accuracy:
0.7891
Epoch 14: val_loss improved from 3.35881 to 3.25377, saving model to
sgdo-10000r-20e-9000t-1000v.hdf5
1125/1125 [==============================] - 256s 228ms/step - loss: 0.4804 -
accuracy: 0.7891 - val_loss: 3.2538 - val_accuracy: 0.5320
Epoch 15/20
1125/1125 [==============================] - ETA: 0s - loss: 0.3479 - accuracy:
0.8333
Epoch 15: val_loss did not improve from 3.25377
1125/1125 [==============================] - 257s 228ms/step - loss: 0.3479 -
accuracy: 0.8333 - val_loss: 3.4540 - val_accuracy: 0.5500
Epoch 16/20
1125/1125 [==============================] - ETA: 0s - loss: 0.2605 - accuracy:
0.8789
Epoch 16: val_loss did not improve from 3.25377
1125/1125 [==============================] - 256s 228ms/step - loss: 0.2605 -
accuracy: 0.8789 - val_loss: 3.3247 - val_accuracy: 0.5480
Epoch 17/20
1125/1125 [==============================] - ETA: 0s - loss: 0.2037 - accuracy:
0.9043
Epoch 17: val_loss did not improve from 3.25377
1125/1125 [==============================] - 257s 229ms/step - loss: 0.2037 -
accuracy: 0.9043 - val_loss: 3.5251 - val_accuracy: 0.5830
Epoch 18/20
1125/1125 [==============================] - ETA: 0s - loss: 0.1490 - accuracy:
0.9363
Epoch 18: val_loss did not improve from 3.25377
1125/1125 [==============================] - 257s 228ms/step - loss: 0.1490 -
accuracy: 0.9363 - val_loss: 3.9268 - val_accuracy: 0.5210
```

```
Epoch 19/20
1125/1125 [==============================] - ETA: 0s - loss: 0.1463 - accuracy:
0.9382
Epoch 19: val_loss did not improve from 3.25377
1125/1125 [==============================] - 259s 230ms/step - loss: 0.1463 -
accuracy: 0.9382 - val_loss: 3.5190 - val_accuracy: 0.5910
Epoch 20/20
1125/1125 [==============================] - ETA: 0s - loss: 0.1125 - accuracy:
0.9549
Epoch 20: val_loss did not improve from 3.25377
1125/1125 [==============================] - 256s 228ms/step - loss: 0.1125 -
accuracy: 0.9549 - val_loss: 3.5831 - val_accuracy: 0.5850
```

This code trains the model using the fit() method of Keras.

The input data for the model training is provided as a tuple of 4 numpy arrays representing the training images, padded training labels, training input length, and training label length. These arrays are passed as x argument.

The y argument is set to an array of zeros of length equal to the number of training images. This is because the output of the model is the loss, which is computed using the CTC lambda function defined earlier, and does not require a target label.

The batch size and number of epochs are set to 8 and 20 respectively, and the validation data is also provided in a similar format as the training data. The verbose parameter is set to 1 to display the training progress.

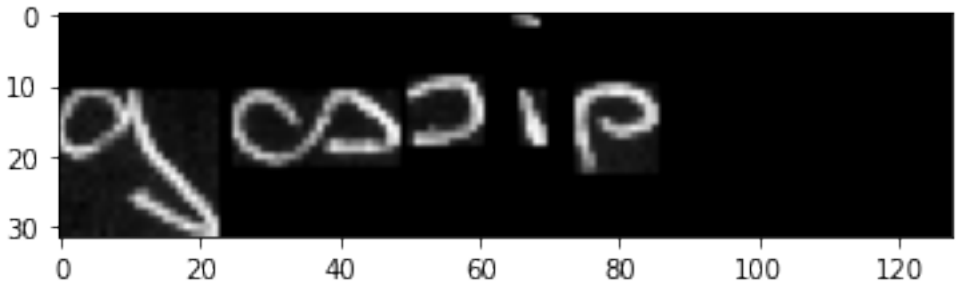
The training is done with the specified optimizer and loss function. During training, a checkpoint is created and saved after every epoch if the validation loss decreases. These checkpoints are saved to a file path that is constructed using the optimizer name, the number of records, number of epochs, number of training images, and number of validation images.

Finally, the history object is returned, which contains the training loss and accuracy, as well as the validation loss and accuracy for each epoch.

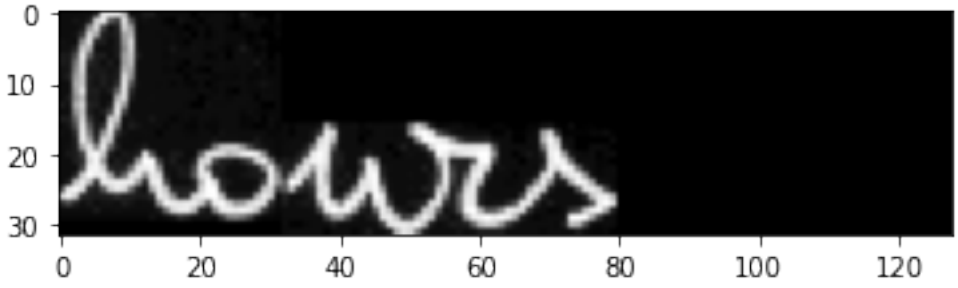
[29]:
```
# load the saved best model weights
act_model.load_weights(filepath)

