



Faculty of Computers &  
Artificial Intelligence



Benha University

# Arabic Sign Language Translator

A neural network application that takes an image as input and outputs what Arabic alphabet character this image refers to.

**Computer Science** Department

By

Amir Nazmy

Abdallah Khairy

---

Abdallah Habsa

Mohammed Hussein

---

Mohammed Mostafa

Youssef Alaa

**Supervised By**

Dr: Ibrahim Zaghloul | Eng: Mahmoud Bassiouny

---

intentionally left blank.

---

---

# Table of Contents

Dataset ..... 3

    About This Dataset .....3

    Sample.....3

**Code** ..... 4

GUI ..... 23

---

# Dataset

## About This Dataset

A fully labelled dataset of Arabic Sign Language (ArSL) images is developed for research related to sign language recognition. The dataset will provide researcher the opportunity to investigate and develop automated systems for the deaf and hard of hearing people using machine learning, computer vision and deep learning algorithms. The contribution is a large fully labelled dataset for Arabic Sign Language (ArSL) which is made publically available and free for all researchers. The dataset which is named ArSL2018 consists of 54,049 images for the 32 Arabic sign language sign and alphabets collected from 40 participants in different age groups. Different dimensions and different variations were present in images which can be cleared using pre-processing techniques to remove noise, center the image, etc. The dataset is made available publicly at:

[Arabic Alphabets Sign Language Dataset \(ArASL\) - Mendeley Data](#)

---

## Sample



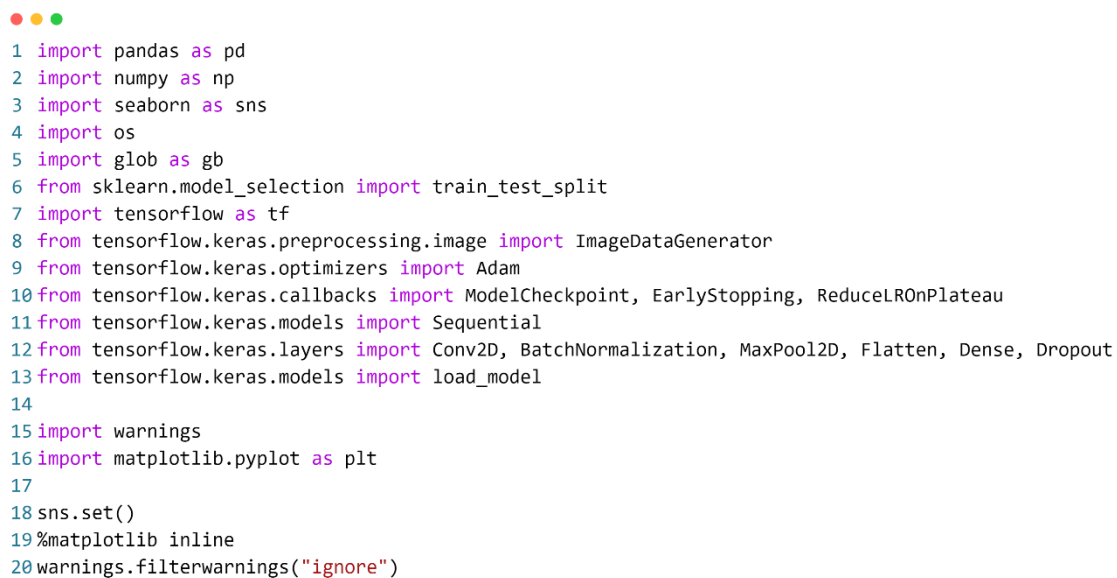
---

# Code

In this section we're going to break down the code, and demonstrate how it works, and how to implement such an app, and how to train such a model.

---

## First n first



```
1 import pandas as pd
2 import numpy as np
3 import seaborn as sns
4 import os
5 import glob as gb
6 from sklearn.model_selection import train_test_split
7 import tensorflow as tf
8 from tensorflow.keras.preprocessing.image import ImageDataGenerator
9 from tensorflow.keras.optimizers import Adam
10 from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping, ReduceLROnPlateau
11 from tensorflow.keras.models import Sequential
12 from tensorflow.keras.layers import Conv2D, BatchNormalization, MaxPool2D, Flatten, Dense, Dropout
13 from tensorflow.keras.models import load_model
14
15 import warnings
16 import matplotlib.pyplot as plt
17
18 sns.set()
19 %matplotlib inline
20 warnings.filterwarnings("ignore")
```

Obviously, this code is a set of import statements.

