Deep Learning Final Project : Hangul OCR Using Jamoembedding

Course : 2023-1 DLIP Final project

Student ID : 21800031

Name : Kook Jinho

Date : 2023-06-16

Reference github link : [craftsangjae/ocr-text-recognition-korean: 한글 단어 혹은 문장 이미지를 받아 텍스트를 반환하는 Text Recognition Model (github.com)]

I. Introduction

1. Why does this topic need to be addressed?

- Individuals often need to digitally document images of test questions, etc., but there are few options for doing so, so images are often typed by hand.
- There are many OCR models for English, but the number of models for Korean is very small and the accuracy is not good.
- A large amount of intellectual property held by public enterprises can be preserved by digitally documenting it.
- Easy access to information for individuals.

2. Prior research on this topic

- There are currently very few Hangul OCR models, and most of them are already optimized for English. In terms of difficulty, Hangul is very difficult because consonants and vowels are combined in two dimensions, vertically and horizontally, while English letters are a one-dimensional array that stretches sideways.
- The most representative Hangul OCR is 'easyOCR', which was launched in 2020 and is still being updated. However, the general consensus is that its performance is much worse than English.

3. Overview of current research on Hangul OCR

1. Recognition algorithms
    - For OCR, it is separated into text detection and text recognition. In general, the algorithms used for each are different. For example, FOTS is used for text detection and RCNN is used for text recognition.
    - In this project, I focus on text recognition and apply the SRN model.
2. Recognized fonts
    - They can be broadly categorized as printed and handwritten. In the case of print, the distinction can be made based on the font. In the case of handwriting, the learning performance is very poor unless it is syllable-by-syllable.
    - In this project, we aim to distinguish between the 'printed' and 'handwritten' fonts of 'NAVER Shared Font'.
3. Recognition target classification
    - There are two main types of recognition objects. Syllables, or more than syllables. Currently, the recognition rate for syllables is quite high.
    - It also depends on whether the object to be recognized is unstructured (such as a photo of a signboard) or structured (such as a document image rather than an actual scanned photo). The accuracy of Hangul for unstructured photo images is currently very low and difficult to use (when text detection and text recognition are performed simultaneously), and the number of related papers is very small and the evaluation methods are different, so it cannot be said to be highly reliable, but it is said to be around 50% for unstructured photos[1].
    - If you randomly detect text in 'relatively' structured scanned documents (such as prescription photos scanned with a camera) and use a commercial API for OCR recognition, the accuracy rate is close to the mid-to-high 90%. As a representative, NAVER's 'Clover' is 95.7%, and 'easyOCR', one of the open sources, is 83.6% [2].
    - Since this project focuses on text recognition, it is based on images that have a high degree of text recognition in structured documents. It is also word-based.

4. What I want to accomplish with this project

- Achieved Hangul recognition accuracy of over 90%.
- Increase the dataset and modify the hyper-parameters to perform better on the existing studied models.
- Evaluate and output directly written characters through 'OPENCV' image processing
- Extracted better performance than easyOCR, a representative free OCR program

5. Introduction to Keywords

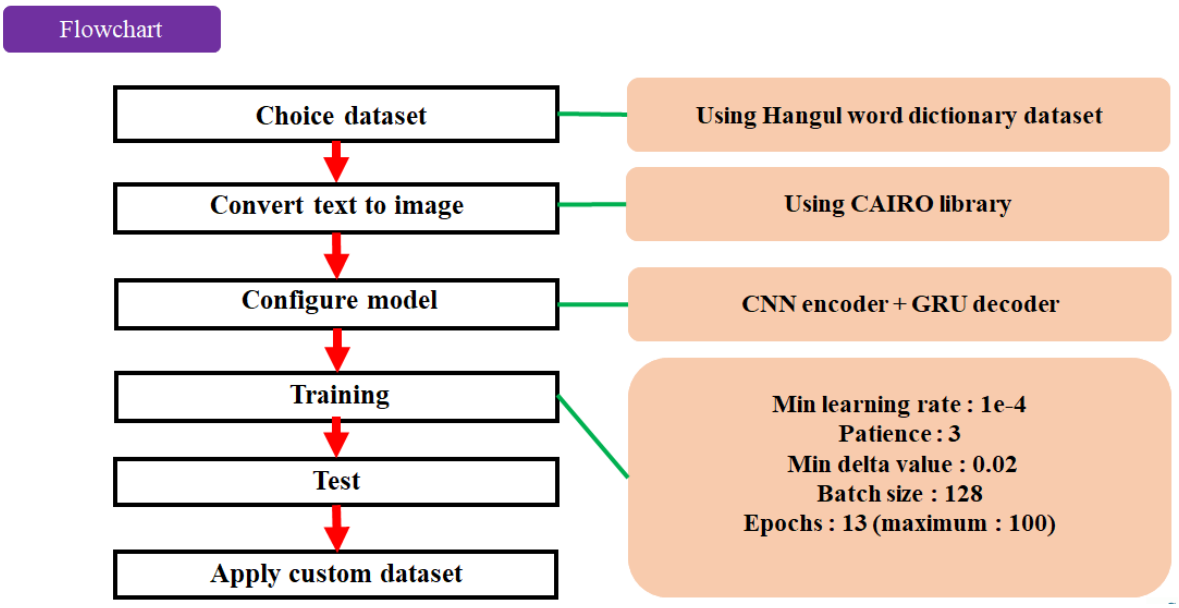
- Hangul OCR is divided into two main steps. text detection and text recognition.
    - text detection: determining the position of letters.
    - text recognition: reading sequentially placed characters.
        - Example: If you see an image of a sign with Korean characters on it, text detection is to distinguish what is Korean and what is not, and text recognition is to read the location of the Korean characters on the sign.