# predict outputs on validation images
prediction = act_model.predict(train_images[150:170])

# use CTC decoder for prediction
decoded = K.ctc_decode(prediction,
                       input_length=np.ones(prediction.shape[0]) * prediction.
 ↪shape[1],
                       greedy=True)[0][0]

# get the predicted result
out = K.get_value(decoded)
```

```
# see the results by enumerate over the predicted result
for i, x in enumerate(out):
    # change the index values (150 + i) to see different result
    plt.imshow(train_images[150+i].reshape(32,128), cmap=plt.cm.gray)
    plt.show()
    print("predicted text = ", end = '')
    for p in x:
        if int(p) != -1:
            print(char_list[int(p)], end = '')
    print('\n')
```

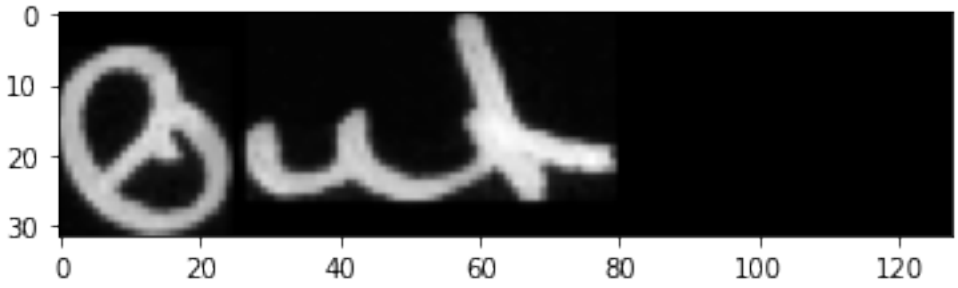
```
1/1 [==============================] - 0s 91ms/step
```



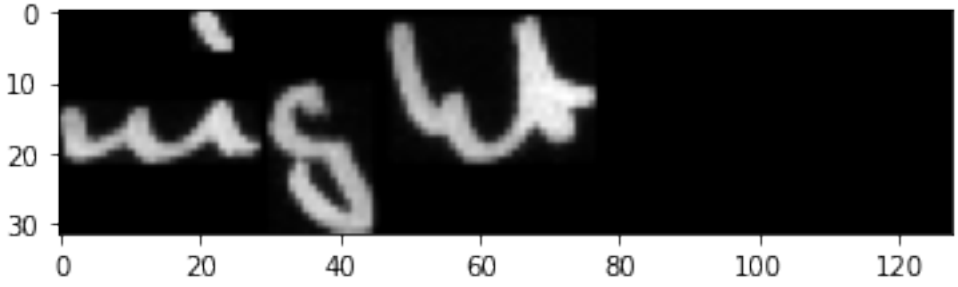

