FULL REPORT

INT3404E 20
Image Processing

Member List – Team 17
Hoàng Mạnh Bình - 20021305
Nguyễn Tiến Đạt - 20021327
Nguyễn Minh Phong - 21021526
Trần Trọng Quân - 21020529

I. Introduction

Overview

Sino-Nom character recognition is a specialized field within optical character recognition (OCR) focused on identifying and interpreting the complex characters used in the Sino-Nom script. Sino-Nom, which refers to traditional Chinese characters, has a rich history and significant cultural value. Accurate recognition and digitization of these characters are essential for preserving historical documents, facilitating linguistic research, and enhancing digital accessibility.

Historical Context

The Sino-Nom script, primarily used in China, consists of traditional Chinese characters that have been used for thousands of years. These characters are not only a means of communication but also an integral part of China's cultural and historical heritage. Traditional Chinese characters are known for their complexity and beauty, with each character often carrying deep meanings and historical significance.

The use of traditional Chinese characters has evolved over time, from ancient inscriptions on oracle bones and bronze vessels to classical literature and modern-day calligraphy. Despite the introduction of simplified Chinese characters in mainland China in the mid-20th century, traditional characters are still widely used in Taiwan, Hong Kong, Macau, and by overseas Chinese communities.

Importance of Sino-Nom Character Recognition

- Preservation of Cultural Heritage: Digitizing documents written in traditional Chinese characters ensures that these cultural treasures are preserved and accessible for future generations.
- Linguistic Research: Scholars can study the evolution of the Chinese language and its script by analyzing digitized texts, providing insights into historical linguistics and philology.
- Education and Accessibility: Digital versions of classical Chinese texts make it easier for students and researchers to access and study these materials, promoting a deeper understanding of Chinese literature and history.
- Technological Advancement: Developing effective character recognition systems for Sino-Nom characters pushes the boundaries of OCR

technology, contributing to advancements in artificial intelligence and machine learning.

Challenges in Sino-Nom Character Recognition

- Complexity of Characters: Traditional Chinese characters often contain many intricate strokes and variations, making accurate recognition challenging.
- Historical Document Quality: Many historical documents are old and may have deteriorated over time, with issues such as faded ink, damaged paper, and handwritten variations adding to the difficulty of recognition.
- Limited Training Data: Effective machine learning models require large datasets of annotated characters. However, the availability of such datasets for traditional Chinese characters is limited, necessitating the creation of comprehensive training data.
- Contextual Understanding: Recognizing individual characters is only part of the challenge. Effective recognition systems must also understand the context in which characters are used to accurately interpret their meaning within a text.

II. Proposal Solutions

Solution 1: Using Traditional Convolutional Neural Networks (CNN)

Overview

Convolutional Neural Networks (CNNs) are a type of deep learning model particularly effective for image recognition tasks. They are well-suited for recognizing complex patterns in images, such as the intricate strokes of traditional Chinese characters. In this solution, we employ a CNN to directly process and classify images of Sino-Nom characters.

Architecture

1. Input Layer: The input is an image of a Sino-Nom character, typically resized to a fixed dimension (e.g., 32x32 or 64x64 pixels).
2. Convolutional Layers: These layers apply a series of convolutional filters to the input image to detect various features such as edges, corners, and textures. Each convolutional layer is followed by an activation function (usually ReLU) and a pooling layer (max pooling) to reduce the spatial dimensions and computational load.

3. Fully Connected Layers: After several convolutional and pooling layers, the resulting feature maps are flattened and passed through one or more fully connected (dense) layers. These layers combine the extracted features to make the final classification.
4. Output Layer: The final layer is a softmax layer that outputs a probability distribution over all possible Sino-Nom characters.