- Jamoembedding : Unlike other languages, Hangul is composed of a combination of consonants and vowels. Unlike English, where a single character occupies a single array, the array and meaning change depending on the position of the initial, neutral, and final vowels, so it is necessary to decompose and combine the letters to understand the meaning.In particular, there are more than 2,000 classes for the letters themselves, but if you separate them into consonants and vowels, the number of classes is definitely reduced to 100 or less. This makes it more useful for smaller datasets.

- SRN model : Structure of CRNN + Seq2Seq + Attention Network
    - SRN's model configuration is the same as CRNN.
    - This project uses a CNN encoder + GRU decoder model as a type of SRN model.

6. Overview

- This is flow chart. Following this flowchart, I have detailed the procedure for the project in the procedure below.



II. Procedure

- If you want to skip all that and just run the code, you can run 'main.ipynb' in the 'ocr-text-recognition-korean-master' folder. The file is in Jupyter format so you can run it step-by-step.
- **All of these procedures are well documented in 'main.pyinb' in the 'ocr-text-recognition-korean-master' folder.**
- ***WARNING !***

- When you run the file, it will set aside variables and allocate VRAM and CUDA as it trains with the GPU, so be careful not to exit the file or do anything else that uses VRAM, as this could cause the train data to be lost.
- If you want to run the code again, or reallocate VRAM and CUDA, you'll need to do a "restart". Be careful, all your work will be lost.
    - Make sure to set the path to the file correctly for your own folder.
    - If you don't run the blocks in order, variables can be updated and cause errors, so consider how they relate to other code blocks.

1. Environment setting

First, you have to set virtual enviroment that I provided and install requirements. But I provided yaml file that includes setting value. You just use it in ANACONDA.

Follow this:

```
conda env create -f DLIP_GPU.yaml
conda activate DLIP_GPU
```

- This format is optimized for CUDA 11.3 and PYTHON 3.9. Anything other than this format may cause errors.
- If you want a different format, create new enviroment(PYTHON), **install 'requirements.txt' and install the other necessary files.**
- tensorflow 1.X is not recommand. And if you use tensorflow cpu version, train speed is very low.

My environment specification

- GPU : RTX 3060ti
- VRAM : 8GB
- GPU capability : 8.6
- cuda version : 11.3
- python version : 3.9.16
- pytorch version : 1.10.1
- cudnn : 8.0
- tensorflow-gpu version : 2.6.0

2. Datasets

- Adds a font in ttf format to font. It is currently applied as 'Nanum Gothic' font. If the font is not supported by CAIRO, the image may be broken.
- Add a Korean word dictionary to dicts.
- Apply the desired type (distortion, blur, etc.). Currently, there are 5 types, and you can add gap, self-destruct, etc.
- example images below

챌린지

에엠지

(1) Font setting

- Set 'Nanum Gothic' font
- Unfortunately, the number of fonts supported by CAIRO is very small, so if you want to use other fonts that are not supported by CAIRO, you will need to use a different dataset.

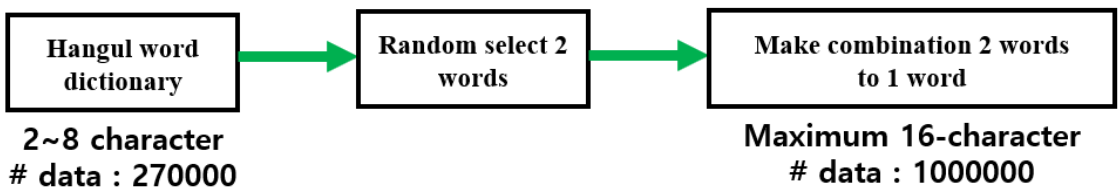
```
# Only use Nanum font. the other fonts make error.
FONT_LIST = [f.name for f in fm.fontManager.ttflist]
FONT_LIST = list(
   set([f for f in FONT_LIST if "Nanum" in f ]))
print(FONT_LIST)
```

(2) data generator setting

- Set font size(24), various noise(bg_noise=0.02,affine_noise=(0.0,0.02))
- If you want color image, you can change 'gray_scale' to 'False'
- font list is set by upper 'font setting'

```
# OCRDataset setting
# dataset consist in only 1 font.
OCRDataset = partial(OCRDataset,
                     font_size=24,
                     bg_noise=0.02,
                     affine_noise=(0.0,0.02),
                     normalize=False,
                     random_shift=False,
                     color_noise=(0.0,0.02),
                     gray_scale=True,
                     font_list=FONT_LIST,
                     )
```

(3) Data augmentation



- This code randomly picks two words from the dictionary and combines them to augment the data.

```
#data augmentation
import random
```