```
predicted text = gospip
```



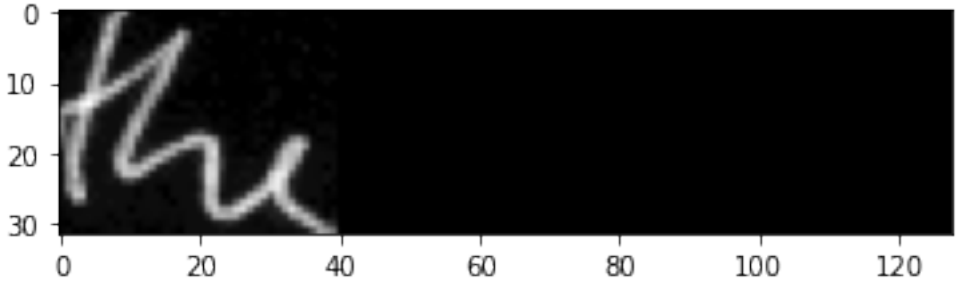

```
predicted text = howrs
```



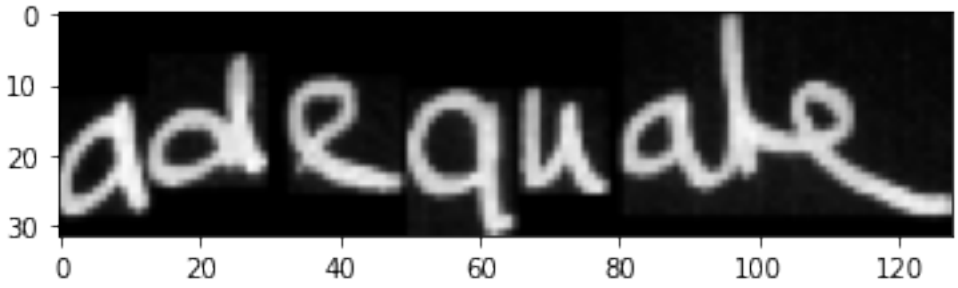

```
predicted text = but
```



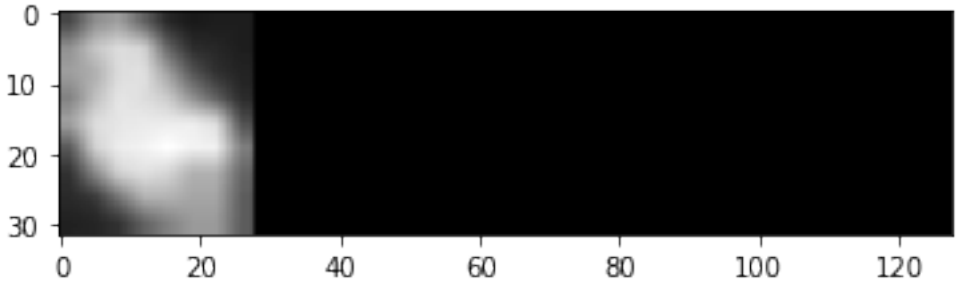

```
predicted text = night
```



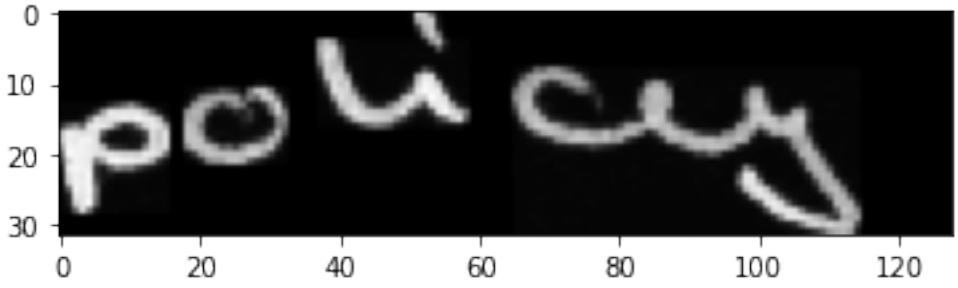