Here's a breakdown of what each import statement:

`import pandas as pd`

This imports the pandas library, which is essential for data manipulation and analysis. It's commonly used for working with structured data like CSV files. The **pd** is an alias, making it easier to reference pandas functions.

`import numpy as np`

Numpy is a library for numerical computing in Python. It provides support for arrays, matrices, and high-level mathematical functions. The **np** alias is standard practice.

`import seaborn as sns`

Seaborn is a statistical plotting library built on top of matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics. **sns** is the conventional alias used.

`import os`

This module provides a way of using operating system-dependent functionality like reading or writing to the filesystem.

---

---

```
import glob as gb
```

The glob module finds all the pathnames matching a specified pattern according to the rules used by the Unix shell. **gb** is an alias for easier access.

```
from sklearn.model_selection import train_test_split
```

**This** function from the scikit-learn library is used to split a dataset into training and testing sets, which is crucial for evaluating the performance of a machine learning model.

```
import tensorflow as tf
```

TensorFlow is an open-source machine learning framework. It's used for both research and production at Google. **tf** is the commonly used alias.

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator
```

This is used to generate batches of tensor image data with real-time data augmentation. It can help improve the model by creating variations of the images.

```
from tensorflow.keras.optimizers import Adam
```

Adam is an optimization algorithm that can be used instead of the classical stochastic gradient descent procedure to update network weights iteratively based on training data.

```
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping, ReduceLROnPlateau
```

These are callbacks in Keras that are used during the training of the neural network to save the model, stop training early if it's not improving, or reduce the learning rate when a metric has stopped improving.

```
from tensorflow.keras.models import Sequential
```

This is used to create a linear stack of layers for the neural network.

```
from tensorflow.keras.layers import Conv2D, BatchNormalization, MaxPool2D, Flatten, Dense, Dropout
```

These are different types of layers that can be added to the model. They are used for convolutional operations, normalizing inputs, downsampling, flattening input, creating fully connected layers, and preventing overfitting, respectively.

```
from tensorflow.keras.models import load_model
```

This function is used to load a saved Keras model.

```
import warnings
```

This module is used to manage warnings in Python.

```
import matplotlib.pyplot as plt
```

Matplotlib is a plotting library, and **pyplot** is its plotting framework. **plt** is the standard alias used.

```
sns.set()
```

This sets the aesthetic parameters in one step for all seaborn plots.

```
%matplotlib inline
```

This is a magic function that renders the figure in a notebook (instead of displaying a dump of the figure object).

```
warnings.filterwarnings("ignore")
```

This line is used to ignore warnings generated by Python, often used to clean up the output of the script.

This script sets up an environment for machine learning with neural networks, particularly for image processing tasks. It includes data handling, model building, and plotting capabilities. The use of aliases like **pd**, **np**, **sns**, and **tf** is a common convention that makes the code more readable and easier to write.

For the dataset loading:

```
1 train_path = 'zaras1-database-54k'
```

This line of code assigns the path where the training data is stored to `train_path` variable.

## The next line of code

```
1 arabic_alphabet = ['ا', 'ب', 'ت', 'ث', 'ج', 'ح', 'خ', 'د', 'ذ', 'ر', 'ز', 'س', 'ش', 'ص', 'ط', 'ظ', 'ع', 'ف', 'ق', 'ك', 'گ', 'ل', 'م', 'ن', 'هـ', 'و', 'ي']
```

creates a list of Arabic alphabet characters and assigns it to the variable `arabic_alphabet`.

For verifying the data:

```
1 data = pd.read_csv('zaras1-database-54k/Labels/ImagesClassPath.csv')
2 data.head(-5)
```

This code performs two main operations using the pandas library in Python:

- 1) `data = pd.read_csv('zarasl-database-54k/Labels/ImagesClassPath.csv')`: This line reads a CSV (Comma-Separated Values) file into a pandas DataFrame. The `pd.read_csv()` function is a powerful tool that allows you to read data from a CSV file and convert it into a DataFrame, which is a 2-dimensional labeled data structure with columns of potentially different types. The file path `'zarasl-database-54k/Labels/ImagesClassPath.csv'` indicates the location of the CSV file relative to the current working directory.

- 
- 2) `data.head(-5)`: The `.head()` method is used to return the first n rows of a DataFrame. However, when you use a negative number as an argument, like `-5`, it returns all rows except the last 5 rows of the DataFrame. This can be useful when you want to view the majority of your dataset quickly without the last few entries.



## Next