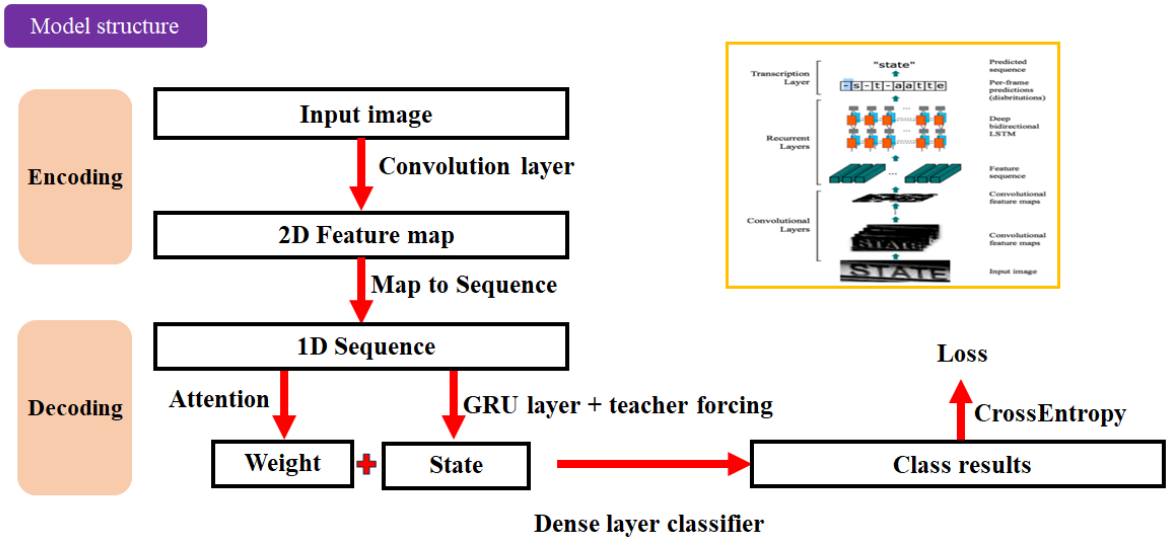
```
def concatenate_words(words):
    concatenated_list = []
    for _ in range(1000000):
        random_index1 = random.randint(0, len(words) - 1)
        random_index2 = random.randint(0, len(words) - 1)
        random_words = [words[random_index1], words[random_index2]]
        concatenated_word = ''.join(random_words)
        concatenated_list.append(concatenated_word)
    return concatenated_list

new_list = concatenate_words(words)
```

3. Models

(1) Model structure

- This model is largely divided into two stages. the encoder and decoder parts.
- The encoder portion extracts a feature map using CNN.
- However, to separate Hangul into consonants and vowels, change it to 1D scale. And I proceed with decoding using GRU.
- Attention refers once again to the input word. At this point, focus on the relevant parts and give weight.
- Eventually, the class is classified by combining the results. And calculate loss.



(2) Model code

- Current best model : batch size : 128, lettering batch : 4 , step epoch = 1000, number of maximum epochs : 100, real epochs : 17, loss : 0.07

```
# 1. Configure Image Feature Extractor
from models.layers import ResidualConvFeatureExtractor
from tensorflow.keras.layers import Input
```

```
K.clear_session()
height = 36
n_conv = 16   # the number of Convolution filter
n_state = 256 # the number of GRU units
n_embed = 16  # the size of embedding vector
n_clf = 256   # the number of units in classifier Dense layer
# Image Encoder
inputs = Input(shape=(height, None, 1),name='images')
conv_maps = ResidualConvFeatureExtractor(n_conv,
                                         name='feature_extractor')(inputs)

# 2. Configure Map2Sequence
from models.layers import Map2Sequence
feature_seqs = Map2Sequence(name='map_to_sequence')(conv_maps)


# 3. Configure Embedding Layer
from models.layers import JamoEmbedding, TeacherForcing
# TeacherForcing training
y_true = Input(shape=(None,), dtype=tf.int32, name='output_sequences')
decoder_inputs = TeacherForcing()(y_true)

jamo_embedding_layer = JamoEmbedding(n_embed)
jamo_embedding = jamo_embedding_layer(decoder_inputs)



# 4. Configure Text Decoder Layer
from tensorflow.keras.layers import GRU
# Text Decoder
gru_layer = GRU(n_state,
                name='decoder_gru',
                return_sequences=True)
states_decoder = gru_layer(jamo_embedding)



# 5. Configure Attention Layer
from models.layers import DotAttention
from tensorflow.keras.layers import Concatenate

dotattend = DotAttention(n_state=n_state)
context, attention = dotattend([feature_seqs, states_decoder])

concat_state = Concatenate(axis=-1, name='concat_output')(
    [context, states_decoder])



# 6. Configure Character Classification Layer
from models.layers import JamoClassifier
# Classifier Layer
jamo_clf = JamoClassifier(n_clf, name='output_seqs')
y_pred = jamo_clf(concat_state)



# 7. Configure Loss Layer
```

```
from models.losses import JamoCategoricalCrossEntropy
loss = JamoCategoricalCrossEntropy(blank_value=-1.)([y_true, y_pred])



# 8. Configure Train Model
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
# For training
trainer = Model([inputs, y_true],[loss], name='trainer')
trainer.add_loss(loss)
# Compile model
trainer.compile(Adam(lr=1e-2))
```

4. Training

- This part represents the training code.
- The training code utilizes the GPU by default, and the maximum number of epochs is 100.
- By enabling early stopping, the code will stop when the number of times the loss does not decrease below a certain range has accumulated.
- At this point, you should check the final loss, as it may stop even if the training is not successful. The value of the final loss should be less than or equal to 0.5.