Training Process

1. Data Preparation: Collect and preprocess a large dataset of labeled Sino-Nom character images. We noticed the data is very unbalanced speaking of quantity.
2. Augmentation: Apply data augmentation techniques such as rotation, scaling, translation, and adding noise to artificially increase the size of the training set and improve generalization.
3. Training: Use a cross-entropy loss function and an optimizer like Adam or SGD to train the CNN. The model is trained over multiple epochs, with the dataset split into training and validation sets to monitor performance and prevent overfitting.
4. Evaluation: Evaluate the model on given test data.

Advantages

High Accuracy: CNNs are highly effective at recognizing visual patterns, making them suitable for the detailed structure of Sino-Nom characters.
Feature Learning: Automatically learns relevant features from the data without the need for manual feature extraction.

Disadvantages

Computationally Intensive: Training deep CNNs requires significant computational resources and time.
Data Requirement: Requires a large amount of labeled data to achieve high performance.

Solution 2: Using Long Short-Term Memory (LSTM) Combined with CNN

Overview

Combining CNNs with Long Short-Term Memory (LSTM) networks leverages the strengths of both architectures. CNNs excel at extracting spatial features from images, while LSTMs are effective at capturing temporal dependencies and sequences. This hybrid approach is particularly useful when recognizing characters in sequences, such as lines of text or continuous scripts.

Architecture

CNN Component:

- Input Layer: Similar to the first solution, the input is an image of a Sino-Nom character or a sequence of characters.
- Convolutional Layers: Extract spatial features from the input images through multiple convolutional and pooling layers.
- Feature Map Extraction: The output of the CNN component is a set of feature maps representing the spatial characteristics of the input.

LSTM Component:

- Sequence Input: The feature maps from the CNN are reshaped and fed into the LSTM network as a sequence of vectors.
- LSTM Layers: One or more LSTM layers process the sequence of feature vectors, capturing the temporal dependencies and contextual information between characters.
- Output: The final hidden state of the LSTM is passed through fully connected layers to produce the classification output.
- Output Layer: Similar to the CNN-only solution, a softmax layer provides the probability distribution over the character classes.

Training Process

1. Data Preparation: Similar to the CNN-only approach
2. Augmentation: Apply similar data augmentation techniques to increase the robustness of the model.
3. Training: Use a combined loss function that accounts for both spatial and temporal aspects of the data. An optimizer like Adam can be used to train the network. The dataset is split into training, validation, and test sets.
4. Evaluation: Evaluate the model on given test data.

Advantages

- Contextual Understanding: The LSTM component allows the model to capture dependencies between characters, improving recognition in sequences.
- Versatility: Can handle both isolated character recognition and continuous text recognition.

Disadvantages

- Increased Complexity: The combined architecture is more complex and requires careful tuning of hyperparameters.

- Higher Computational Cost: Training and inference can be more computationally intensive due to the addition of the LSTM component.

Solution 3: Using Residual Learning

Overview

Residual learning operates on the principle of learning residual functions with reference to the layer inputs, rather than learning unreferenced functions. This is realized through the use of residual blocks, which add shortcut connections that bypass one or more layers.

Architecture

Residual block

The core building block of residual networks is the residual block, typically consisting of two convolutional layers with batch normalization and ReLU activation, alongside a shortcut connection. The skip connection adds the original input to the output of the convolutional layers, allowing the network to learn residual mappings.

Training Process

1. Data Preparation: Similar to the CNN-only approach
2. Augmentation: Apply similar data augmentation techniques to increase the robustness of the model.
3. Training: Use sparse_categorical_crossentropy as loss function.
4. Evaluation: Evaluate the model on given test data.

Advantages

- Increased Depth: Residual learning allows for the construction of very deep networks (hundreds or even thousands of layers) without encountering optimization difficulties.
- Improved Accuracy: Deeper networks enabled by residual learning often lead to improved accuracy and generalization on various tasks, including image classification, object detection, and segmentation.

Disadvantages

- Overfitting: Deeper networks are more prone to overfitting, requiring careful regularization techniques and hyperparameter tuning to prevent.