```
predicted text = the
```



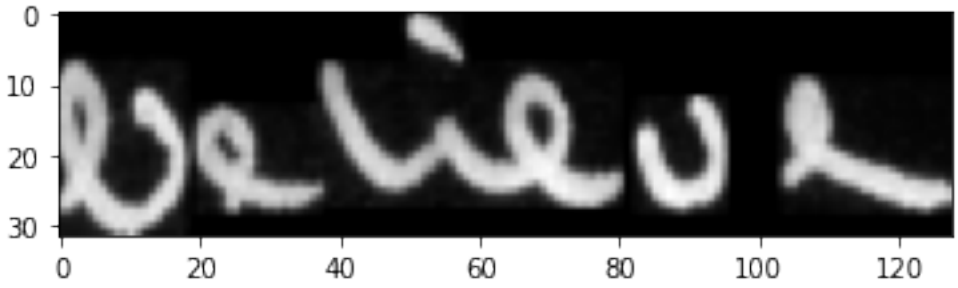
predicted text = adequate



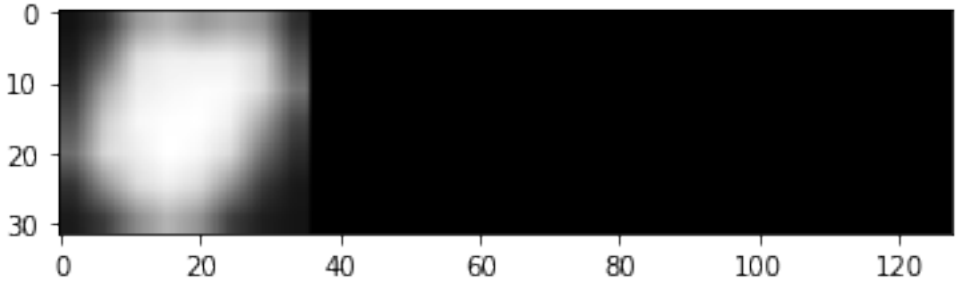
predicted text = .



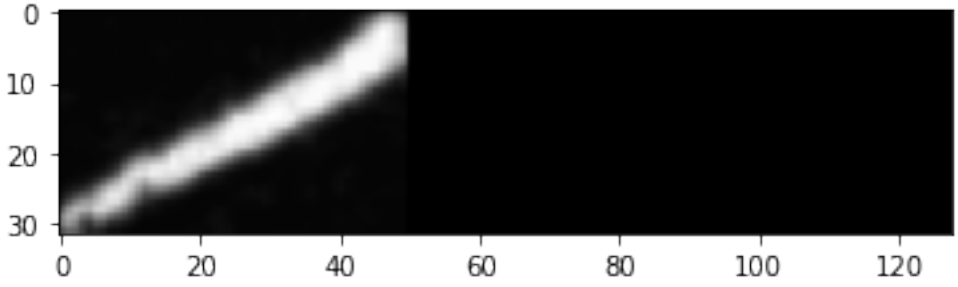
predicted text = policyy



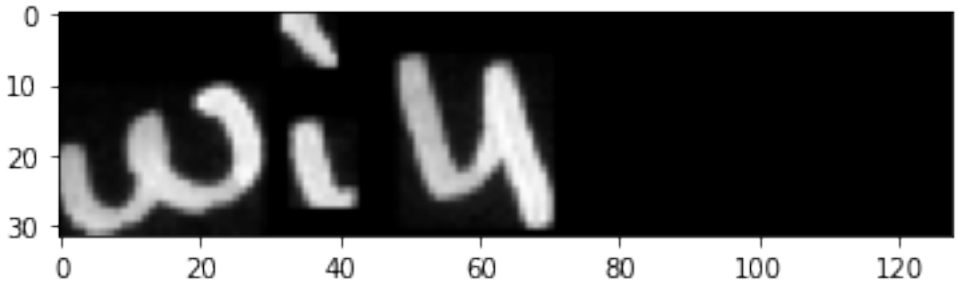

