Title: AI-Based Handwritten Text Recognition

Abstract

Handwritten text recognition (HTR) is a rapidly advancing field that leverages artificial intelligence to transform handwritten notes, forms, and scanned documents into machine-readable text. The challenges of handwritten recognition include dealing with varied handwriting styles, noise in images, and low-quality scans. This paper explores state-of-the-art techniques in HTR, covering methods in data preprocessing, model building, and evaluation. We discuss the use of deep learning, particularly Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs), as well as the role of data preprocessing and feature extraction in enhancing model performance.

➔ (i) Literature Survey Related to the Topic

1) OPTICAL CHARACTER RECOGNITION (Dr. V. Geetha1, Ch V V Sudheer2, A V Saikumar3 , Dr C K Gomathy4 (Kancheepuram, TamilNadu, India))

Link : https://www.researchgate.net/publication/360620085_OPTICAL_CHARACTER_RECOGNITION

Summary:

Optical Character Recognition (OCR) technology converts printed and handwritten text into digital formats for efficient data handling. Widely applied in tools like Apple's Live Text and Google Lens, OCR faces challenges with complex documents that have varied text styles. To improve accuracy, the proposed method uses Tesseract OCR with OpenCV and Python for real-time text recognition and keyword search, achieving 96.5% accuracy. This highlights OCR's role in enhancing document accessibility.

2) A Study of the OCR Development History and Directions of Development (Junmiao Wang Nanjing University of Posts and Telecommunications Nanjing, China)

Link: https://www.researchgate.net/publication/377803605_A_Study_of_The_OCR_Development_History_and_Directions_of_Development

Summary:

The journey of Optical Character Recognition (OCR) technology from its early days of basic character recognition to today's sophisticated systems powered by neural networks. OCR now plays a crucial role in digitizing texts, boosting accuracy, and supporting diverse languages. Recent breakthroughs, especially with deep learning techniques like CNNs and RNNs, have greatly improved OCR's adaptability and precision. The research also envisions OCR's future in tackling complex documents and helping preserve endangered languages.

3) Review of Optical Character Recognition for Power System Image Based on Artifcial Intelligence Algorithm (Xun Zhang1,WanrongBai 1and Haoyang Cu)

Link: https://www.researchgate.net/publication/366834297_Review_of_Optical_Character_Recognition_for_Power_System_Image_Based_on_Artificial_Intelligence_Algorithm

Summary:

Optical Character Recognition (OCR) technology is advancing in power systems through artificial intelligence (AI). It highlights the shift from early neural network-based OCR, which had issues with accuracy and adaptability, to newer machine learning and deep learning models. Although deep learning has made OCR more precise and flexible, challenges with optimizing and speeding up the process still exist. The study emphasizes the importance of developing high-precision OCR solutions to handle the unique demands of power system environments effectively.

4) Handwritten Optical Character Recognition (OCR): A Comprehensive Systematic Literature Review (SLR)

Link:

https://www.researchgate.net/publication/343273822_Handwritten_Optical_Character_Recognition_OCR_A_Comprehensive_Systematic_Literature_Review_SLR

Summary:

This systematic review gives a detailed look at research on handwritten Optical Character Recognition (OCR) from 2000 to 2019, highlighting advances across multiple languages, including English, Arabic, Urdu, and Chinese. It explores the use of machine learning techniques like Support Vector Machines, Random Forests, and deep learning models, each contributing to more accurate character recognition. The review also points out areas where OCR can improve, especially in handling diverse languages and complex handwriting styles, and includes key datasets and methods that have pushed OCR technology forward.

5) A Survey of OCR Applications (Amarjot Singh, Ketan Bacchuwar, and Akshay Bhasin)

Link:

https://www.researchgate.net/publication/269395249_A_Survey_of_OCR_Applications

Summary:

This paper explores the many uses of Optical Character Recognition (OCR), a technology that converts scanned text into searchable digital content. OCR is applied across diverse fields, including banking, healthcare, and online security (like CAPTCHA systems). It also plays a key role in digital libraries, music recognition, and automatic license plate recognition. The authors introduce an advanced OCR method that uses genetic algorithms and histogram equalization, making it effective even with noisy, blurry, or unevenly lit images.

6) Optical Character Recognition by Open Source OCR Tool Tesseract: A Case Study

Link:

https://www.researchgate.net/publication/235956427_Optical_Character_Recognition_by_Open_source_OCR_Tool_Tesseract_A_Case_Study

Summary:

The article examines Tesseract, an open-source OCR tool for converting printed text to editable form, particularly in applications like license plate recognition. Covering Tesseract's architecture and text extraction process, the study compares it with the commercial tool Transym. Tesseract demonstrated superior accuracy, especially on grayscale images (70% vs. 47%), and was faster overall. Concluding that Tesseract's open-source nature and platform flexibility make it highly effective, the paper underscores its advantages in specialized OCR applications.

7) A Comparative Study on OCR Tools (Carlos Mello and Rafael Dueire Lins)

Link:

https://www.researchgate.net/publication/2555664_A_Comparative_Study_on_OCR_Tools

Summary:
The article "A Comparative Study on OCR Tools" evaluates the performance of six commercial OCR software options, including Omnipage and Corel OCR Trace, focusing on factors like resolution, rotation, brightness, and background colors. Omnipage achieved the best results overall, especially at lower resolutions and with grayscale images. Corel OCR Trace performed well in specific cases, particularly with complex historical documents. The study highlights each tool’s suitability for different use cases and suggests that no single OCR tool excels universally across all scenarios.

8) A Review of Research on Optical Character Recognition for Devanagari Language (Shubham Agrawal, Sumit Mathur)

Link:
https://www.researchgate.net/publication/358607896_A_Review_of_Research_on_Optical_Character_Recognition_for_Devanagari_Language

Summary:

The article "A Review of Research on Optical Character Recognition for Devanagari Language" explores advancements in OCR systems for recognizing Devanagari script. It outlines OCR methods such as image acquisition, preprocessing, segmentation, feature extraction, and classification. The study discusses unique challenges in recognizing Devanagari script due to its complex character structure. By reviewing various techniques and classifiers, it highlights promising research directions for enhancing OCR accuracy in handwritten and printed Devanagari, emphasizing the importance of linguistic-specific OCR systems for Indian languages.

9) Optical Character Recognition from Text Image

Link:
https://www.researchgate.net/publication/272709776_Optical_Character_Recognition_from_Text_Image

Summary:

The paper discusses Optical Character Recognition (OCR), a technology for converting text images into editable documents. It details a method using texture and topological features to enhance character recognition accuracy. The process involves image preprocessing, feature extraction, and template matching. Experiments show high accuracy for certain fonts, achieving up to 100% for 'Berlin Sans.' OCR applications range across sectors, including finance and law, where automation aids in digitizing documents, making them searchable and more accessible.

10) A Detailed Analysis of Optical Character Recognition Technology (Karez Abdulwahhab Hamad , Mehmet Kaya)

Link:

https://www.researchgate.net/publication/311851325_A_Detailed_Analysis_of_Optical_Character_Recognition_Technology

Summary:

The paper provides a comprehensive overview of Optical Character Recognition (OCR) technology, focusing on converting printed or handwritten text into editable digital formats. It examines challenges such as font variations, lighting, and image quality, and details OCR phases like preprocessing, segmentation, feature extraction, and classification. Applications of OCR span banking, healthcare, and legal fields, aiding in automating document management. The paper also traces OCR’s historical development and discusses advancements in overcoming limitations for enhanced accuracy.

11) **A Review on OCR Technology (Jay Dilipbhai Thanki, Priyank Dineshbhai Davda, Dr. Priya Swaminarayan)**

Summary:

The article reviews Optical Character Recognition (OCR) technology, which converts scanned text into machine-readable data for digital storage, editing, and retrieval. Key techniques like pre-processing, segmentation, and neural networks improve OCR accuracy. Applications include data entry, assisting the visually impaired, and traffic sign recognition. OCR supports the “Digital India” initiative, aiming for paper-free storage and greater efficiency across sectors, benefiting areas such as banking, legal, and transportation.

12) Research Paper on Text Extraction using OCR (Prof. Anuradha Thorat, Mayur Zagade, Shivani More, Manish Pasalkar, Anand Narute)

Summary:

The article discusses text extraction from images using Optical Character Recognition (OCR), a key technique in document processing and computer vision. OCR enables converting text in images into digital, editable formats. The process includes steps like image preprocessing, text detection, segmentation, and recognition, often using neural networks for accuracy. OCR applications span fields such as document digitization and information retrieval, helping automate processes by efficiently handling large volumes of visual data, even with complex backgrounds and varying text styles.