III. Experiment Design on ResNet.

1. Experiment Setup

a) Data Collection and Preprocessing

The dataset used in this experiment consists of Sino-Nom character

images. The preprocessing steps ensure that the dataset is prepared correctly for training the deep learning model.

- Ensure Sufficient PNG Files

```python
def ensure_png_files(root_folder, target_count):
    for dirpath, _, filenames in os.walk(root_folder):
        png_files = [file for file in filenames if file.lower().endswith('.png')]
        num_png_files = len(png_files)

        if num_png_files < target_count:
            if num_png_files == 0:
                print(f"Folder '{dirpath}' has no PNG files to copy from.")
                continue

            print(f"Folder '{dirpath}' has {num_png_files} PNG files. Adding {target_count - num_png_files} more PNG files.")

            for i in range(num_png_files + 1, target_count + 1):
                src_file = os.path.join(dirpath, random.choice(png_files))
                new_file_name = f"{os.path.splitext(os.path.basename(src_file))[0]}_copy_{i}.png"
                dest_file_path = os.path.join(dirpath, new_file_name)
                shutil.copy2(src_file, dest_file_path)
                print(f"Copied {src_file} to {dest_file_path}")
        else:
            print(f"Folder '{dirpath}' already has {num_png_files} PNG files. No action needed.")

root_folder = "wb_recognition_dataset/wb_recognition_dataset/train"
target_png_count = 30

ensure_png_files(root_folder, target_png_count)
```

  - We decided to increase the number of PNG files for training for increased accuracy

```python
def ensure_png_files(root_folder, target_count):
```

    - root_folder: The root directory containing subdirectories that need to be checked for PNG files
    - target_count: The minimum number of PNG files each subdirectory should have

```python
for dirpath, _, filenames in os.walk(root_folder):
    png_files = [file for file in filenames if file.lower().endswith('.png')]
    num_png_files = len(png_files)
```

    - The function uses os.walk to traverse the directory tree starting from root_folder
    - dirpath is the current directory path
    - _ (unused variable) would be the list of subdirectories in the current directory
    - filenames is the list of filenames in the current directory
    - This list comprehension filters out the files that have a .png extension (case-insensitive)
    - The number of PNG files in the current directory is stored in num_png_files

```
for i in range(num_png_files + 1, target_count + 1):
    src_file = os.path.join(dirpath, random.choice(png_files))
    new_file_name = f"{os.path.splitext(os.path.basename(src_file))[0]}_copy_{i}.png"
    dest_file_path = os.path.join(dirpath, new_file_name)
    shutil.copy2(src_file, dest_file_path)
    print(f"Copied {src_file} to {dest_file_path}")
```

- A loop runs to create the required number of PNG file copies
- random.choice(png_files) selects a random PNG file from the existing ones in the directory to copy
- new_file_name generates a new name for the copied file by appending _copy_i to the original filename
- dest_file_path constructs the full path for the new copied file
- shutil.copy2(src_file, dest_file_path) copies the selected file to the new location with the new name
- A message is printed each time a file is copied

- And remove the unprocessed file

```
#deleting unprocessed files
def delete_files_starting_with(directory):
    regex_pattern = r'^nlvnpf.*'
    deleted_files_count = 0
    for root, dirs, files in os.walk(directory):
        for file in files:
            file_path = os.path.join(root, file)
            if re.match(regex_pattern, file):
                os.remove(file_path)
                deleted_files_count = deleted_files_count + 1
    print(deleted_files_count," files deleted")

# Example usage:
directory_path = "/kaggle/input/wb-data/wb_recognition_dataset/train"

delete_files_starting_with(directory_path)
```