(1) training code

```
from tensorflow.keras.callbacks import ReduceLROnPlateau
from multiprocessing import cpu_count
from utils.generator import JAMOSeq2SeqGenerator
from tensorflow.keras.callbacks import EarlyStopping

callbacks =[]
rlrop = ReduceLROnPlateau(
    factor=0.5, patience=2,
    min_lr=1e-5, verbose=1,cooldown=5, monitor='loss')
callbacks.append(rlrop)

#set epoch number
epochs = 100

# I recommand batch size number over 64
train_gen = JAMOSeq2SeqGenerator(train_set.config, batch_size=128)
valid_gen = JAMOSeq2SeqGenerator(valid_set.config, batch_size=128)

# Earystopping code on loss
early_stopping = EarlyStopping(patience=2, monitor='loss',mode='min',
min_delta=0.02, verbose=1)
callbacks.append(early_stopping)
```

```
#train model
hist = trainer.fit(train_gen, steps_per_epoch=1000,

                                epochs=epochs, callbacks=callbacks,
                                use_multiprocessing=True,
                                workers=cpu_count())
```

(2) train history plot code

- This code graphs the loss. It uses the 'matplot' library.

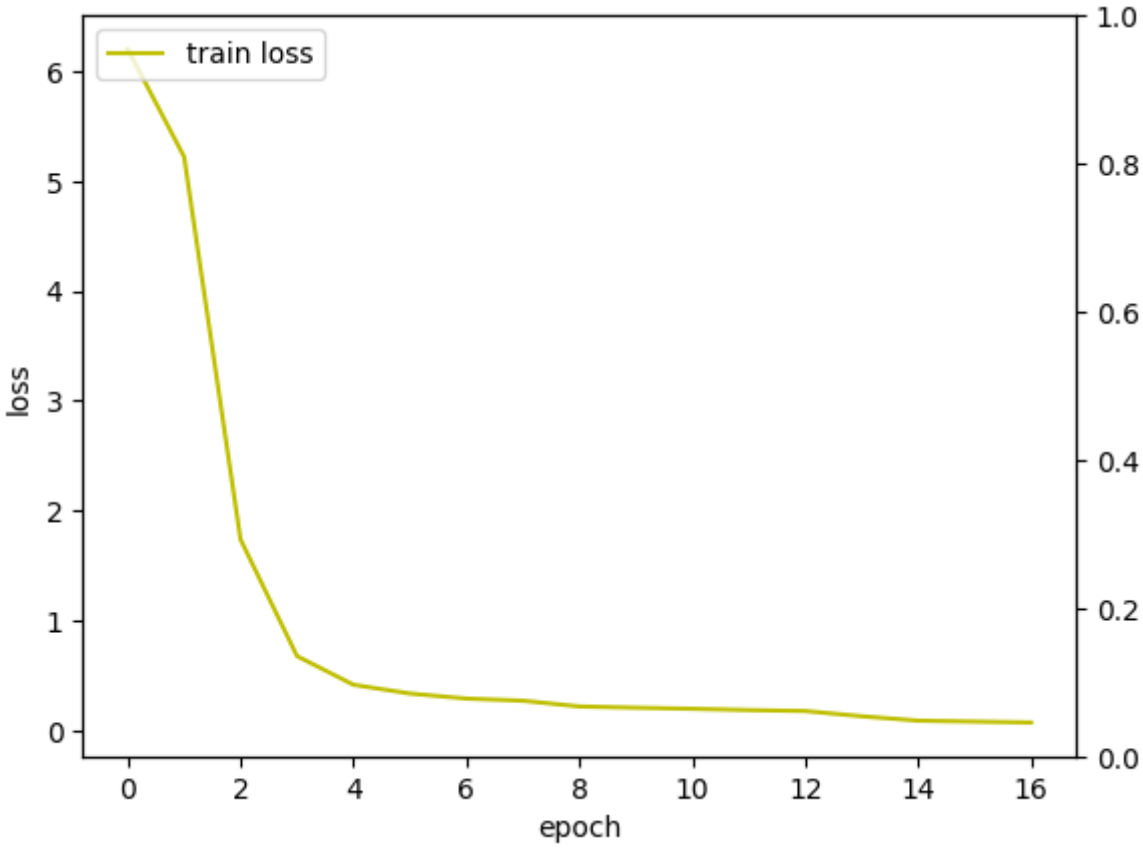
```
# plot progress of training

fig, loss_ax = plt.subplots()
acc_ax = loss_ax.twinx()
loss_ax.plot(hist.history['loss'], 'y', label='train loss')
loss_ax.set_xlabel('epoch')
loss_ax.set_ylabel('loss')
loss_ax.legend(loc='upper left')
plt.show()
```