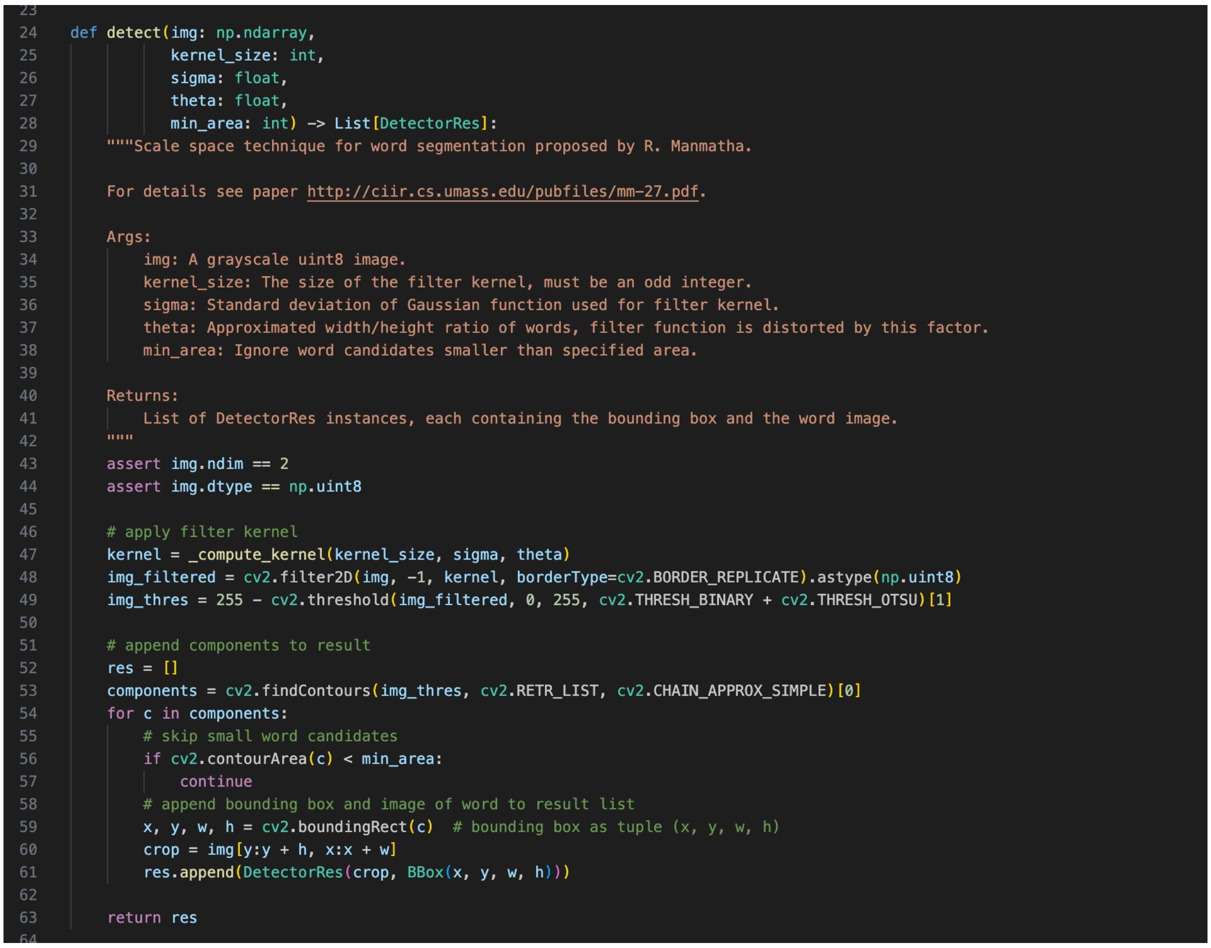
→(ii) Data Pre-Processing

Data preprocessing here is designed to prepare images effectively for text detection. It includes steps that enhance image clarity and standardize sizes, ensuring consistent input for accurate word segmentation and recognition in later stages.

Following are the expanded explanation of each preprocessing step :

1. **Grayscale Conversion** (prepare_img function):
    - The function converts images with 3 color channels (RGB) to grayscale if necessary. This reduces the image to a single intensity channel, which simplifies further processing and enhances contrast, making it easier to detect text boundaries.
2. **Resizing** (prepare_img function):

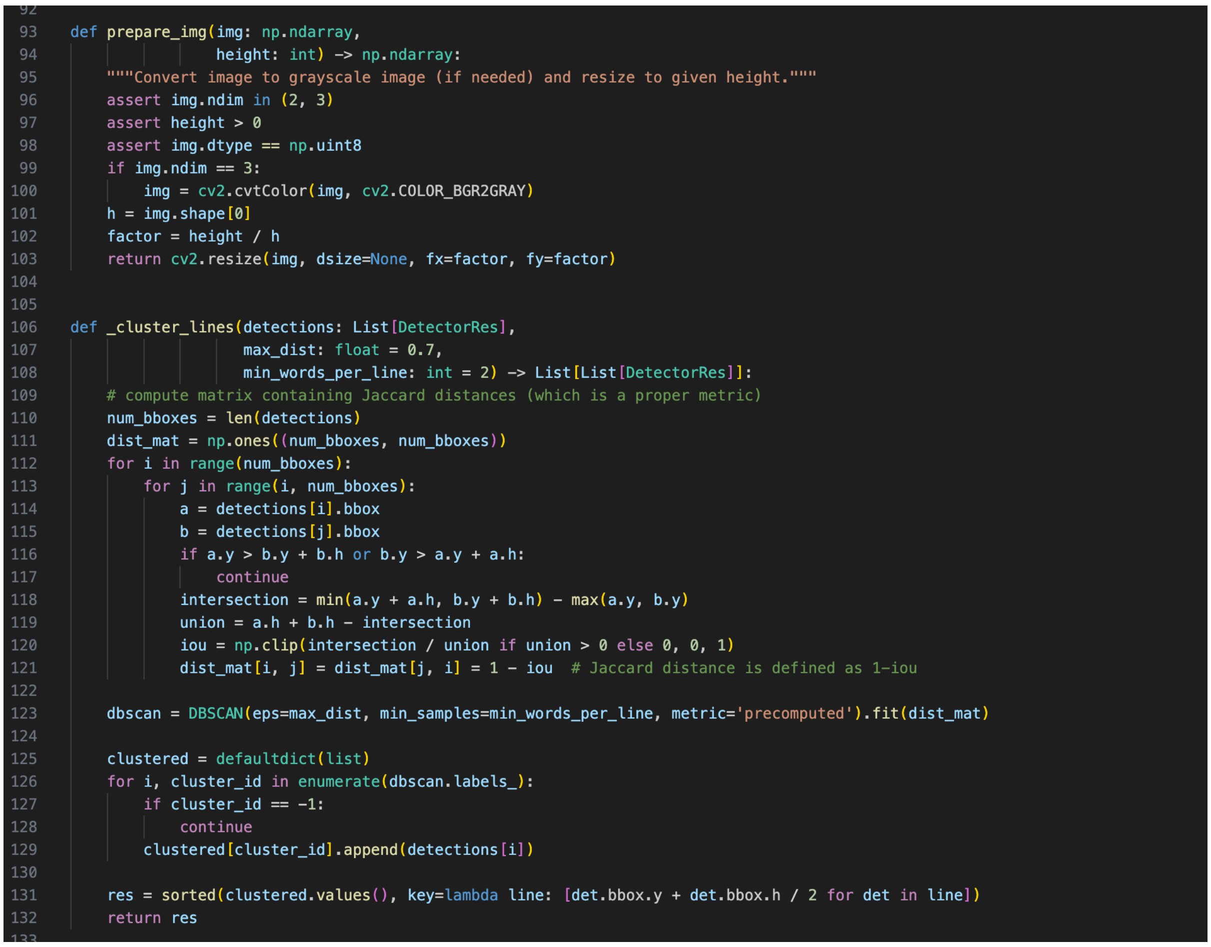
- Resizing to a specified height maintains a consistent scale across images, critical for word detection. It adjusts the scale factor based on the height of the image, ensuring that word dimensions remain proportional across different input images.



```python
def detect(img: np.ndarray,
           kernel_size: int,
           sigma: float,
           theta: float,
           min_area: int) -> List[DetectorRes]:
    """Scale space technique for word segmentation proposed by R. Manmatha.

    For details see paper http://ciir.cs.umass.edu/pubfiles/mm-27.pdf.

    Args:
        img: A grayscale uint8 image.
        kernel_size: The size of the filter kernel, must be an odd integer.
        sigma: Standard deviation of Gaussian function used for filter kernel.
        theta: Approximated width/height ratio of words, filter function is distorted by this factor.
        min_area: Ignore word candidates smaller than specified area.

    Returns:
        List of DetectorRes instances, each containing the bounding box and the word image.
    """
    assert img.ndim == 2
    assert img.dtype == np.uint8

    # apply filter kernel
    kernel = _compute_kernel(kernel_size, sigma, theta)
    img_filtered = cv2.filter2D(img, -1, kernel, borderType=cv2.BORDER_REPLICATE).astype(np.uint8)
    img_thres = 255 - cv2.threshold(img_filtered, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)[1]

    # append components to result
    res = []
    components = cv2.findContours(img_thres, cv2.RETR_LIST, cv2.CHAIN_APPROX_SIMPLE)[0]
    for c in components:
        # skip small word candidates
        if cv2.contourArea(c) < min_area:
            continue
        # append bounding box and image of word to result list
        x, y, w, h = cv2.boundingRect(c)  # bounding box as tuple (x, y, w, h)
        crop = img[y:y + h, x:x + w]
        res.append(DetectorRes(crop, BBox(x, y, w, h)))

    return res
```