- Data Augmentation and Normalization

```
# Path where the zip file was extracted
train_directory = '/kaggle/input/wb-recognition-dataset-plus/wb_recognition_dataset/wb_recognition_dataset/train'
# Load images from the extracted directory
train_dataset = tf.keras.preprocessing.image_dataset_from_directory(
    train_directory,
    labels = "inferred",
    class_names=None,
    label_mode='int',
    color_mode=COLOR_MODE,
    validation_split=VALIDATION_SPLIT,
    subset= 'training',
    seed=111,
    batch_size= BATCH_SIZE,
    image_size= IMAGE_SIZE
)

validate_dataset = tf.keras.preprocessing.image_dataset_from_directory(
    train_directory,
    labels = "inferred",
    class_names=None,
    label_mode='int',
    color_mode=COLOR_MODE,
    validation_split=VALIDATION_SPLIT,
    subset= 'validation',
    seed=222,
    batch_size=BATCH_SIZE,
    image_size=IMAGE_SIZE
)

# train_dataset = train_dataset.concatenate(test_dataset)
number_of_classes = len(train_dataset.class_names)
```

- The data is then loaded using TensorFlow's image_dataset_from_directory method, which automatically handles splitting the dataset into training and validation sets

b) Model Implementation

The experiment evaluated different versions of the ResNet model architecture. The ResNet50 architecture was primarily used, but ResNet50V2, ResNet101, ResNet101V2, ResNet152, and ResNet152V2 were also explored.

- Model Setup and Compilation
  - The model is set up and compiled with the option to use transfer learning. If transfer learning is enabled, a pre-trained ResNet50 model with ImageNet weights is used, and layers up to a specified point are frozen

```
if TRANFER_LEARNING == True:
    resnet_layer = tf.keras.applications.ResNet50(
        include_top=False,
        weights= 'imagenet',
        input_shape = (IMAGE_SIZE[0], IMAGE_SIZE[1], 3),
        pooling='max',
    )
    # Transfer learning
    count = 0
    for layer in resnet_layer.layers[:FREEZING_RANGE]:
        layer.trainable = False
        count = count+1
    print(count)
else:
    resnet_layer = tf.keras.applications.ResNet50(
        weights = None,
        include_top=False,
        input_shape = (IMAGE_SIZE[0], IMAGE_SIZE[1], CHANNELS),
        pooling='max',
    )
    # Transfer learning
    count = 0
    for layer in resnet_layer.layers:
        count = count+1
    print(count)
model = Sequential()
if RANDOM_ROTATION == True:
    model.add(RandomRotation(factor=FACTORS))
model.add(Rescaling(1./255))
model.add(resnet_layer)
model.add(Dense(number_of_classes, activation='softmax'))
model.build((None, IMAGE_SIZE[0], IMAGE_SIZE[1], CHANNELS))
model.summary()
```

- If transfer learning is enabled, it initializes a ResNet50 model pre-trained on the ImageNet dataset

```
resnet_layer = tf.keras.applications.ResNet50(
    include_top=False,
    weights= 'imagenet',
    input_shape = (IMAGE_SIZE[0], IMAGE_SIZE[1], 3),
    pooling='max',
)
```

  - include_top=False: Excludes the fully connected layers at the top of the network, making it suitable for feature extraction

- weights='imagenet': Uses weights pre-trained on the ImageNet dataset
- input_shape=(IMAGE_SIZE[0], IMAGE_SIZE[1], 3): Specifies the input shape of the images (height, width, 3 color channels)
- pooling='max': Adds a global max pooling layer after the convolutional layers
- The script then freezes a range of layers so their weights are not updated during training

```
count = 0
for layer in resnet_layer.layers[:FREEZING_RANGE]:
    layer.trainable = False
    count = count+1
print(count)
```

- layer.trainable = False: Freezes the layer
- FREEZING_RANGE: Number of layers to freeze
- count: Counts the number of layers frozen

- If transfer learning is not enabled, it initializes a ResNet50 model without pre-trained weights

```
else:
    resnet_layer = tf.keras.applications.ResNet50(
        weights = None,
        include_top=False,
        input_shape = (IMAGE_SIZE[0], IMAGE_SIZE[1], CHANNELS),
        pooling='max',
    )
```