```
predicted text = believe
```



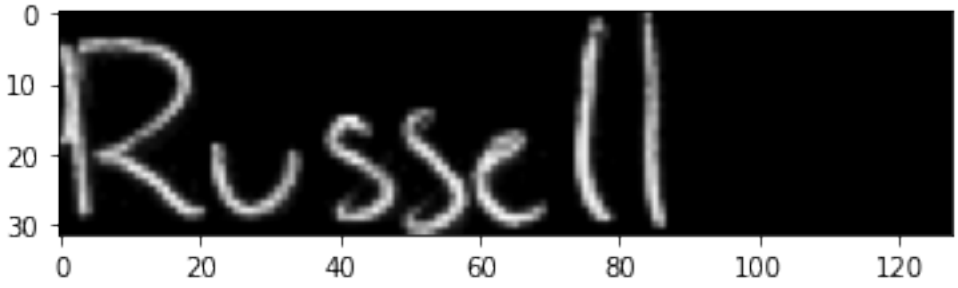

```
predicted text = .
```



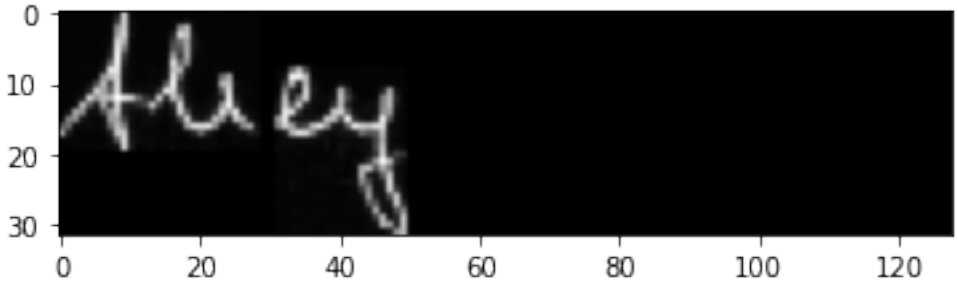

```
predicted text = ,
```



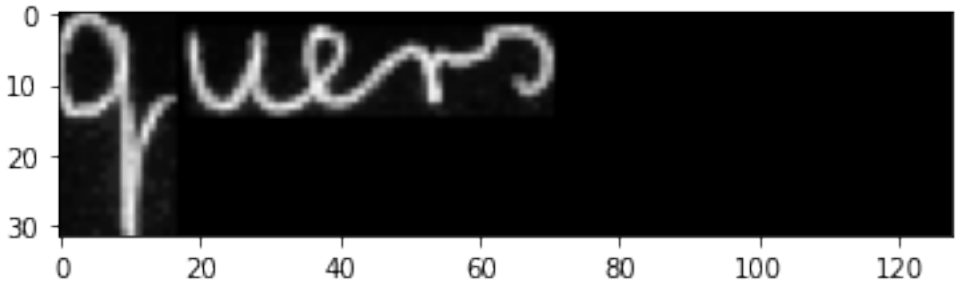

```
predicted text = will
```



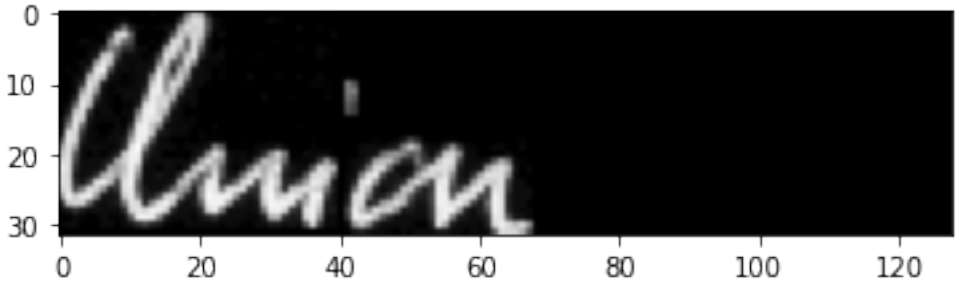

```
predicted text = Russell
```



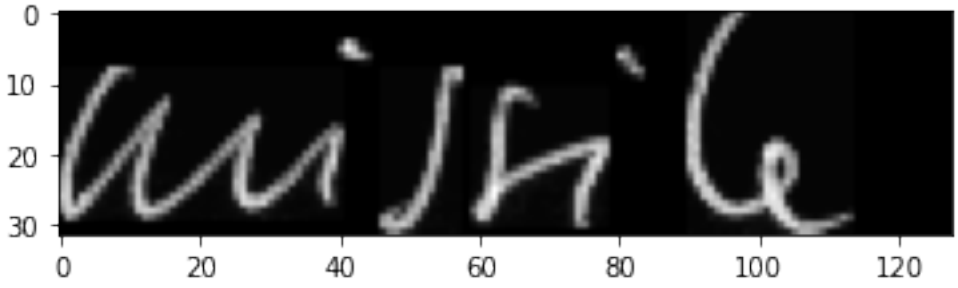

```
predicted text = they
```



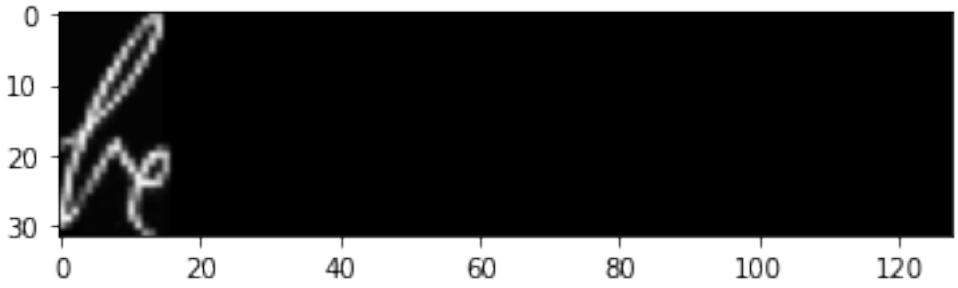

```
predicted text = guers
```



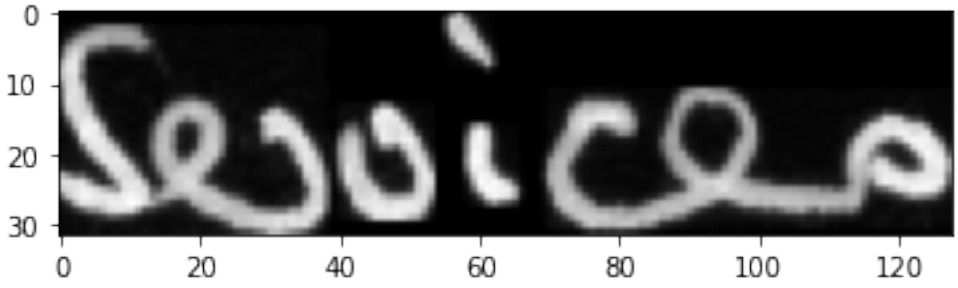

```
predicted text = Umion
```



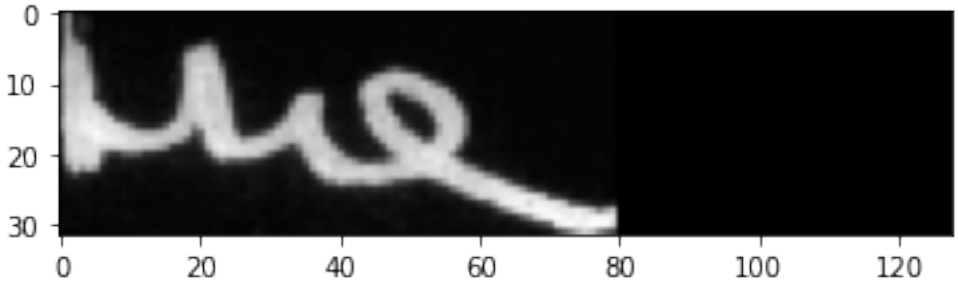

```
predicted text = missile
```




```
predicted text = be
```




```
predicted text = Services
```




```
predicted text = the
```

This code loads the saved weights of the best model from the file path specified earlier. It then predicts outputs for a subset of validation images using act_model.predict(), and uses the CTC decoder to decode the output predictions. The decoded predictions are then printed out in a loop that iterates over each predicted output.

For each predicted output, it displays the corresponding image and prints out the predicted text by converting the CTC decoded index values to corresponding characters using the char_list defined earlier.

[ ]:
```
# plot accuracy and loss
def plotgraph_accuracy(epochs, acc, val_acc):