```
1 for i in range(data.shape[0]):  
2     data.loc[i, 'ClassId'] = arabic_alphabet[data.loc[i, 'ClassId']]  
3 data.head(-5)
```

The code is a Python loop that iterates over the rows of a pandas DataFrame and updates a column based on a mapping from another list. Here's a step-by-step explanation:

- 1) `for i in range(data.shape[0]):`: This line starts a for-loop that will iterate over each row in the DataFrame `data`. The `data.shape[0]` expression returns the number of rows in the DataFrame. The `range()` function generates a sequence of numbers, which in this case, will be from `0` to `data.shape[0] - 1`.
- 2) `data.loc[i, 'ClassId'] = arabic_alphabet[data.loc[i, 'ClassId']]`: Inside the loop, for each iteration (row), the `.loc` method is used to access the 'ClassId' column of the DataFrame `data` at the current row `i`. The value in 'ClassId' is assumed to be an integer that acts as an index to the `arabic_alphabet` list. The `arabic_alphabet` list contains Arabic characters, and this line replaces the integer in 'ClassId' with the corresponding Arabic character from the `arabic_alphabet` list.
- 3) `data.head(-5)`: After the loop has updated all rows, this line displays the DataFrame `data`, excluding the last 5 rows. This is useful for quickly inspecting the changes made to the majority of the DataFrame without loading the entire dataset into view.

```
1 df = pd.DataFrame({'img':data['ImagePath'],  
2                   'class':data['ClassId']})  
3 df.head(-5)
```

- 1) we create a **Pandas DataFrame** called `df`.
- 2) The **DataFrame** has two columns:
  - a. The first column is named `'img'` and contains data from the `'ImagePath'` column of the data object.
  - b. The second column is named `'class'` and contains data from the `'ClassId'` column of the data object.
- 3) The `.head(-5)` method is used to display all rows in the DataFrame except for the last 5 rows.
- 4) The `-5` argument in `.head(-5)` means that the last 5 rows will be excluded from the output. If you want to see the entire DataFrame, you can simply use `.head()` without any arguments.




```

1 df_train, df_test = train_test_split(df, test_size=0.1, shuffle=True)
2
3 print('Train shape:', df_train.shape)
4 print('Test shape:', df_test.shape)

```

- 1) We use the `train_test_split` function from the `sklearn.model_selection` module (This function is commonly used to split a dataset into training and testing subsets for machine learning tasks).
- 2) The `train_test_split` function takes the following parameters:
  - `df`: The DataFrame you want to split.
  - `test_size`: The proportion of the dataset to include in the test split. In your case, it's set to 0.1 (10%).
  - `shuffle`: Whether or not to shuffle the data before splitting. You've set it to True, which means the data will be randomly shuffled.
- 3) The resulting `df_train` and `df_test` DataFrames contain the training and testing subsets, respectively.
- 4) Finally, you're printing the shapes of these subsets using the `.shape` attribute.



```

1 size = 64
2 channels = 1
3 batch = 128
4 epochs = 25
5 steps_per_epoch= data.shape[0] // batch

```

1. **Size**: The size parameter is set to 64. This likely refers to the size of the input data or the dimensions of the input images. In machine learning, this value often corresponds to the height and width of the images in pixels.
2. **Channels**: The channels parameter is set to 1. This typically indicates that the input data is grayscale (single-channel) rather than color (which would have 3 channels for red, green, and blue).
3. **Batch Size**: The batch parameter is set to 128. In training neural networks, batch size determines how many examples are processed together before updating the model's weights. A larger batch size can lead to faster training but requires more memory.
4. **Epochs**: The epochs parameter is set to 25. An epoch represents one complete pass through the entire training dataset. Training a model for multiple epochs allows it to learn from the data over several iterations.
5. **Steps per Epoch**: The `steps_per_epoch` value is calculated based on the total data size and batch size. It determines how many batches of data are processed in each epoch. For example, if you have 1000 samples and a batch size of 128, you'd have approximately 7-8 steps per epoch.