3. **Anisotropic Filtering** (detect function):
    - The _compute_kernel function creates a custom anisotropic filter kernel based on parameters like kernel_size, sigma, and theta. This filter emphasizes edges and word-like shapes, using Gaussian distribution to focus on potential word regions, which are often elongated and narrow.
4. **Thresholding** (detect function):

- After filtering, binary thresholding with Otsu's method inverts the filtered image and converts it to a binary image. This makes word boundaries more distinct by setting text regions to white (255) and the background to black (0), isolating text for segmentation.



```python
def prepare_img(img: np.ndarray,
                height: int) -> np.ndarray:
    """Convert image to grayscale image (if needed) and resize to given height."""
    assert img.ndim in (2, 3)
    assert height > 0
    assert img.dtype == np.uint8
    if img.ndim == 3:
        img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    h = img.shape[0]
    factor = height / h
    return cv2.resize(img, dsize=None, fx=factor, fy=factor)


def _cluster_lines(detections: List[DetectorRes],
                   max_dist: float = 0.7,
                   min_words_per_line: int = 2) -> List[List[DetectorRes]]:
    # compute matrix containing Jaccard distances (which is a proper metric)
    num_bboxes = len(detections)
    dist_mat = np.ones((num_bboxes, num_bboxes))
    for i in range(num_bboxes):
        for j in range(i, num_bboxes):
            a = detections[i].bbox
            b = detections[j].bbox
            if a.y > b.y + b.h or b.y > a.y + a.h:
                continue
            intersection = min(a.y + a.h, b.y + b.h) - max(a.y, b.y)
            union = a.h + b.h - intersection
            iou = np.clip(intersection / union if union > 0 else 0, 0, 1)
            dist_mat[i, j] = dist_mat[j, i] = 1 - iou  # Jaccard distance is defined as 1-iou

    dbscan = DBSCAN(eps=max_dist, min_samples=min_words_per_line, metric='precomputed').fit(dist_mat)

    clustered = defaultdict(list)
    for i, cluster_id in enumerate(dbscan.labels_):
        if cluster_id == -1:
            continue
        clustered[cluster_id].append(detections[i])

    res = sorted(clustered.values(), key=lambda line: [det.bbox.y + det.bbox.h / 2 for det in line])
    return res
```

These preprocessing steps prepare the images for accurate word boundary detection and clustering for effective text segmentation.

→(iii) Data Visualization / EDA using Python Libraries

Exploratory Data Analysis (EDA) and visualization provide insights into the dataset, helping identify potential preprocessing needs and understand the distribution of different characters or digits.

Libraries Used :

Matplotlib:

A grid layout (4x4) displays multiple images simultaneously, helping visualize a batch of data at once.

Images are shown in grayscale, and decoded labels are displayed above each image to represent what the model sees versus its expected output.

The figure is rendered with clean, axis-free visuals for readability.

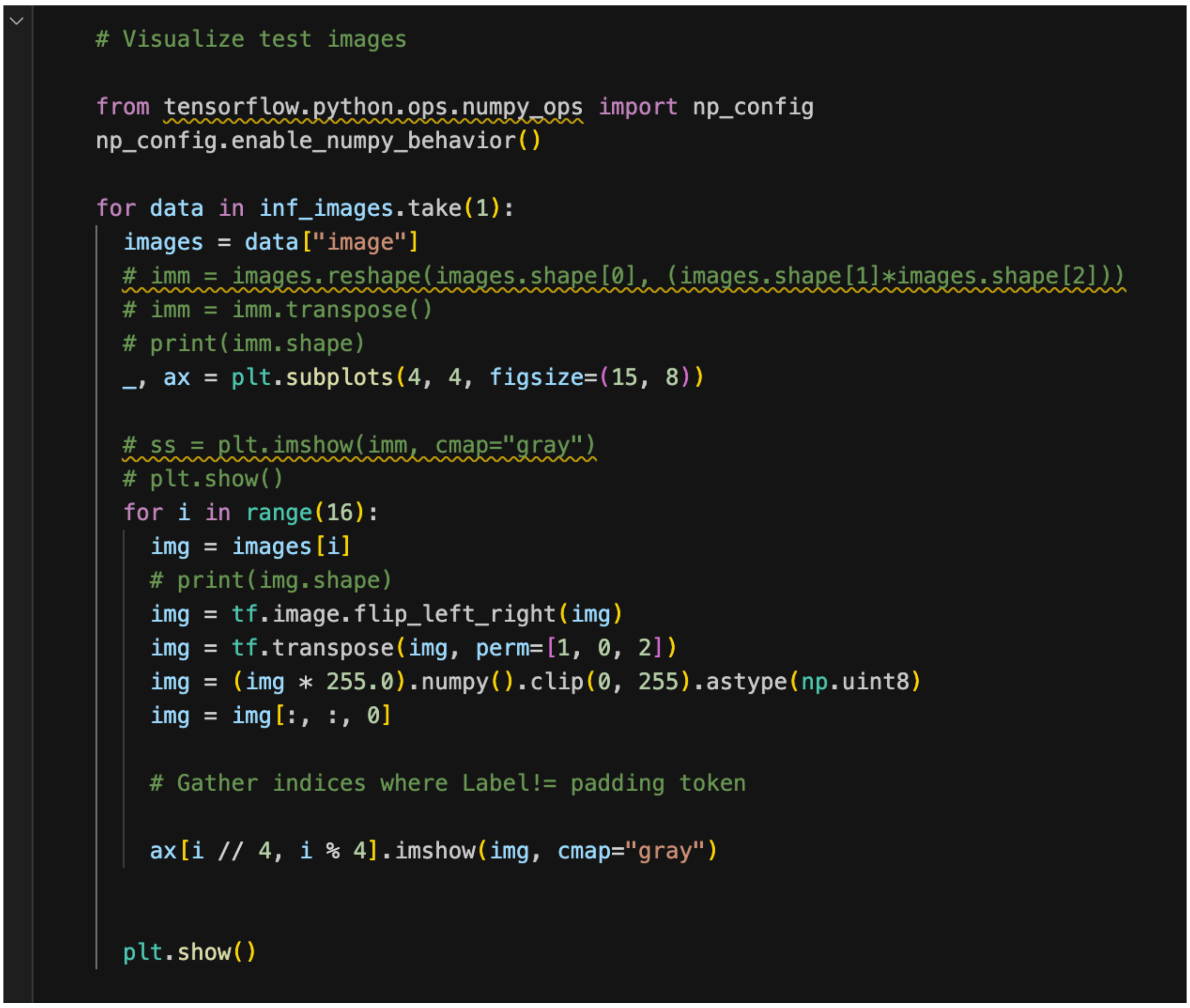
TensorFlow:

Functions adjust image orientation and prepare data in the right format for visualization.

Labels are decoded by removing padding tokens and converting numeric encodings back to text, allowing direct comparison with the images.