    # Plot training & validation accuracy values
    plt.plot(epochs, acc, 'b')
    plt.plot(epochs, val_acc, 'r')
    plt.title('Model accuracy')
    plt.ylabel('Accuracy')
    plt.xlabel('Epoch')
    plt.legend(['Train', 'Val'], loc='upper left')
    plt.show()
```

This code defines a function called plotgraph_accuracy that takes in three arguments: epochs, acc, and val_acc. These arguments represent the number of epochs, training accuracy, and validation accuracy, respectively.

The function plots the training and validation accuracy on a graph with the number of epochs on the x-axis and the accuracy on the y-axis.

The training accuracy is represented by a blue line and the validation accuracy is represented by a red line.

The function also sets the title of the graph to “Model accuracy” and the labels of the x and y axes to “Epoch” and “Accuracy”, respectively.

Finally, the function adds a legend to the graph to distinguish between the training and validation accuracy lines, and displays the graph using plt.show().

[ ]:
```
# plot accuracy and loss
def plotgraph_loss(epochs, loss, val_loss):
    # Plot training & validation accuracy values
    plt.plot(epochs, loss, 'b')
    plt.plot(epochs, val_loss, 'r')
    plt.title('Model loss')
    plt.ylabel('Loss')
    plt.xlabel('Epoch')