- The image below shows the progression of an actual epoch. It takes a lot of time, many hours or more.

```
Epoch 1/100
1000/1000 [==============================] - 383s 330ms/step - loss: 6.2008
Epoch 2/100
1000/1000 [==============================] - 582s 543ms/step - loss: 5.2203
Epoch 3/100
1000/1000 [==============================] - 442s 402ms/step - loss: 1.7354
Epoch 4/100
1000/1000 [==============================] - 320s 282ms/step - loss: 0.6734
Epoch 5/100
1000/1000 [==============================] - 324s 288ms/step - loss: 0.4124
Epoch 6/100
1000/1000 [==============================] - 356s 319ms/step - loss: 0.3326
Epoch 7/100
1000/1000 [==============================] - 422s 382ms/step - loss: 0.2868
Epoch 8/100
1000/1000 [==============================] - 442s 402ms/step - loss: 0.2672
Epoch 9/100
1000/1000 [==============================] - 459s 420ms/step - loss: 0.2149
Epoch 10/100
1000/1000 [==============================] - 534s 495ms/step - loss: 0.2035
Epoch 11/100
1000/1000 [==============================] - 419s 380ms/step - loss: 0.1938
Epoch 12/100
1000/1000 [==============================] - 441s 402ms/step - loss: 0.1818
Epoch 13/100
...
1000/1000 [==============================] - 722s 681ms/step - loss: 0.0773
Epoch 17/100
1000/1000 [==============================] - 553s 511ms/step - loss: 0.0688
Epoch 00017: early stopping
```

- In my experience, the optimal number of epochs is less than 25.
- In my experience, 10 to 25 epochs is the best learning curve. This can take up to 5 hours of training time.
- The graph below is an actual learning outcome.



5. Test text recognition

(1) infer model for dictionary

- The code below shows the process of encoding and decoding.
- mounting data

```
X, = valid_gen[0]
print(len(valid_gen[0]))
# Target image
target_images = X['images'][:10]
print(target_images.shape)
```

- encoding setting

```
from models.layers import JamoCompose

# - (1) Encoder
encoder = Model(inputs, feature_seqs,
                name='encoder')
num_feature = feature_seqs.shape.as_list()[-1]
```

- decoding setting

```
# - (2) Decoder
states_encoder_input = Input((None, num_feature),
                             name='states_encoder_input')
decoder_state_inputs = Input(shape=(n_state,), name='decoder_state')
decoder_inputs = Input(shape=(None,), name='decoder_inputs')

jamo_embedding = jamo_embedding_layer(decoder_inputs)
```

```
states_decoder = gru_layer(jamo_embedding,
                           initial_state=decoder_state_inputs)
context, attention = dotattend([states_encoder_input,
                                states_decoder])
concat_state = Concatenate(axis=-1,name='concat_output')(
    [context, states_decoder])
predictions = jamo_clf(concat_state)
output_decoder = JamoCompose(name='jamocompose')(predictions)

decoder = Model([states_encoder_input,
                 decoder_inputs,
                 decoder_state_inputs],
                [states_decoder, output_decoder], name='decoder')
```

- encoding and decoding

```
# Calculate encoder results
states_encoder_ = encoder.predict(target_images)

EOS_TOKEN = ord('\n') # <EOS> token

# Decoder
batch_size = target_images.shape[0]

prev_inputs = np.ones((batch_size,1)) * EOS_TOKEN
prev_states = np.zeros((batch_size, n_state))

result = np.zeros([batch_size,0])

while True:

    states_decoder_, predictions_ = decoder.predict({
        "states_encoder_input" : states_encoder_,
        "decoder_inputs": prev_inputs,
        "decoder_state": prev_states
    })
    prev_states = states_decoder_[:,-1,:]
    prev_inputs = predictions_

    result = np.concatenate([result, prev_inputs], axis=-1)

    print(result)

    if np.all(np.any(result==EOS_TOKEN,axis=1)):
        break

result = result[:,:].squeeze()
result = result.astype(np.int)
```

(2) show image code

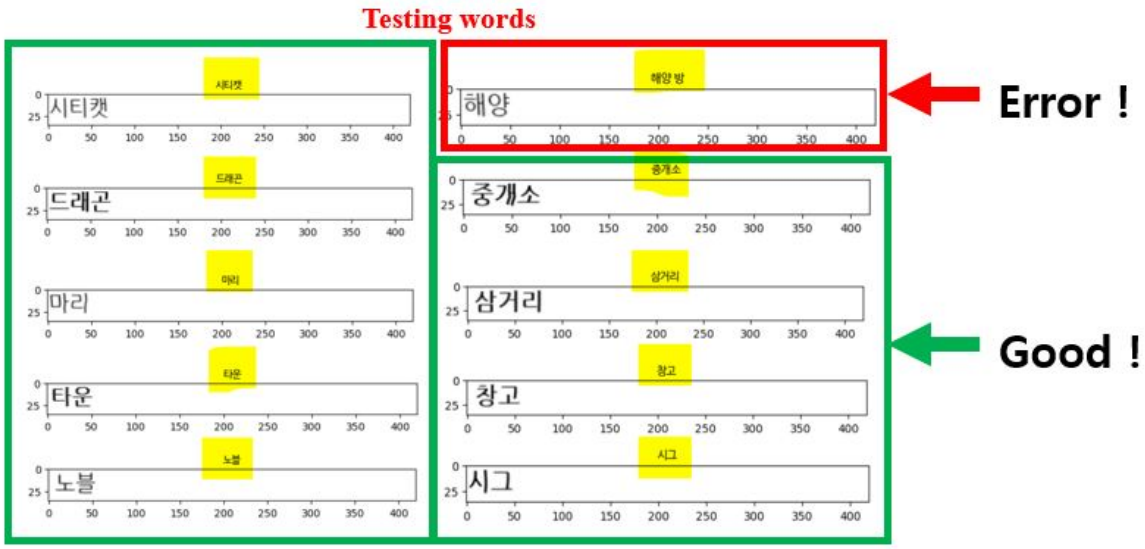
- This code shows the image and the prediction results.

```
for image, seq in zip(target_images, result):

    #print(seq)
    text_seq = "".join([chr(char) for char in seq])
    print(len(text_seq))
    text_seq = text_seq.split('\n')[0]

    plt.title(text_seq,fontproperties=fontprop)
    plt.imshow(image[:,:,0],cmap='gray')
    plt.show()
```