- weights=None: No pre-trained weights are used
- CHANNELS: Number of channels in the input images

- Counts the total number of layers in the ResNet50 model

```
count = 0
for layer in resnet_layer.layers:
    count = count+1
print(count)
```

- Constructs the overall model

```
model = Sequential()
if RANDOM_ROTATION == True:
    model.add(RandomRotation(factor=FACTORS))
model.add(Rescaling(1./255))
model.add(resnet_layer)
model.add(Dense(number_of_classes, activation='softmax'))
model.build((None, IMAGE_SIZE[0], IMAGE_SIZE[1], CHANNELS))
model.summary()
```

- Sequential(): Initializes a sequential model
- RandomRotation(factor=FACTORS): Adds random rotations to the input images during training to augment the dataset
- Rescaling(1./255): Rescales pixel values from [0, 255] to [0, 1]
- Adds the ResNet50 layer (either pre-trained or not):
- Dense(number_of_classes, activation='softmax'): Adds a dense layer with a number of units equal to the number of classes and a softmax activation function for classification
- model.build((None, IMAGE_SIZE[0], IMAGE_SIZE[1], CHANNELS)): Builds the model with the specified input shape, where None is the batch size
- model.summary(): Prints a summary of the model architecture

c) Model training

- The model was trained with the following setup
  - Batch Size: 64
  - Epochs: 50
  - Callbacks: Early stopping, model checkpointing, and learning rate reduction on plateau

```
checkpoint_path = "/kaggle/input/model-ver15/resnet_checkpoint.model (1).keras"
# Create a callback that saves the model's weights
cp_callback = ModelCheckpoint(
    filepath=checkpoint_path,
    monitor='val_loss',
    save_best_only=True,
    mode='min',
    verbose=1
)
reduce_lr = tf.keras.callbacks.ReduceLROnPlateau(
    monitor='val_loss',
    factor=0.2,
    patience=5,
    verbose = 1
)

model.compile(
    optimizer=Adam(),
    loss= 'sparse_categorical_crossentropy',
    metrics=['accuracy']
    )
```

```
checkpoint_path = "/kaggle/input/model-ver15/resnet_checkpoint.model (1).keras"
```

- checkpoint_path: Specifies the path where the model checkpoints (weights) will be saved

```
cp_callback = ModelCheckpoint(
    filepath=checkpoint_path,
    monitor='val_loss',
    save_best_only=True,
    mode='min',
    verbose=1
)
```

- ModelCheckpoint: A Keras callback that saves the model after every epoch
- filepath: Path where the model weights will be saved
- monitor='val_loss': Monitors the validation loss during training
- save_best_only=True: Saves the model only if the monitored metric (val_loss) has improved
- mode='min': Indicates that the monitored metric should be minimized
- verbose=1: Prints messages when the model is being saved

```
reduce_lr = tf.keras.callbacks.ReduceLROnPlateau(
    monitor='val_loss',
    factor=0.2,
    patience=5,
    verbose = 1
)
```

- ReduceLROnPlateau: A Keras callback that reduces the learning rate when a metric has stopped improving
- monitor='val_loss': Monitors the validation loss during training
- factor=0.2: Reduces the learning rate by a factor of 0.2.
- patience=5: Number of epochs with no improvement after which the learning rate will be reduced
- verbose=1: Prints messages when the learning rate is reduced

```
model.compile(
    optimizer=Adam(),
    loss= 'sparse_categorical_crossentropy',
    metrics=['accuracy']
    )
```