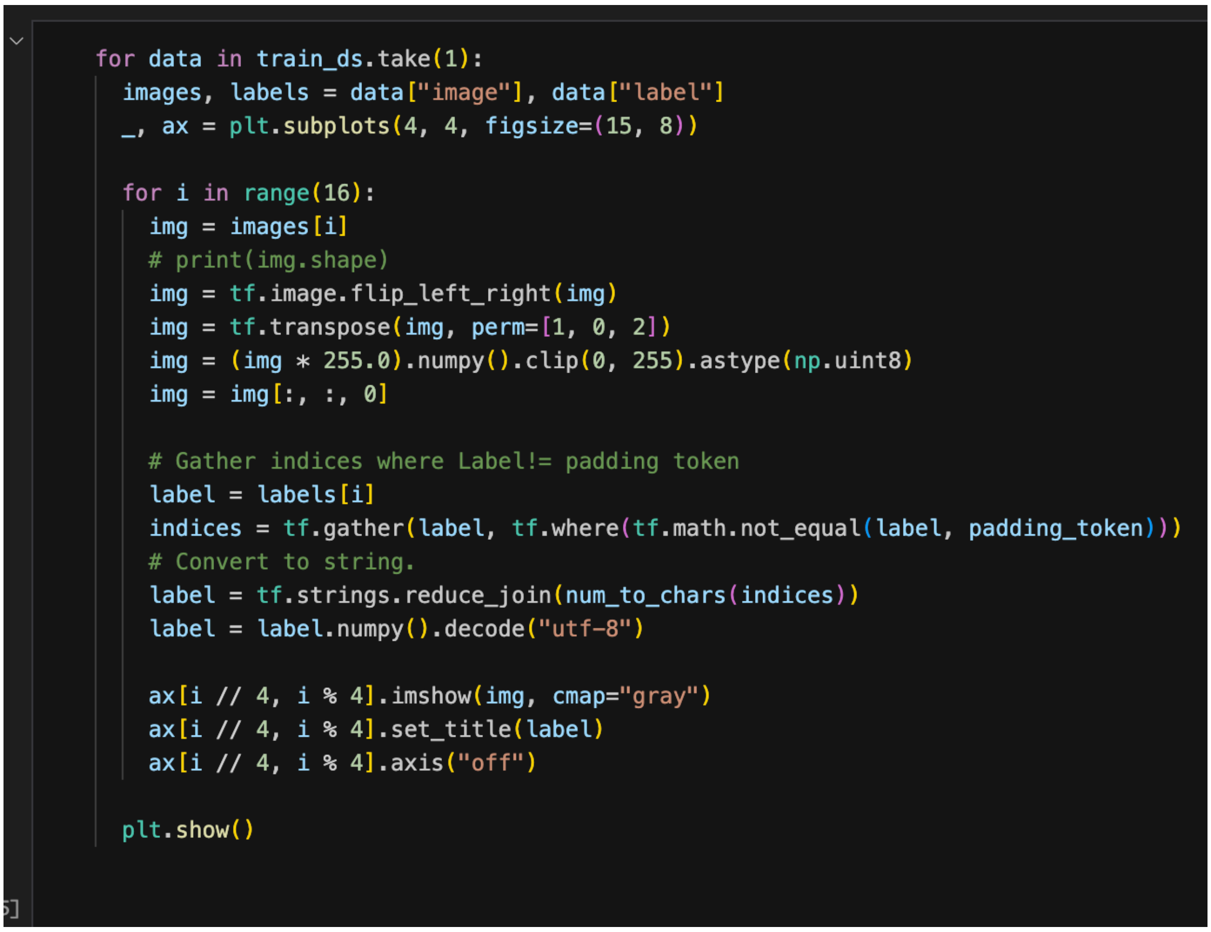
<u>The data visualization code is implemented in the following parts of the document, using matplotlib to display images and their labels from the dataset:</u>

1.Visualizing New Test Images: This part visualizes a batch of test images without labels.



```python
# Visualize test images

from tensorflow.python.ops.numpy_ops import np_config
np_config.enable_numpy_behavior()

for data in inf_images.take(1):
    images = data["image"]
    # imm = images.reshape(images.shape[0], (images.shape[1]*images.shape[2]))
    # imm = imm.transpose()
    # print(imm.shape)
    _, ax = plt.subplots(4, 4, figsize=(15, 8))

    # ss = plt.imshow(imm, cmap="gray")
    # plt.show()
    for i in range(16):
        img = images[i]
        # print(img.shape)
        img = tf.image.flip_left_right(img)
        img = tf.transpose(img, perm=[1, 0, 2])
        img = (img * 255.0).numpy().clip(0, 255).astype(np.uint8)
        img = img[:, :, 0]

        # Gather indices where Label!= padding token

        ax[i // 4, i % 4].imshow(img, cmap="gray")


    plt.show()
```

2. Visualizing Samples with Labels: This part visualizes a batch of training images along with their predicted labels.



```
for data in train_ds.take(1):
    images, labels = data["image"], data["label"]
    _, ax = plt.subplots(4, 4, figsize=(15, 8))

    for i in range(16):
        img = images[i]
        # print(img.shape)
        img = tf.image.flip_left_right(img)
        img = tf.transpose(img, perm=[1, 0, 2])
        img = (img * 255.0).numpy().clip(0, 255).astype(np.uint8)
        img = img[:, :, 0]

        # Gather indices where Label!= padding token
        label = labels[i]
        indices = tf.gather(label, tf.where(tf.math.not_equal(label, padding_token)))
        # Convert to string.
        label = tf.strings.reduce_join(num_to_chars(indices))
        label = label.numpy().decode("utf-8")

        ax[i // 4, i % 4].imshow(img, cmap="gray")
        ax[i // 4, i % 4].set_title(label)
        ax[i // 4, i % 4].axis("off")

    plt.show()
```

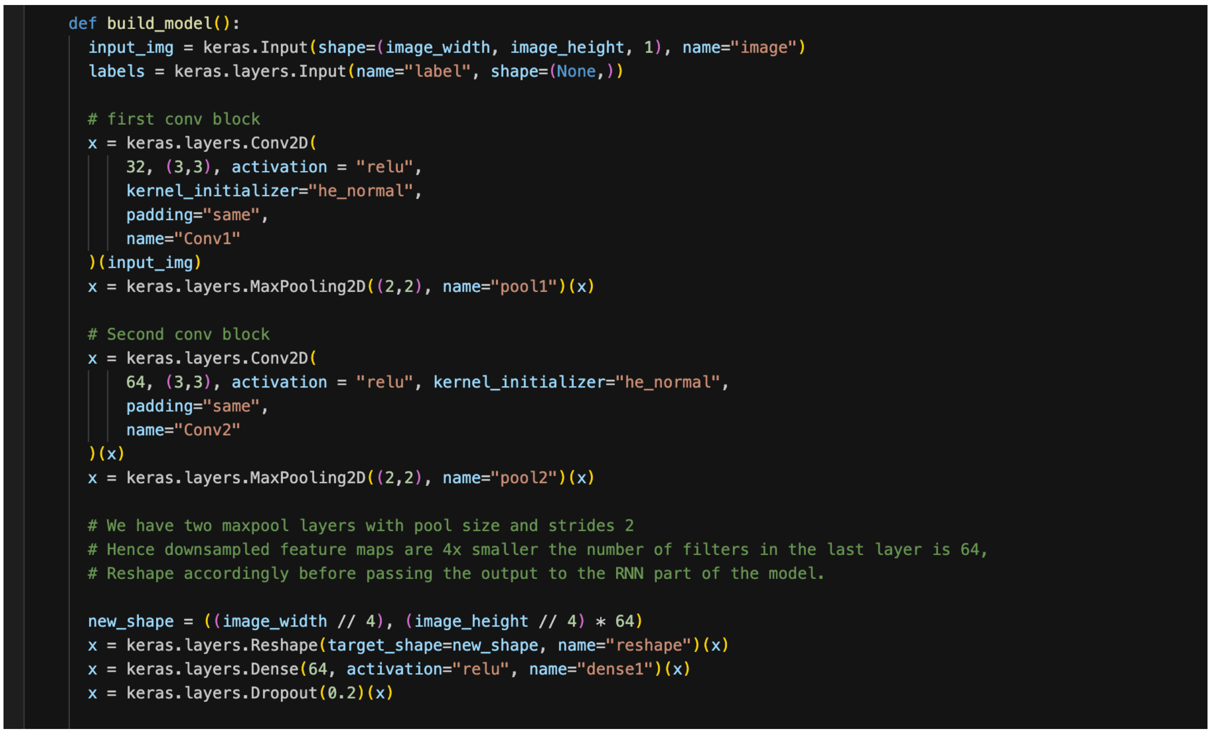
→(iv) Model Creation and Testing

Model building involves selecting appropriate algorithms, training models on preprocessed data, and evaluating performance on a test set. For handwritten recognition, deep learning models such as CNNs, RNNs, and hybrids (CNN-RNN) are commonly used.