```
1 datagen = ImageDataGenerator(rescale=1./255,  
2                               zoom_range=0.2,  
3                               width_shift_range=.2, height_shift_range=.2,  
4                               rotation_range=30,  
5                               brightness_range=[0.8, 1.2],  
6                               horizontal_flip=False)  
7  
8 datagenScale = ImageDataGenerator(rescale=1./255)
```

Let's break down the provided information about the two `ImageDataGenerator` instances:

1. datagen:

- This `ImageDataGenerator` is used for data augmentation during training. Data augmentation helps improve model generalization by creating variations of the input data.
- Here are the specific augmentations applied:
  - **Rescale:** The pixel values are rescaled to a range of [0, 1] by dividing each pixel value by 255. This ensures consistent scaling across all images.
  - **Zoom Range:** Randomly zooms in or out on the image by a factor specified in the `zoom_range`. For example, if `zoom_range=0.2`, the image can be zoomed by up to 20%.
  - **Width and Height Shift Range:** Randomly shifts the image horizontally and vertically by a fraction of its width and height, respectively. The specified values (e.g., `width_shift_range=.2`) control the maximum shift.
  - **Rotation Range:** Randomly rotates the image by an angle within the specified range (e.g., `rotation_range=30`). This introduces robustness to rotation variations.
  - **Brightness Range:** Randomly adjusts the brightness of the image. The specified range (e.g., [0.8, 1.2]) determines the brightness factor.
  - **Horizontal Flip:** The `horizontal_flip` parameter is set to `False`, meaning that horizontal flipping is not applied. If set to `True`, it would randomly flip images horizontally.

2. datagenScale:

- This `ImageDataGenerator` is simpler and mainly used for scaling the pixel values.
- The only transformation applied here is:
  - **Rescale:** Similar to the first generator, it rescales pixel values to the range [0, 1].

```

1 X_train = datagen.flow_from_dataframe(
2     df_train,
3     directory = train_path,
4     x_col = 'img',
5     y_col = 'class',
6     target_size = (size,size),
7     color_mode = 'grayscale',
8     shuffle = True,
9     batch_size = batch)
10
11 X_test = datagenScale.flow_from_dataframe(
12     df_test,
13     directory = train_path,
14     x_col = 'img',
15     y_col = 'class',
16     target_size= (size,size),
17     color_mode = 'grayscale',
18     shuffle = True,
19     batch_size = batch)

```

Let's break down the provided code snippets related to creating data generators for training and testing:

### 1. X\_train:

- This line of code creates a data generator for training data (X\_train).
- Here are the key parameters:
  - df\_train: The training data is assumed to be stored in a pandas DataFrame called df\_train.
  - directory: The directory where the training images are located (specified by train\_path).
  - x\_col: The column in the DataFrame that contains the image filenames (specified as 'img').
  - y\_col: The column in the DataFrame that contains the corresponding class labels (specified as 'class').
  - target\_size: The desired size for input images (specified as (size, size)).
  - color\_mode: The color mode for the images (set to 'grayscale' for single-channel images).
  - shuffle: Whether to shuffle the data during training (set to True).
  - batch\_size: The batch size for training (specified as batch).
- The resulting X\_train data generator will yield batches of training data during model training.

### 2. X\_test:

- This line of code creates a data generator for testing/validation data (X\_test).
- Similar to X\_train, it uses the same parameters but with the testing/validation data (df\_test).
- The key difference is that X\_test does not apply data augmentation (only rescaling), as it's used for evaluation rather than training.

```
1 checkpoint_filepath = 'ArSL-BestModel.keras'
2 optimizer = Adam(learning_rate=0.001)
3
4 callback_checkpoint = ModelCheckpoint(filepath=checkpoint_filepath, save_weights_only=False, monitor='val_accuracy', mode='max', save_best_only=True)
5 callback_earlystopping = EarlyStopping(monitor='val_accuracy', mode='max', min_delta=0.003, patience=25)
6 callback_learningrate = ReduceLROnPlateau(monitor='accuracy', mode='max', min_delta=0.01, patience=5, factor=.05, verbose=1)
7
8 callbacks = [callback_checkpoint, callback_earlystopping, callback_learningrate]
```

Let's break down the provided code related to model training callbacks:

1. checkpoint\_filepath:

- This variable specifies the file path where the best model weights will be saved during training.
- The ModelCheckpoint callback will monitor the validation accuracy ('val\_accuracy') and save the model weights when the validation accuracy improves (mode: 'max').

2. optimizer:

- The optimizer used for training the neural network. In this case, it's an instance of the Adam optimizer with a learning rate of 0.001.

3. Callbacks:

- Callbacks are functions that can be applied during training to perform specific actions at certain points.
- Here are the three callbacks defined:
  - callback\_checkpoint:
    - ModelCheckpoint saves the best model weights based on the validation accuracy.
    - It saves the entire model (including architecture and weights) to the specified file path (checkpoint\_filepath).
  - callback\_earlystopping:
    - EarlyStopping monitors the validation accuracy and stops training if the accuracy does not improve by at least 0.003 (min\_delta) for 25 consecutive epochs (patience).
    - This prevents overfitting and saves time by stopping early if the model performance plateaus.
  - callback\_learningrate:
    - ReduceLROnPlateau adjusts the learning rate during training.
    - It monitors the training accuracy and reduces the learning rate by a factor of 0.05 if the accuracy does not improve by at least 0.01 (min\_delta) for 5 consecutive epochs (patience).
    - This helps fine-tune the model as it approaches convergence.

These callbacks are essential for efficient training and preventing overfitting.