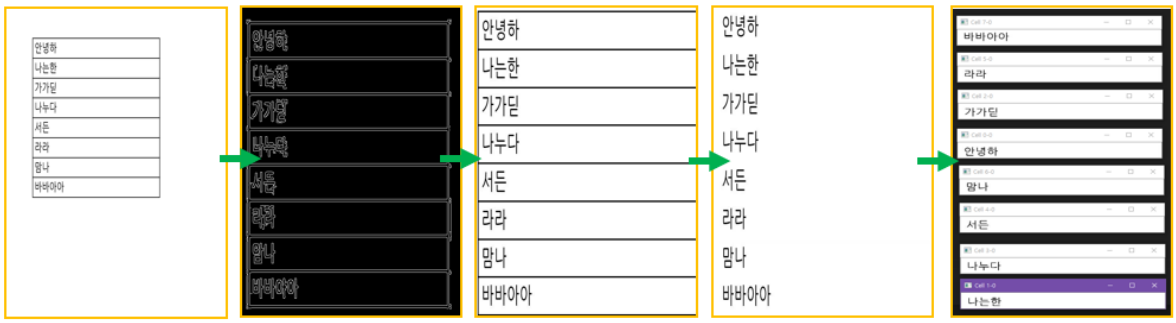
- The result examples represented below. '해양' is not recognized correctly.



6. OPENCV Text detection + Text recognition

(1) OPENCV text detection

- To implement the flow of text detection and text recognition, I went through a simple text detection process with opencv.
- Write the letters in the table in a Word file, then use image processing to extract just the letters.
- The image is then modified to fit the model format and evaluated with the trained model.
- The image below is an easy-to-understand illustration of this process.



Raw image | Canny image | Line recognition image | Eliminated line image | Cropped image for format

(2) OPENCV text detection code

- This code is image processing code that performs text detection. It is divided into two parts.
- The first part detects the outline of the entire table, and the second part cuts each word and saves it as an image.

```
import cv2
import numpy as np

# 1st image processing
# This process cuts the outline of the image.

# Read image
image =
cv2.imread('C:/Users/ririk/Desktop/DLIP_Final_Project/test_image/TEST20.JPG')

# Structuring element kernel, a rectangle (3x3) is created ---①
k = cv2.getStructuringElement(cv2.MORPH_RECT, (3,3))
# Apply erosion operation ---②
image_copy = cv2.erode(image, k)

# Canny edges
edges_1st = cv2.Canny(image_copy, 0, 250)

cv2.imshow("OutlineImage", edges_1st)
cv2.waitKey(0)


# Detect contours
contours, _ = cv2.findContours(edges_1st, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)

if contours:
    # Cut outline
    max_contour = max(contours, key=cv2.contourArea)

    # Calculate positions of the outline box
    x, y, w, h = cv2.boundingRect(max_contour)

    # Cut for outline
    cropped_image = image[y:y + h, x:x + w]

    # Print image
    cv2.imshow("OutlineImage", cropped_image)
    cv2.waitKey(0)
```

```
else:
    print("Contour not found.")



# ----------------------------------------------------------------------
# 2nd image processing
# This process cuts the image for word boxes.

# Canny edge detection for edges
edges_2nd = cv2.Canny(cropped_image, 50, 150)

# Detect lines using HoughLinesP
lines = cv2.HoughLinesP(edges_2nd, 1, np.pi/180, threshold=100,
minLineLength=50, maxLineGap=10)

if lines is not None:
    for i in range(lines.shape[0]):
        pt1 = (lines[i][0][0], lines[i][0][1])  # starting point coordinates x,
y
        pt2 = (lines[i][0][2], lines[i][0][3])  # ending point coordinates,
always 0 in the middle
        cv2.line(cropped_image, pt1, pt2, (255, 255, 255), 6, cv2.LINE_AA)

cv2.imshow("A", cropped_image)
cv2.waitKey(0)

# Split the image into a 8*1 grid
print(cropped_image.shape)
height, width, _ = cropped_image.shape
cell_height = height // 8
cell_width = width // 1

for row in range(8):
    for col in range(1):
        # Calculate the top-left and bottom-right coordinates of the image to be
split
        top_left = (col * cell_width, row * cell_height)
        bottom_right = ((col + 1) * cell_width, (row + 1) * cell_height)

        # Split the image
        cell_image = cropped_image[top_left[1]:bottom_right[1],
top_left[0]:bottom_right[0]]

        # Resize cropped images for the model's format.
        cell_image = cv2.resize(cell_image, (420, 36))

        # Save the split image
        cv2.imwrite(f"cropped_img/cell_{row}_{col}.jpg", cell_image)

        # Display the split image
        cv2.imshow(f"Cell {row}-{col}", cell_image)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

7. Compare EASYOCR

- The results were saved in the file 'easy_ocr_results.txt'.
- The original image files are stored in the 'easyocr_test_image' folder.

(1) EASY OCR code

- I used the same set of images for 50 random words that I used for the performance evaluation for EASY OCR.