<u>**Training Process:**</u>

Model Architecture:

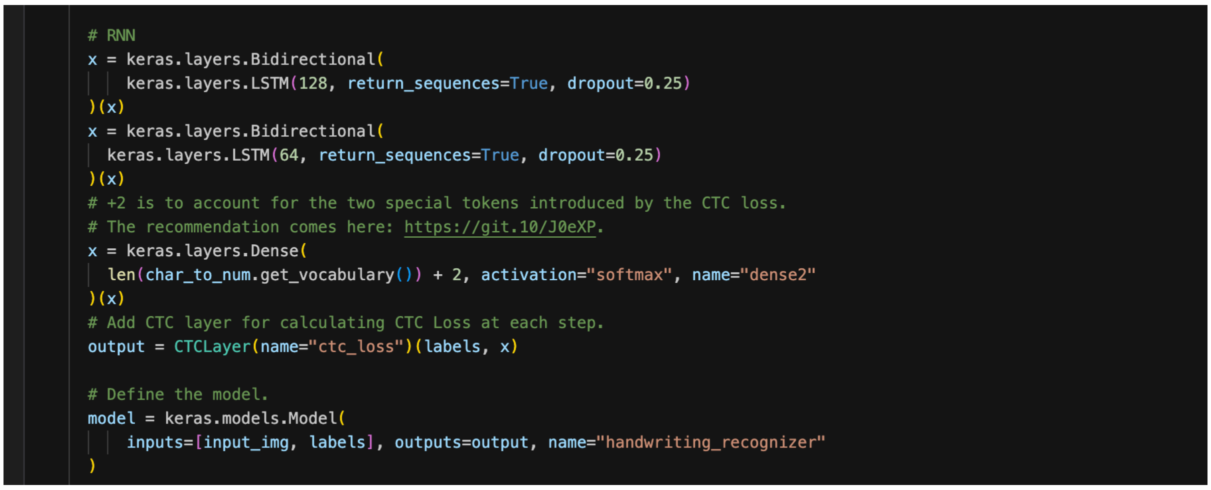
- **Convolutional Neural Networks (CNNs):**
    - **The model begins with two Convolutional Layers followed by MaxPooling layers. These are responsible for feature extraction from the input handwriting images. CNNs are highly effective in capturing spatial features such as edges and textures, which are crucial for recognizing characters in handwritten text.**



```python
def build_model():
    input_img = keras.Input(shape=(image_width, image_height, 1), name="image")
    labels = keras.layers.Input(name="label", shape=(None,))

    # first conv block
    x = keras.layers.Conv2D(
        32, (3,3), activation = "relu",
        kernel_initializer="he_normal",
        padding="same",
        name="Conv1"
    )(input_img)
    x = keras.layers.MaxPooling2D((2,2), name="pool1")(x)

    # Second conv block
    x = keras.layers.Conv2D(
        64, (3,3), activation = "relu", kernel_initializer="he_normal",
        padding="same",
        name="Conv2"
    )(x)
    x = keras.layers.MaxPooling2D((2,2), name="pool2")(x)

    # We have two maxpool layers with pool size and strides 2
    # Hence downsampled feature maps are 4x smaller the number of filters in the last layer is 64,
    # Reshape accordingly before passing the output to the RNN part of the model.

    new_shape = ((image_width // 4), (image_height // 4) * 64)
    x = keras.layers.Reshape(target_shape=new_shape, name="reshape")(x)
    x = keras.layers.Dense(64, activation="relu", name="dense1")(x)
    x = keras.layers.Dropout(0.2)(x)
```

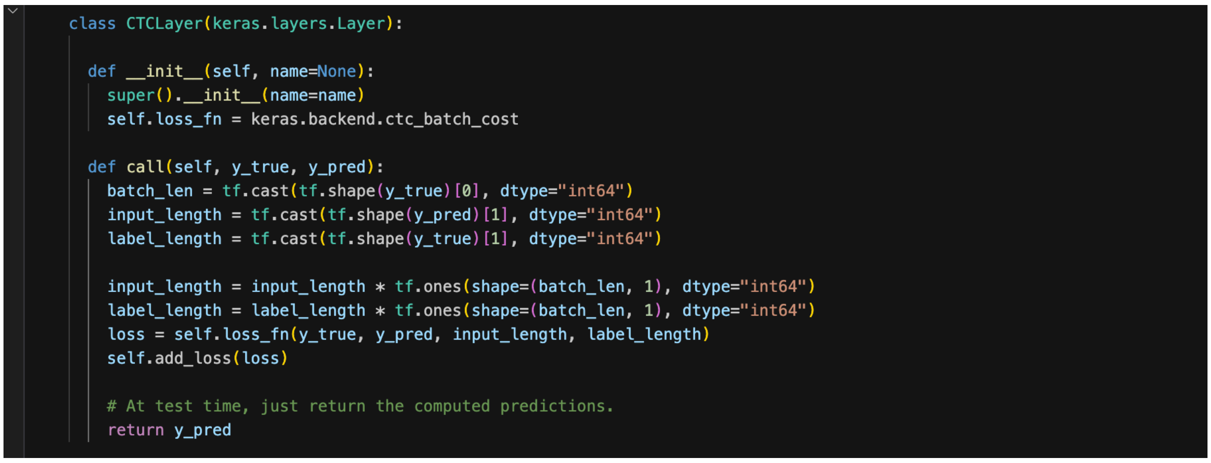
- **RNN & Bidirectional LSTM Layers:**
    - **After the CNN feature extraction, the model includes two Bidirectional LSTM (Long Short-Term Memory) layers. These LSTMs are responsible for sequence modeling, allowing the model to understand the dependencies between the characters in a word. This is essential for**

predicting sequences of characters accurately, especially in handwritten text where letters may be connected or written in a flowing style.



```python
# RNN
x = keras.layers.Bidirectional(
    keras.layers.LSTM(128, return_sequences=True, dropout=0.25)
)(x)
x = keras.layers.Bidirectional(
  keras.layers.LSTM(64, return_sequences=True, dropout=0.25)
)(x)
# +2 is to account for the two special tokens introduced by the CTC loss.
# The recommendation comes here: https://git.io/J0eXP.
x = keras.layers.Dense(
  len(char_to_num.get_vocabulary()) + 2, activation="softmax", name="dense2"
)(x)
# Add CTC layer for calculating CTC Loss at each step.
output = CTCLayer(name="ctc_loss")(labels, x)

# Define the model.
model = keras.models.Model(
    inputs=[input_img, labels], outputs=output, name="handwriting_recognizer"
)
```

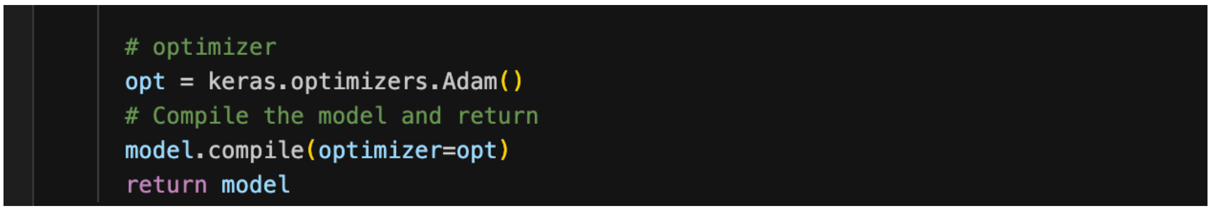
- CTC Loss Layer:
    - The final layer in the model is a CTC (Connectionist Temporal Classification) loss layer. CTC allows the model to align predicted sequences with actual labels without requiring character-level segmentation. This is ideal for sequence-to-sequence tasks like handwriting recognition, where the length of the predicted sequence may not always match the length of the actual sequence.



```python
class CTCLayer(keras.layers.Layer):

  def __init__(self, name=None):
    super().__init__(name=name)
    self.loss_fn = keras.backend.ctc_batch_cost

  def call(self, y_true, y_pred):
    batch_len = tf.cast(tf.shape(y_true)[0], dtype="int64")
    input_length = tf.cast(tf.shape(y_pred)[1], dtype="int64")
    label_length = tf.cast(tf.shape(y_true)[1], dtype="int64")

    input_length = input_length * tf.ones(shape=(batch_len, 1), dtype="int64")
    label_length = label_length * tf.ones(shape=(batch_len, 1), dtype="int64")
    loss = self.loss_fn(y_true, y_pred, input_length, label_length)
    self.add_loss(loss)

    # At test time, just return the computed predictions.
    return y_pred
```

Optimizer and Loss Function:

- **Optimizer:**
    - The Adam optimizer is used for training. It adapts the learning rate throughout the training process, making it a popular and efficient choice for deep learning models. Adam helps the model converge faster and more efficiently by adjusting the learning rate based on the first and second moments of the gradient.