```
1 X_train.class_indices
```

The `class_indices` attribute provides a mapping of class names to their corresponding integer labels. In your case, it represents the mapping for the classes in your training data. Here's an example of what it might look like:

```
{'class1': 0, 'class2': 1, 'class3': 2, ...}
```

- Each class name (e.g., 'class1', 'class2', etc.) corresponds to a specific category or label.
- The associated integer value (e.g., 0, 1, 2, etc.) is used internally by the model during training and prediction.

```
1 Model = Sequential([Conv2D(filters=32, kernel_size=(3,3), activation="relu", input_shape=(size,size,channels)),
2   BatchNormalization(),
3   MaxPool2D(2,2, padding='same'),
4   Dropout(0.25),
5   Conv2D(filters=128, kernel_size=(3,3), activation="relu"),
6   BatchNormalization(),
7   MaxPool2D(2,2, padding='same'),
8   Dropout(0.25),
9   Conv2D(filters=512, kernel_size=(3,3), activation="relu"),
10  BatchNormalization(),
11  MaxPool2D(2,2, padding='same'),
12  Dropout(0.25),
13
14  Conv2D(filters=2048, kernel_size=(3,3), activation="relu"),
15  BatchNormalization(),
16  MaxPool2D(2,2, padding='same'),
17  Dropout(0.25),
18  Flatten(),
19  BatchNormalization(),
20  Dense(units=4096, activation="relu"),
21  BatchNormalization(),
22  Dropout(0.25),
23  Dense(units=1024, activation="relu"),
24  BatchNormalization(),
25  Dropout(0.25),
26  Dense(units=256, activation="relu"),
27  BatchNormalization(),
28  Dropout(0.25),
29  Dense(units=64, activation="relu"),
30  BatchNormalization(),
31  Dropout(0.5),
32  Dense(units=32, activation="softmax"),
33 ])
34
35
36 Model.compile(optimizer=optimizer, loss="categorical_crossentropy", metrics=["accuracy"])
37
38
39 Model.summary()
```

---

Let's break down the architecture of the neural network model you've provided:

**1. Input Layer:**

- The input shape is determined by (`size, size, channels`), where `size` represents the height and width of the input image, and `channels` represents the number of color channels (e.g., 3 for RGB images).
- The input layer consists of a 2D convolutional layer (`Conv2D`) with 32 filters (also known as kernels) and a kernel size of 3x3.
- The activation function used is `ReLU` (Rectified Linear Unit).
- Batch normalization is applied after the convolutional layer.
- A 2x2 max-pooling layer (`MaxPool2D`) with padding is used to downsample the feature maps.
- Dropout with a rate of 0.25 is applied to prevent overfitting.

**2. Intermediate Convolutional Layers:**

- Three additional convolutional layers follow the input layer.
- The second convolutional layer has 128 filters, the third has 512 filters, and the fourth has 2048 filters.
- Each layer is followed by batch normalization, max-pooling, and dropout.

**3. Flatten Layer:**

- After the convolutional layers, the feature maps are flattened into a 1D vector.
- This prepares the data for the fully connected layers.

**4. Fully Connected Layers (Dense Layers):**

- The flattened vector is passed through a series of dense (fully connected) layers.
- The first dense layer has 4096 units with `ReLU` activation.
- Batch normalization and dropout are applied after this layer.
- The second dense layer has 1024 units with `ReLU` activation.
- Again, batch normalization and dropout are applied.
- The third dense layer has 256 units with `ReLU` activation.
- Batch normalization and dropout follow.
- The fourth dense layer has 64 units with `ReLU` activation.
- Batch normalization and dropout are applied.

**5. Output Layer:**

- The final dense layer has 32 units with `softmax` activation.
- `Softmax` converts the output values into probabilities, suitable for multi-class classification tasks.
- The model is compiled with categorical cross-entropy loss and accuracy as the evaluation metric.

**6. Optimizer:**

- The optimizer used for training is not specified in the code snippet. You would need to define an optimizer (e.g., `Adam`, `SGD`) separately.

Overall, this architecture is a deep convolutional neural network (CNN) with multiple convolutional and dense layers. It's designed for image classification tasks, where the input images are expected to have the specified shape.

---