- model.compile: Configures the model for training
- optimizer=Adam(): Uses the Adam optimizer, which is a popular optimization algorithm combining the advantages of both AdaGrad and RMSProp
- loss='sparse_categorical_crossentropy': Uses sparse categorical cross entropy as the loss function. This is appropriate for classification tasks where the labels are provided as integers
- metrics=['accuracy']: Evaluates the model's performance based on accuracy

d) Evaluation

- The trained model was evaluated on the validation set to assess its performance

```
model.evaluate(validate_dataset, verbose=1,batch_size=BATCH_SIZE)
# Saving Model
model.save('sinonom_resnet.keras')
```

2. Testing

- The model's accuracy was recorded during training and evaluation. The final accuracy was noted after evaluating the model on the validation set.

```
# Load the CSV file
csv_path = '/kaggle/input/wb-data/wb_recognition_dataset/val/labels.csv'
data = pd.read_csv(csv_path)

# Load the images from the folder
image_folder = '/kaggle/input/wb-data/wb_recognition_dataset/val/images'
image_paths = [os.path.join(image_folder, str(image_name)+".jpg") for image_name in data['image_name']]
images = [tf.keras.preprocessing.image.load_img(image_path,COLOR_MODE,target_size=(IMAGE_SIZE[0], IMAGE_SIZE[1])) for image_path in image_paths]
images = [tf.keras.preprocessing.image.img_to_array(image) for image in images]
images = np.array(images)

# Preprocess the images

# Predict the labels
predictions = model.predict(images)
predicted_labels = [train_dataset.class_names[np.argmax(prediction)] for prediction in predictions]

# Add the predicted labels to the dataframe
data['predicted_label'] = predicted_labels
correct = 0
for i in range(len(data)):
    if str(data['label'][i]) == str(data['label_pred'][i]):
        correct += 1
TEST_ACCURACY = round(correct/len(data), 5)



print("Correct predictions:", correct)
print("Total predictions:", len(data))
print("Accuracy:", TEST_ACCURACY)
```

- Loading the CSV File with Labels

```
# Load the CSV file
csv_path = '/kaggle/input/wb-data/wb_recognition_dataset/val/labels.csv'
data = pd.read_csv(csv_path)
```

  - csv_path: Path to the CSV file containing the validation labels
  - data = pd.read_csv(csv_path): Loads the CSV file into a pandas DataFrame named data

- Loading the Images

```
# Load the images from the folder
image_folder = '/kaggle/input/wb-data/wb_recognition_dataset/val/images'
image_paths = [os.path.join(image_folder, str(image_name)+".jpg") for image_name in data['image_name']]
images = [tf.keras.preprocessing.image.load_img(image_path,COLOR_MODE,target_size=(IMAGE_SIZE[0], IMAGE_SIZE[1])) for image_path in image_paths]
images = [tf.keras.preprocessing.image.img_to_array(image) for image in images]
images = np.array(images)
```

  - image_folder: Path to the folder containing the validation images
  - image_paths: List comprehension to generate the full paths to the images based on their names in the CSV file
  - tf.keras.preprocessing.image.load_img: Loads each image from its path, resizing it to the specified target_size and using the specified COLOR_MODE (e.g., 'rgb' or 'grayscale')

- tf.keras.preprocessing.image.img_to_array: Converts the loaded images to NumPy arrays
- images = np.array(images): Converts the list of image arrays to a single NumPy array

- Making predictions

```
# Predict the labels
predictions = model.predict(images)
predicted_labels = [train_dataset.class_names[np.argmax(prediction)] for prediction in predictions]
```

- model.predict(images): Uses the model to predict the labels for the loaded images
- Converts the raw prediction outputs into class labels by taking the index of the highest predicted value (np.argmax) and mapping it to the corresponding class name from the training dataset's class names

- Adding Predicted Labels to the DataFrame

```
# Add the predicted labels to the dataframe
data['predicted_label'] = predicted_labels
correct = 0
for i in range(len(data)):
    if str(data['label'][i]) == str(data['label_pred'][i]):
        correct += 1
TEST_ACCURACY = round(correct/len(data), 5)
```