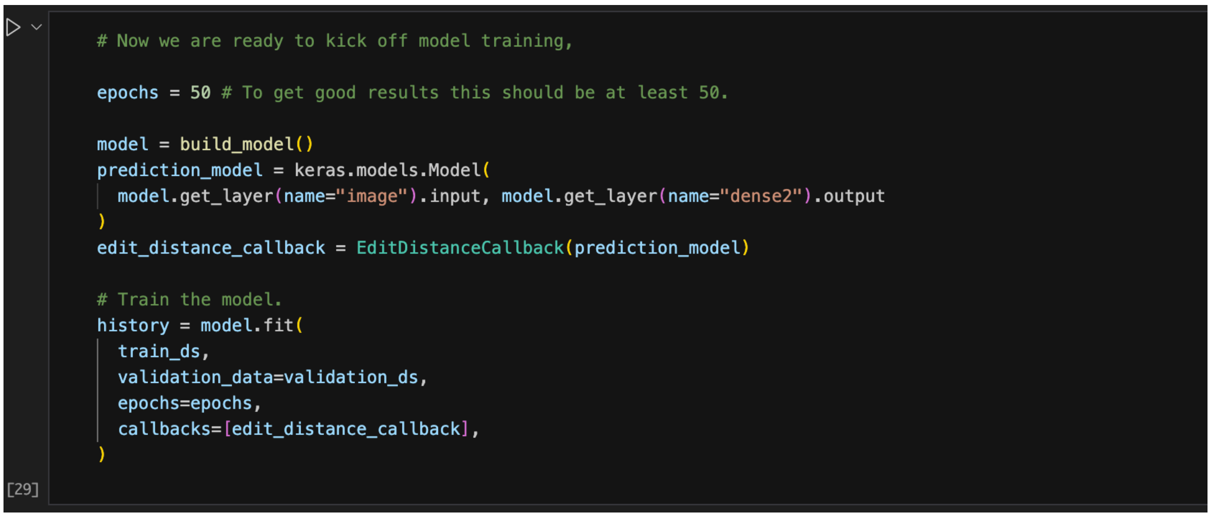


```
# optimizer
opt = keras.optimizers.Adam()
# Compile the model and return
model.compile(optimizer=opt)
return model
```

- **Loss Function:**
    - The CTC loss function is used to handle sequence prediction.

Training Configuration :

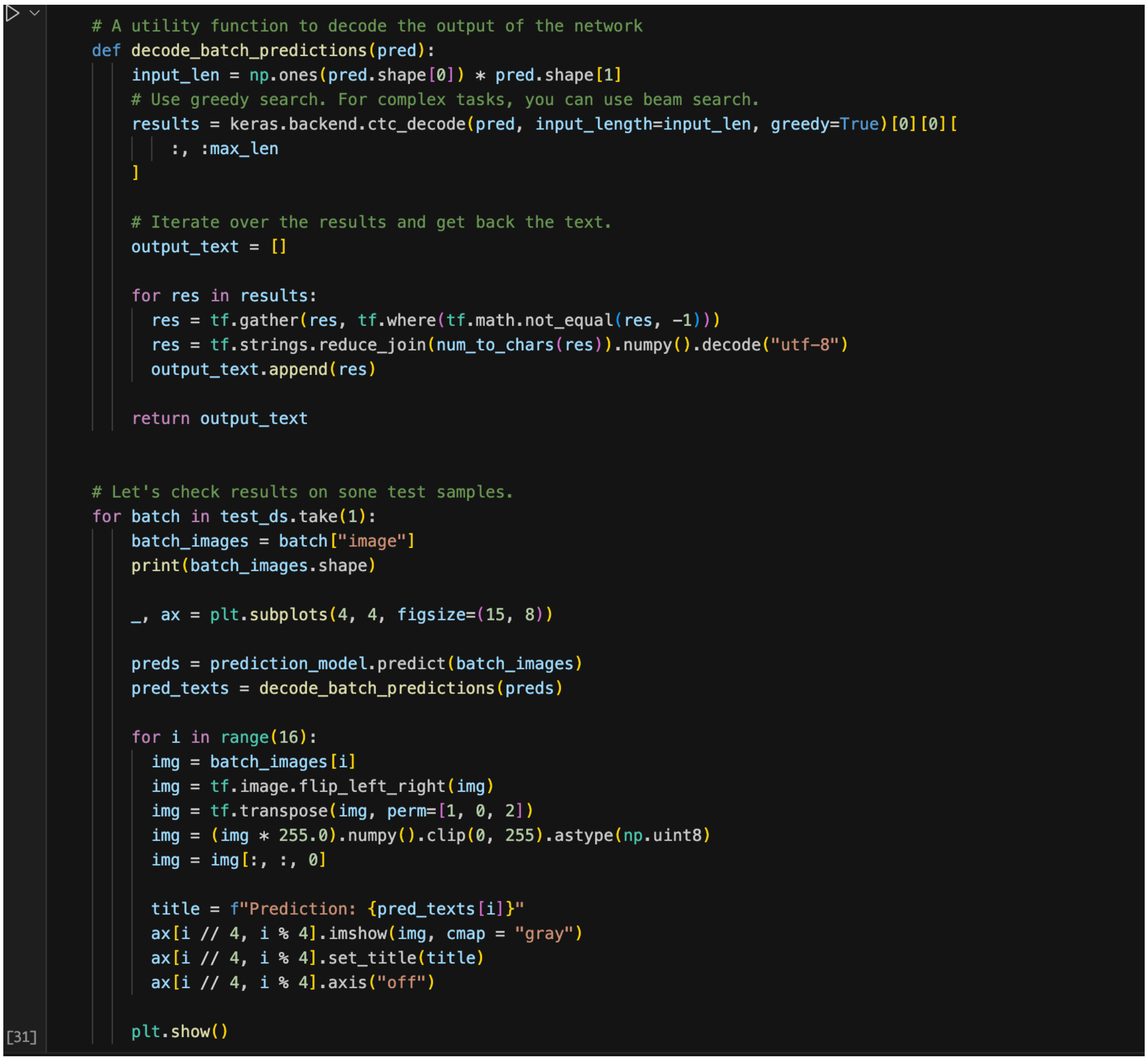
- Epochs: The model is trained for 50 epochs, providing enough time to learn meaningful patterns while balancing performance and training duration.



```
# Now we are ready to kick off model training,

epochs = 50 # To get good results this should be at least 50.

model = build_model()
prediction_model = keras.models.Model(
  model.get_layer(name="image").input, model.get_layer(name="dense2").output
)
edit_distance_callback = EditDistanceCallback(prediction_model)

# Train the model.
history = model.fit(
  train_ds,
  validation_data=validation_ds,
  epochs=epochs,
  callbacks=[edit_distance_callback],
)
```

[29]