    plt.legend(['Train', 'Val'], loc='upper left')
    plt.show()
```

This code defines a function called plotgraph_loss that takes in three parameters: epochs, loss, and val_loss.

The function uses the matplotlib library to plot the loss and val_loss values against the number of epochs.

The resulting plot has the training loss values in blue and validation loss values in red, with a title “Model loss” and axes labels “Epoch” and “Loss”.

The plot also has a legend indicating which line represents the training and validation loss values.

```
[ ]: # Get the accuracy, loss, and other information
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1,len(loss)+1)
```

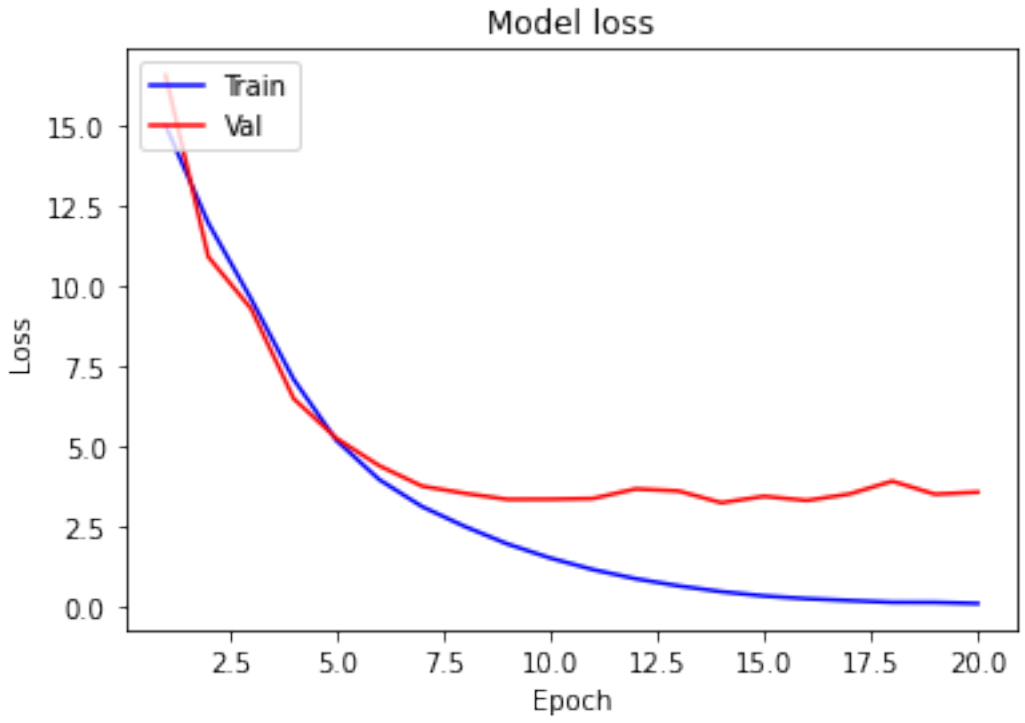
This code snippet retrieves the accuracy, loss, and other information about the training and validation of the model stored in the history object after training.

Specifically, it gets the accuracy and loss values for the training and validation sets (acc, val_acc, loss, and val_loss, respectively), and the number of epochs for which the model was trained (epochs).

The history object is a dictionary that contains the values of different metrics computed during training and validation, such as accuracy and loss, at each epoch.

The values of these metrics are then plotted using the plotgraph_accuracy and plotgraph_loss functions.