- Adds the predicted labels as a new column to the DataFrame
- Initializes a counter for correct predictions
- Loops through each row in the DataFrame, comparing the true label (data['label'][i]) with the predicted label (data['predicted_label'][i])
- Increments the counter if the prediction is correct
- Calculates the test accuracy as the number of correct predictions divided by the total number of predictions, rounded to five decimal places

- Print out the results

```
print("Correct predictions:", correct)
print("Total predictions:", len(data))
print("Accuracy:", TEST_ACCURACY)
```

- Exporting result to ***google sheet*** for better clearance

```
gc = gspread.service_account(filename='/kaggle/input/helicopter/helical-decoder-382901-2b30ac9f8085.json')

sh = gc.open("resnet_report")
worksheet = sh.get_worksheet(0)
row_index = GSHEET_ROW_INDEX

current_row_values = worksheet.row_values(row_index)
is_empty_row = all(value == '' for value in current_row_values)
if is_empty_row:
    print("adding data to row..")
    TRAIN_ACCURACY = history.history['accuracy'][-1]
    VALIDATION_ACCURACY = history.history['val_accuracy'][-1]
    TEST_ACCURACY = correct/len(data)
    OUTPUTTIME=get_time_string()

    row_data = [IMAGE_SIZE, VALIDATION_SPLIT, TRANFER_LEARNING, FREEZING_RANGE, COLOR_MODE, POOLING, RANDOM_ROTATION, FACTORS, TRAIN_ACCURACY, VALIDATION_ACCURACY, TEST_ACCURACY,OUTPUTTIME]
#     for i, value in enumerate(row_data):
#         worksheet.update_cell(row_index, i+1, str(value))
else:
    raise ValueError("Row is not empty")
```

- Integrate google sheet

```
gc = gspread.service_account(filename='/kaggle/input/helicopter/helical-decoder-382901-2b30ac9f8085.json')
```

  - Initializes the Google Sheets client using a service account file stored in the Kaggle input directory
- Accessing the spreadsheet

```
sh = gc.open("resnet_report")
worksheet = sh.get_worksheet(0)
row_index = GSHEET_ROW_INDEX
```

  - Opens the Google Sheets document named "resnet_report" and fetches the first worksheet
- Checking if the row is empty or not

```
current_row_values = worksheet.row_values(row_index)
is_empty_row = all(value == '' for value in current_row_values)
if is_empty_row:
    print("adding data to row..")
    TRAIN_ACCURACY = history.history['accuracy'][-1]
    VALIDATION_ACCURACY = history.history['val_accuracy'][-1]
    TEST_ACCURACY = correct/len(data)
    OUTPUTTIME=get_time_string()

    row_data = [IMAGE_SIZE, VALIDATION_SPLIT, TRANFER_LEARNING, FREEZING_RANGE, COLOR_MODE, POOLING, RANDOM_ROTATION, FACTORS, TRAIN_ACCURACY, VALIDATION_ACCURACY, TEST_ACCURACY,OUTPUTTIME]
```

  - Fetches the values of the specified row from the worksheet
  - Checks if all values in the row are empty to determine if the row is empty
  - If the row is empty
    - Calculates certain metrics such as train accuracy, validation accuracy, test accuracy, and output time
    - Constructs a list row_data containing these metrics
    - The row is updated with these calculated values.
  - If the row is not empty

```
else:
    raise ValueError("Row is not empty")
```