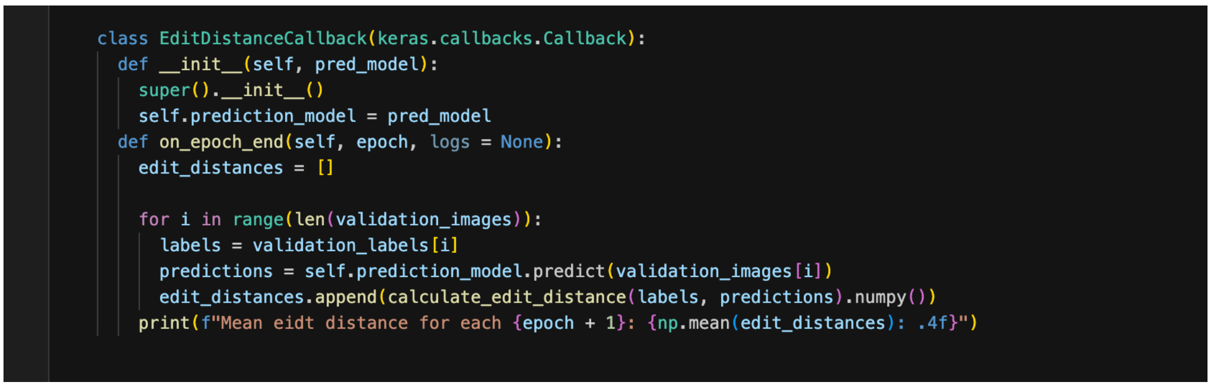
- **Batch Size: A batch size of 64 is chosen to optimize training efficiency and memory usage.**



```
# A utility function to decode the output of the network
def decode_batch_predictions(pred):
    input_len = np.ones(pred.shape[0]) * pred.shape[1]
    # Use greedy search. For complex tasks, you can use beam search.
    results = keras.backend.ctc_decode(pred, input_length=input_len, greedy=True)[0][0][
        :, :max_len
    ]

    # Iterate over the results and get back the text.
    output_text = []

    for res in results:
      res = tf.gather(res, tf.where(tf.math.not_equal(res, -1)))
      res = tf.strings.reduce_join(num_to_chars(res)).numpy().decode("utf-8")
      output_text.append(res)

    return output_text


# Let's check results on sone test samples.
for batch in test_ds.take(1):
    batch_images = batch["image"]
    print(batch_images.shape)

    _, ax = plt.subplots(4, 4, figsize=(15, 8))

    preds = prediction_model.predict(batch_images)
    pred_texts = decode_batch_predictions(preds)

    for i in range(16):
      img = batch_images[i]
      img = tf.image.flip_left_right(img)
      img = tf.transpose(img, perm=[1, 0, 2])
      img = (img * 255.0).numpy().clip(0, 255).astype(np.uint8)
      img = img[:, :, 0]

      title = f"Prediction: {pred_texts[i]}"
      ax[i // 4, i % 4].imshow(img, cmap = "gray")
      ax[i // 4, i % 4].set_title(title)
      ax[i // 4, i % 4].axis("off")

    plt.show()
```

[31]

- **Callbacks: A custom callback monitors edit distance at the end of each epoch, ensuring the model's character-level accuracy and tracking progress.**

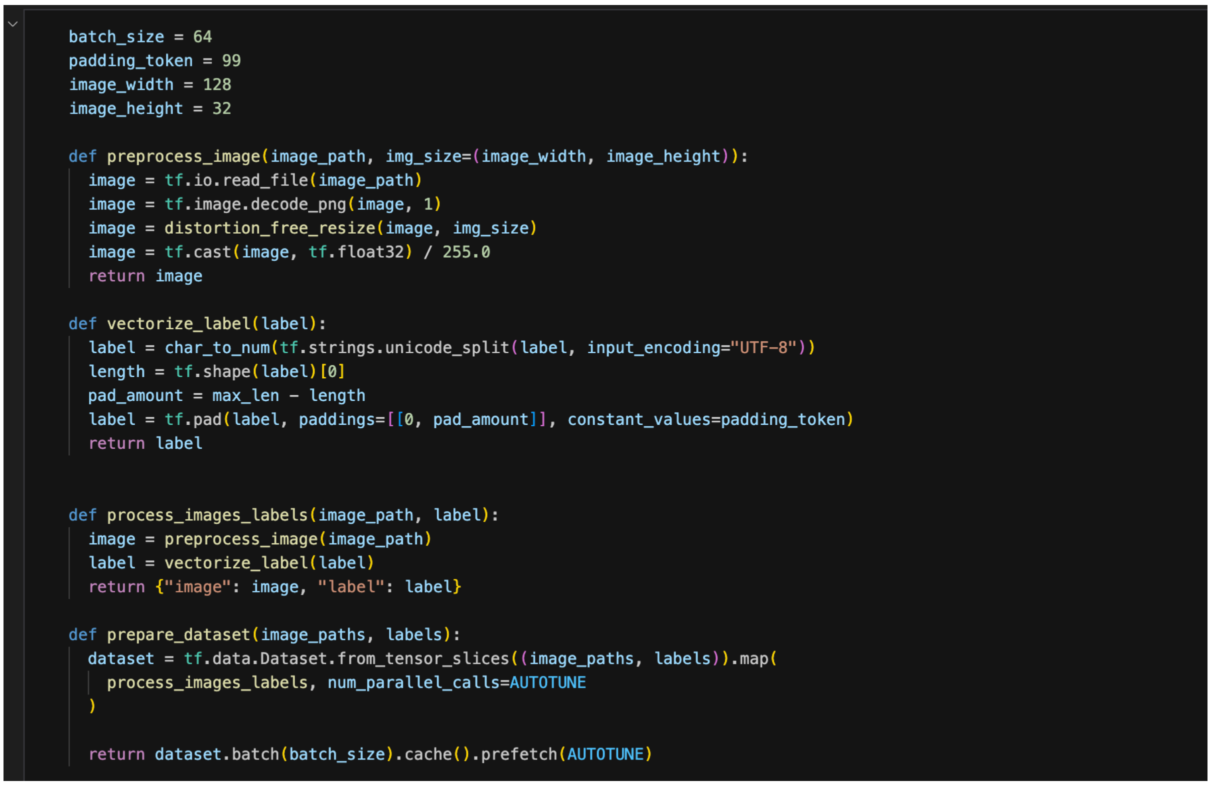


```
class EditDistanceCallback(keras.callbacks.Callback):
  def __init__(self, pred_model):
    super().__init__()
    self.prediction_model = pred_model
  def on_epoch_end(self, epoch, logs = None):
    edit_distances = []

    for i in range(len(validation_images)):
      labels = validation_labels[i]
      predictions = self.prediction_model.predict(validation_images[i])
      edit_distances.append(calculate_edit_distance(labels, predictions).numpy())
    print(f"Mean eidt distance for each {epoch + 1}: {np.mean(edit_distances): .4f}")
```

<u>Testing Process</u>

Test Image Preprocessing:

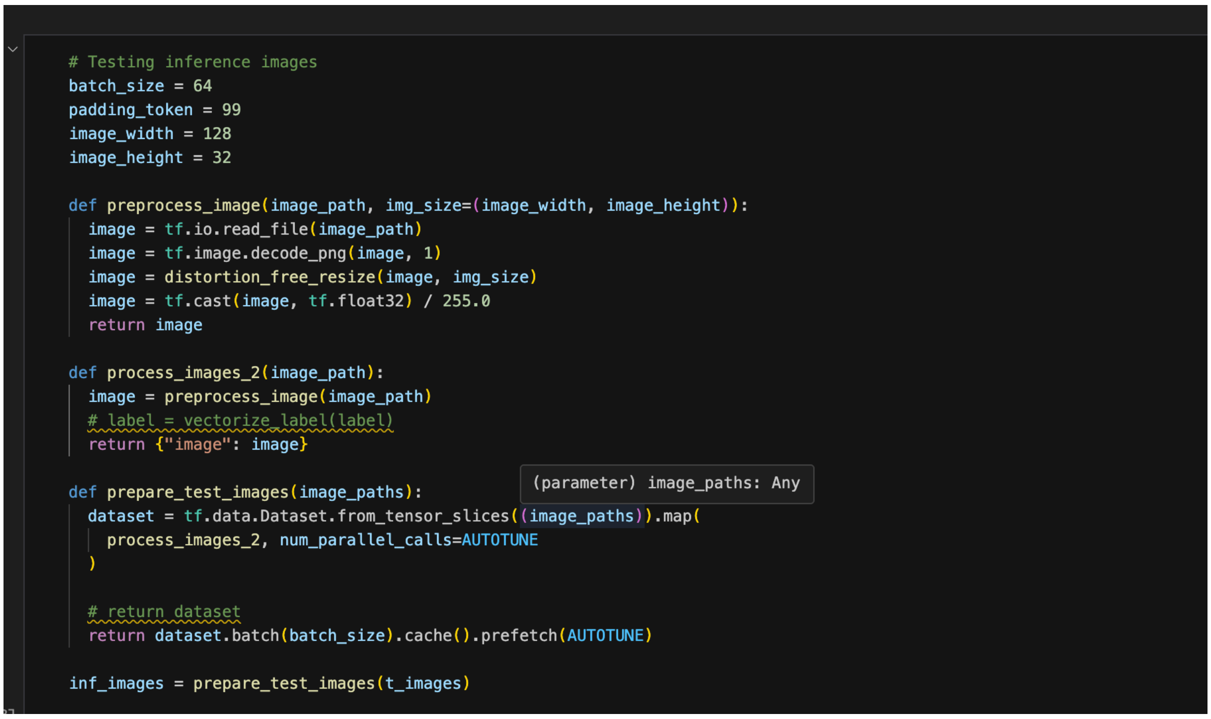
- **The test images are preprocessed in the same way as the training images. This includes:**
  - **Resizing each image to 128x32 pixels while maintaining the aspect ratio to prevent distortion.**
  - **Normalization of pixel values by scaling them between 0 and 1 (dividing by 255) to make the model's training more stable and consistent with the test data.**



```python
batch_size = 64
padding_token = 99
image_width = 128
image_height = 32

def preprocess_image(image_path, img_size=(image_width, image_height)):
  image = tf.io.read_file(image_path)
  image = tf.image.decode_png(image, 1)
  image = distortion_free_resize(image, img_size)
  image = tf.cast(image, tf.float32) / 255.0
  return image

def vectorize_label(label):
  label = char_to_num(tf.strings.unicode_split(label, input_encoding="UTF-8"))
  length = tf.shape(label)[0]
  pad_amount = max_len - length
  label = tf.pad(label, paddings=[[0, pad_amount]], constant_values=padding_token)
  return label


def process_images_labels(image_path, label):
  image = preprocess_image(image_path)
  label = vectorize_label(label)
  return {"image": image, "label": label}

def prepare_dataset(image_paths, labels):
  dataset = tf.data.Dataset.from_tensor_slices((image_paths, labels)).map(
    process_images_labels, num_parallel_calls=AUTOTUNE
  )

  return dataset.batch(batch_size).cache().prefetch(AUTOTUNE)
```

Inference:

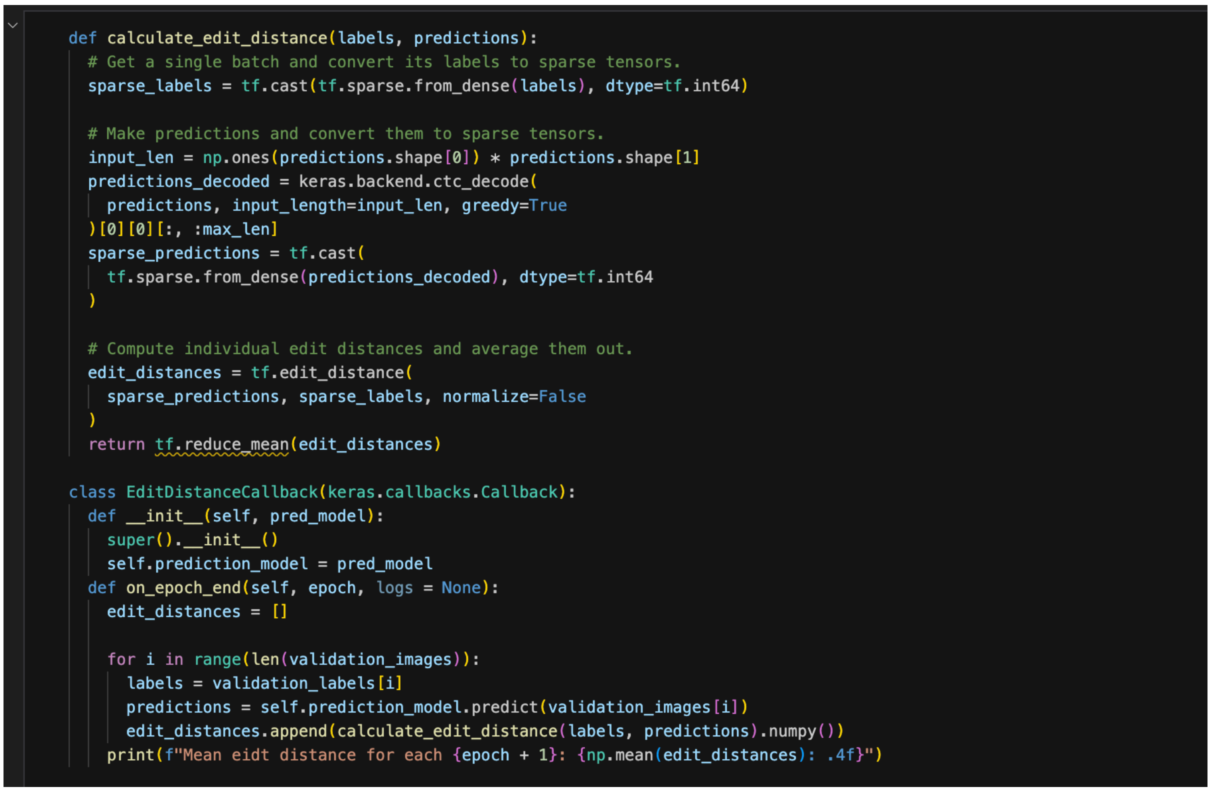
- **During inference, the trained model predicts a sequence of characters for each test image. The output from the model is passed through the CTC layer to decode the predicted sequence. This step produces the most likely sequence of characters for each handwritten word or line in the test images.**



```
# Testing inference images
batch_size = 64
padding_token = 99
image_width = 128
image_height = 32

def preprocess_image(image_path, img_size=(image_width, image_height)):
    image = tf.io.read_file(image_path)
    image = tf.image.decode_png(image, 1)
    image = distortion_free_resize(image, img_size)
    image = tf.cast(image, tf.float32) / 255.0
    return image

def process_images_2(image_path):
    image = preprocess_image(image_path)
    # label = vectorize_label(label)
    return {"image": image}

def prepare_test_images(image_paths):           (parameter) image_paths: Any
    dataset = tf.data.Dataset.from_tensor_slices((image_paths)).map(
        process_images_2, num_parallel_calls=AUTOTUNE
    )

    # return dataset
    return dataset.batch(batch_size).cache().prefetch(AUTOTUNE)

inf_images = prepare_test_images(t_images)
```

Evaluation Metrics:

- **Edit Distance:**
    - **Edit distance is used to evaluate the model's performance on the test set. This metric calculates the number of edits (insertion, deletion, substitution) required to transform the predicted sequence into the correct sequence (ground truth). A lower edit distance indicates better performance and higher accuracy in recognizing handwritten text.**



```python
def calculate_edit_distance(labels, predictions):
    # Get a single batch and convert its labels to sparse tensors.
    sparse_labels = tf.cast(tf.sparse.from_dense(labels), dtype=tf.int64)

    # Make predictions and convert them to sparse tensors.
    input_len = np.ones(predictions.shape[0]) * predictions.shape[1]
    predictions_decoded = keras.backend.ctc_decode(
        predictions, input_length=input_len, greedy=True
    )[0][0][:, :max_len]
    sparse_predictions = tf.cast(
        tf.sparse.from_dense(predictions_decoded), dtype=tf.int64
    )

    # Compute individual edit distances and average them out.
    edit_distances = tf.edit_distance(
        sparse_predictions, sparse_labels, normalize=False
    )
    return tf.reduce_mean(edit_distances)

class EditDistanceCallback(keras.callbacks.Callback):
    def __init__(self, pred_model):
        super().__init__()
        self.prediction_model = pred_model
    def on_epoch_end(self, epoch, logs = None):
        edit_distances = []

        for i in range(len(validation_images)):
            labels = validation_labels[i]
            predictions = self.prediction_model.predict(validation_images[i])
            edit_distances.append(calculate_edit_distance(labels, predictions).numpy())
        print(f"Mean eidt distance for each {epoch + 1}: {np.mean(edit_distances): .4f}")
```

- **CTC Loss:**
    - **The CTC loss is also calculated during testing to evaluate the model's performance in aligning the predicted and actual sequences.**

By following these steps, an AI-based handwritten recognition system can achieve high accuracy and robustness, effectively transforming handwritten text into digital form for various applications. This approach leverages a combination of data preprocessing, feature extraction, and state-of-the-art model architectures to address the challenges of handwritten text recognition.