```
import easyocr
import os

# EasyOCR setting
reader = easyocr.Reader(['ko'])  # set the korean

# image path
image_folder = 'C:/Users/ririk/Desktop/DLIP_Final_Project/easyocr_test_image'

# path
image_paths = [os.path.join(image_folder, f'image{i}.jpg') for i in range(1,
51)]

# result
for image_path in image_paths:
    result = reader.readtext(image_path)
    print(f"=== Results for {image_path} ===")
    for text in result:
        print(text)
    print()
```

(2) Custom dataset creation code

- This code saves the image matrix drawn by CAIRO as an image file.
- I used the OPENCV and numpy modules.
- The output is saved in the 'output_folder' path, so you'll need to modify it to the desired folder.

```
import cv2
import os
import numpy as np

# Load the images
image = Z['images'][:]
print(image.shape)

# Specify the folder path to save the images
output_folder = 'C:/Users/ririk/Desktop/DLIP_Final_Project/easyocr_test_image'
```

```
# Create the folder if it doesn't exist
if not os.path.exists(output_folder):
    os.makedirs(output_folder)

# Save the images
for i in range(len(image)):
    # Set the image file name
    filename = f'image{i+1}.jpg'

    # Convert the image to 2D
    img_2d = image[i].squeeze()

    # Normalize the image values to the range of 0 and 1
    img_normalized = (img_2d - np.min(img_2d)) / (np.max(img_2d) -
np.min(img_2d))

    # Check the data type of the image and convert if necessary
    if img_normalized.dtype != np.uint8:
        img_normalized = (img_normalized * 255).astype(np.uint8)

    # Check the color mode of the image and convert grayscale images to BGR
    if len(img_normalized.shape) == 2:
        img_bgr = cv2.cvtColor(img_normalized, cv2.COLOR_GRAY2BGR)
    else:
        img_bgr = img_normalized

    # Save the image
    output_path = os.path.join(output_folder, filename)
    cv2.imwrite(output_path, img_bgr)

    # Read the saved image and display it
    saved_img = cv2.imread(output_path)
    cv2.imshow('Saved Images', saved_img)
    cv2.waitKey(0)

cv2.destroyAllWindows()
```

III. Results

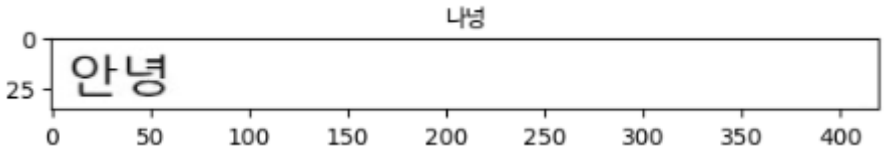
1. 50 words custom test dataset results

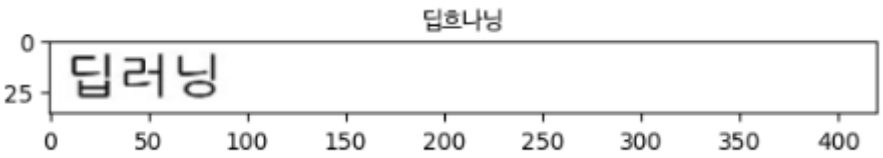
- It used a total of 50 words and did not recognize 2 words correctly. Therefore, the test accuracy was 96%.

	data number	error number	loss[%]
train	950000	665	0.07
test	50	2	4

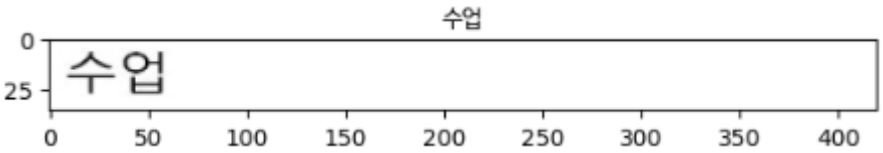
2. OPENCV text detection

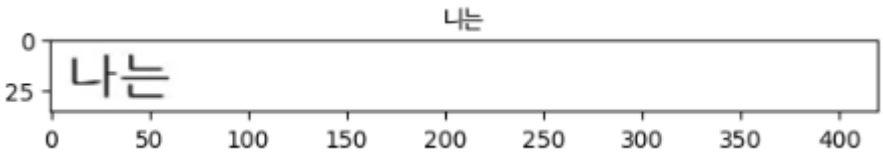
- A total of 8 words were tested. The results are shown below.





- These are not well-recognized examples.





- These are well-recognized examples.

- The results of using opencv for text detection and inferring the results were very poor.
- The recognition rate was not good, even though I ran many tests, so I ran one image (8 words) and then stopped.

	data number	error number	loss[%]
text detection + text recognition test	8	5	62.5

3. compare easy ocr with 50 words custom test dataset results

(1) EASY OCR results

```
=== Results for C:/Users/ririk/Desktop/DLIP_Final_Project/easyocr_test_image\image16.jpg ===
([[8, 2], [103, 2], [103, 33], [8, 33]], '혼자가기', 0.9997740387916565)

=== Results for C:/Users/ririk/Desktop/DLIP_Final_Project/easyocr_test_image\image17.jpg ===
([[2, 2], [46, 2], [46, 30], [2, 30]], '영구', 0.9999254851849975)