    - Throws an error

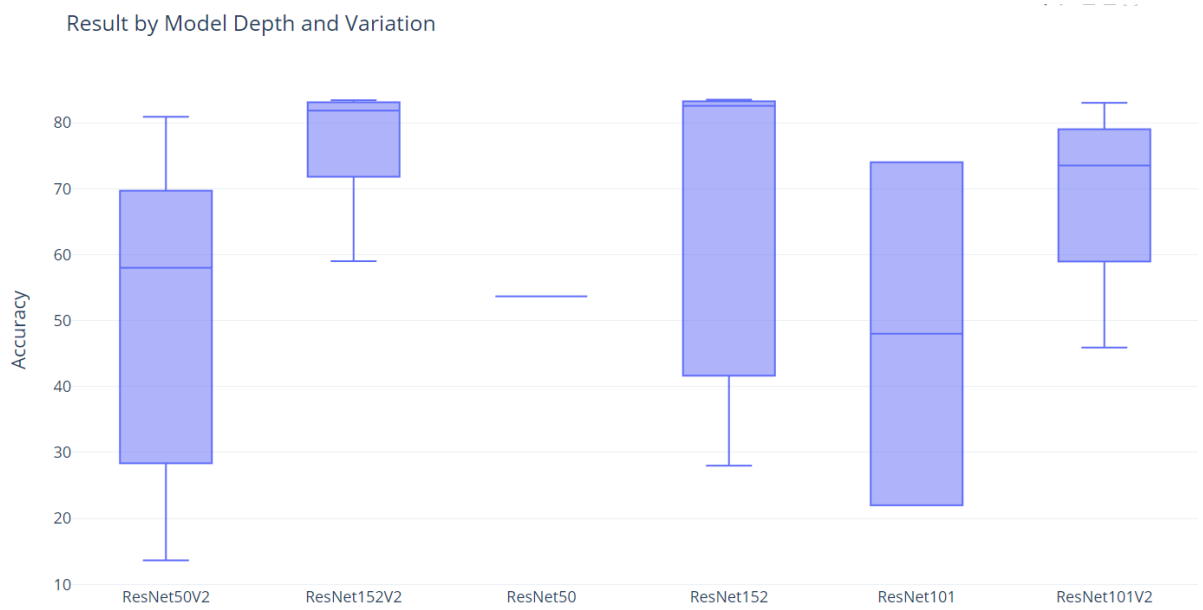
IV. Experiment Result

1. Analysis and Discussion

Our Top-1 Accuracy model reached 83.5%.

Overfitting Analysis:

- Deeper ResNet architecture, specifically ResNet152, exhibited no signs of overfitting.
- Despite the increased depth, the model maintained generalization performance, suggesting robustness.



Data Augmentation Evaluation:

- RGB image decoding did not yield notable improvements in recognition accuracy.
- Addition of minor data augmentation techniques, such as rotation (5-10 degrees), resulted in a slight enhancement in recognition performance.
- Rotation augmentation contributed to better model generalization without significantly increasing computational complexity.

Transfer Learning Experiment Outcome:

- Transfer learning using pre-trained weights from ImageNet did not show promising results.
- Freezing a portion of layers and leveraging pre-learned features did not lead to significant improvements in Sino-Nomcharacter recognition accuracy.

- Indicates the necessity for domain-specific feature learning in character recognition tasks, as features learned from generic datasets like ImageNet may not fully capture the nuances of Sino-Nomcharacters.

2. Conclusion

In conclusion, the experiment successfully demonstrated the efficacy of ResNet models for Sino-Nom character recognition. The use of deeper and improved ResNet architectures, along with robust data augmentation and preprocessing techniques, resulted in high accuracy and reliable performance, highlighting the potential for advanced deep learning models in OCR applications for complex scripts.

IV. Contribution

Nguyen Tien Dat is the person who proposes the idea using Resnet for this problem, who also implements the model.
Hoang Manh Binh supported Dat to code the Model using Resnet, particularly: Parameter Configuration, Model compiling, Model testing and joining to training models in some different versions of Resnet: Resnet50, Resnet50V2
Nguyen Minh Phong joined to train models in more different versions: Resnet101, Resnet101V2, Resnet152, Resnet152V2.
Tran Trong Quan took the mission of Data preprocessing, he used Python and Algorithms to scale, convert, and combine the old dataset to the new data as expected.
Disregarding the difficulty of the tasks and the amount of work, we unanimously agree that the contribution level of all members is 25%.