```
[ ]: # Plot the LOSS VS EPOCH
plotgraph_loss(epochs, loss, val_loss)
```



This code calls the plotgraph_loss function to plot the training and validation loss values against the number of epochs. It takes in four arguments:

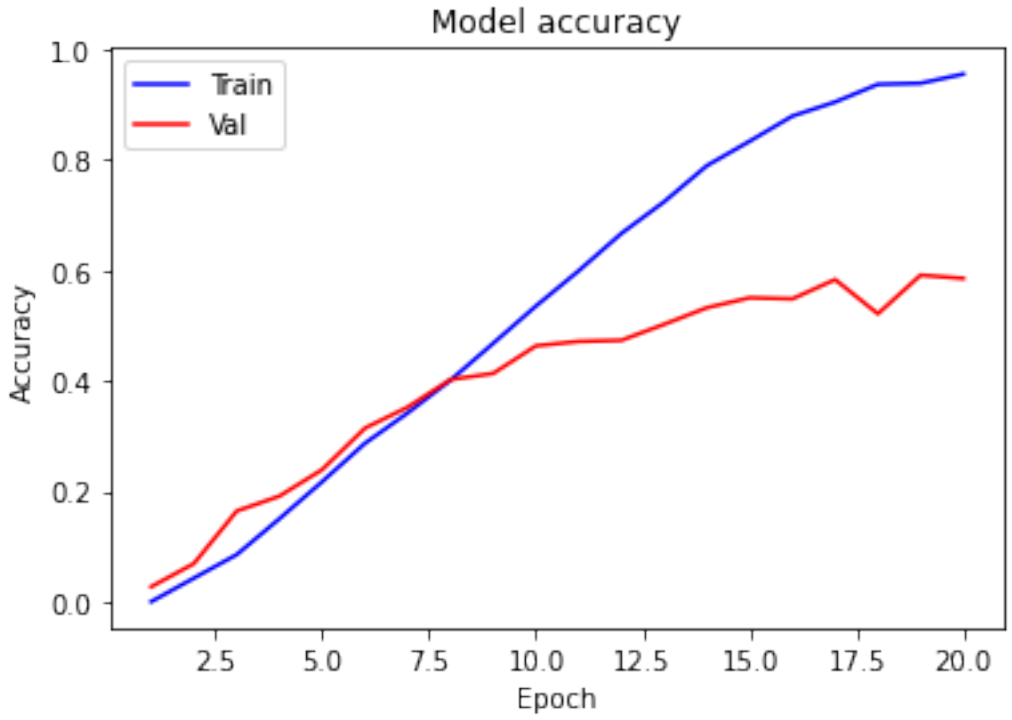
epochs: a range of integers from 1 to the number of epochs the model was trained for.

loss: a list of float values representing the training loss values for each epoch.

val_loss: a list of float values representing the validation loss values for each epoch.

The plotgraph_loss function uses matplotlib to plot the training and validation loss curves on the same graph with different colors. It also sets labels for the x-axis, y-axis, and title of the plot. Finally, it displays the plot using plt.show().

[ ]:
```
# Plot the ACCURACY VS EPOCH
plotgraph_accuracy(epochs, acc, val_acc)
```



This code calls the plotgraph_accuracy function to plot the training and validation accuracy values against the number of epochs. It takes in four arguments:

acc: a list of float values representing the training accuracy values for each epoch.

val_acc: a list of float values representing the validation accuracy values for each epoch.

The plotgraph_accuracy function uses matplotlib to plot the training and validation accuracy curves on the same graph with different colors. It also sets labels for the x-axis, y-axis, and title of the plot. Finally, it displays the plot using plt.show().