=== Results for C:/Users/ririk/Desktop/DLIP_Final_Project/easyocr_test_image\image18.jpg ===
([[4, 2], [56, 2], [56, 32], [4, 32]], '외쳐', 0.9989943557757909)

=== Results for C:/Users/ririk/Desktop/DLIP_Final_Project/easyocr_test_image\image19.jpg ===
([[2, 2], [74, 2], [74, 32], [2, 32]], '대학원', 0.9999890567510715)

=== Results for C:/Users/ririk/Desktop/DLIP_Final_Project/easyocr_test_image\image20.jpg ===
([[2, 2], [56, 2], [56, 34], [2, 34]], '광계', 0.9486626569629272)

=== Results for C:/Users/ririk/Desktop/DLIP_Final_Project/easyocr_test_image\image21.jpg ===
([[2, 2], [54, 2], [54, 30], [2, 30]], '가기', 0.9999892947172615)

=== Results for C:/Users/ririk/Desktop/DLIP_Final_Project/easyocr_test_image\image22.jpg ===
([[2, 2], [54, 2], [54, 32], [2, 32]], '하다', 0.9998675120587224)

=== Results for C:/Users/ririk/Desktop/DLIP_Final_Project/easyocr_test_image\image23.jpg ===
([[2, 2], [72, 2], [72, 28], [2, 28]], '포스템', 0.554571339060492)

=== Results for C:/Users/ririk/Desktop/DLIP_Final_Project/easyocr_test_image\image24.jpg ===
([[6, 2], [56, 2], [56, 32], [6, 32]], '용키', 0.4503454381032036)

=== Results for C:/Users/ririk/Desktop/DLIP_Final_Project/easyocr_test_image\image25.jpg ===

=== Results for C:/Users/ririk/Desktop/DLIP_Final_Project/easyocr_test_image\image26.jpg ===
([[6, 2], [54, 2], [54, 32], [6, 32]], '안녕', 0.9988237998682177)

=== Results for C:/Users/ririk/Desktop/DLIP_Final_Project/easyocr_test_image\image27.jpg ===
([[0, 2], [76, 2], [76, 34], [0, 34]], '삼거리', 0.9985492303773179)
```

- A total of 32 words out of 50 were correctly recognized, which is a very low recognition rate of 64% accuracy.

(2) Compare results table

	data number	error number	loss[%]
EASY OCR	50	28	36
trained model	50	2	4

IV. Conclusion

- In this study, I implemented and evaluated the Hangul OCR task based on the existing Hangul OCR model structure.
  OCR is divided into text detection and text recognition, and in this study, I focused on text recognition.
  For the image dataset, we used the CAIRO library.
  The model structure is SRN with CNN+GRU format.
- For my custom dataset, I got very good training results with 96% accuracy. Therefore, I concluded that the training was successful and exceeded our original expectations.
- Inferring the post-processing results after text detection through opencv was not very good.
- A comparison with EASY OCR showed that it performed much better.

V. Discussion and trouble shooting

1. Discussion

- Regarding the performance of EASY OCR and the trained model: Unlike EASY OCR, the trained model is optimized for the font format and font size of the image. Therefore, the performance of EASY OCR trained on random fonts and formats would naturally be lower.
- Low recognition rate for OPENCV processed images: This is likely due to the dataset being 'overfitted' in a different sense. The reason is that I tried to keep the font size and image shape as similar as possible, but not exactly, So that the dataset itself was a fixed environment with a limited font, limited size, and image settings, so it was not able to infer correctly when a slightly different image was input.

2. Trouble shooting

- Dataset Issue: Generating images with CAIRO from a large number of words is a simple method that does not require any dataset preparation, but it has the disadvantage of having very few variables in the environment, which can easily lead to overfitting and is very vulnerable to other variable inputs. This can be seen in the opencv processed image, where even a small change in the distance between letters, position, font size, etc. can lead to a sharp drop in prediction performance. If I had used more anomalous data for the various word images, I would expect to see much better results for the opencv images.
- Training model issue: After training a model, it was not applied to the saved model. The process of saving the model itself was fine, but after loading the model, the decoding process was stuck in an infinite loop. This could not be solved, so I had to spend a lot of time learning each time. The model was also not trained well every time, causing too much time to be spent on training.
- Font issue: I originally planned to train with multiple fonts, but this was unsuccessful because the number of Korean fonts supported by the CAIRO library is very small. Therefore, it is necessary to reconsider the dataset for further research.

VI. further works

- It is necessary to train on a much larger number of different datasets in different formats. It needs to be able to respond to different inputs.
- Currently, I use the jamoembedding method, but there are also methods for learning the letters themselves, so you can try different methods.
- Since I have researched the latter of text detection and text recognition, you can combine the text detection deep learning model to create a complete Hangeul recognition model.

VII. REFERENCE

- Base model for this study : **craftsangjae/ocr-text-recognition-korean: 한글 단어 혹은 문장 이미지를 받아 텍스트를 반환하는 Text Recognition Model (github.com)**

[1] 김정원, 김성범.(2022).장면 이미지 속 한글 문자를 종단 간 검출 및 인식 가능한 딥러닝 네트워크 모델.대한산업공학회지,48(4),398-408.

[2] 문대정, 황재문, 김지효, 황상흠.(2021).한글 문서 OCR에서의 상용 API 성능 비교 연구.대한산업공학회 추계학술대회 논문집,(),1542-1552.