```
1 history = Model.fit(X_train, validation_data=X_test, epochs=epochs, callbacks=callbacks)
```

The line of code you've provided is related to training the neural network model using the training data (X\_train) and validating it using the validation data (X\_test). Let's break it down:

### 1. Training the Model:

- The Model.fit() method is used to train the neural network.
- The X\_train dataset contains input images along with their corresponding labels (target values).
- During training, the model learns to adjust its weights and biases to minimize the specified loss function (in this case, categorical cross-entropy).
- The training process involves forward and backward passes (forward propagation and backpropagation) to update the model's parameters.

### 2. Validation Data:

- The validation\_data parameter specifies the validation dataset (X\_test).
- Validation data is used to evaluate the model's performance during training without affecting the model's weights.
- It helps prevent overfitting by monitoring how well the model generalizes to unseen data.

### 3. Epochs:

- The epochs parameter determines the number of times the entire training dataset is passed through the model.
- Each epoch consists of multiple iterations (batches) of training.
- Increasing the number of epochs can improve model performance, but too many epochs may lead to overfitting.

### 4. Callbacks:

- The callbacks parameter allows you to specify custom functions or predefined callbacks during training.
- Callbacks can be used for tasks like saving model checkpoints, early stopping, or adjusting learning rates dynamically.



```

1 acc = history.history['accuracy']
2 val_acc = history.history['val_accuracy']
3 loss = history.history['loss']
4 val_loss = history.history['val_loss']
5 epochs_range = range(len(acc))
6
7 plt.figure(figsize=(15, 15))
8 plt.subplot(2, 2, 1)
9 plt.plot(epochs_range, acc, label='Training Accuracy')
10 plt.plot(epochs_range, val_acc, label='Validation Accuracy')
11 plt.legend(loc='lower right')
12 plt.title('Training and Validation Accuracy')
13
14 plt.subplot(2, 2, 2)
15 plt.plot(epochs_range, loss, label='Training Loss')
16 plt.plot(epochs_range, val_loss, label='Validation Loss')
17 plt.legend(loc='upper right')
18 plt.title('Training and Validation Loss')
19 plt.show()

```

Let's break down the code snippet you've provided, which visualizes the training and validation accuracy as well as the training and validation loss during the training process:

### 1. Training and Validation Accuracy Plot:

- The first subplot (`plt.subplot(2, 2, 1)`) displays the training accuracy and validation accuracy over epochs.
- The x-axis represents the number of epochs, and the y-axis represents the accuracy values.
- The blue line represents the training accuracy, while the orange line represents the validation accuracy.
- As the number of epochs increases, you can observe how well the model performs on both the training and validation data. Ideally, both curves should increase and converge.

### 2. Training and Validation Loss Plot:

- The second subplot (`plt.subplot(2, 2, 2)`) shows the training loss and validation loss over epochs.
- Similar to the accuracy plot, the x-axis represents epochs, and the y-axis represents the loss values.
- The blue line represents the training loss, and the orange line represents the validation loss.
- The goal is to minimize the loss function during training. If the training loss decreases while the validation loss increases, it may indicate overfitting.

### 3. Interpreting the Plots:

- **Accuracy Plots:**
  - If the training accuracy increases while the validation accuracy remains low, it suggests overfitting (the model is memorizing the training data).
  - If both curves increase and converge, it indicates good generalization.

---

- **Loss Plots:**

- A decreasing training loss is desirable, as it means the model is learning from the data.
- The validation loss should also decrease initially but then stabilize or increase slightly (indicating good generalization).
- If the validation loss diverges from the training loss, it may indicate overfitting.

Remember that these plots provide insights into the model's performance during training. Adjusting hyperparameters, using regularization techniques, and monitoring these metrics can help improve the model's accuracy and prevent overfitting.



When you use `Model.save('finalmodel.h5')` in Keras, it saves your trained neural network model to a file named `'finalmodel.h5'`. Here's what gets saved:

1. **Model Architecture:**

- The structure of your neural network (layers, connections, etc.).
- This allows you to recreate the same model later.

2. **Model Weights:**

- The learned weights for each connection between neurons.
- These weights represent the knowledge the model gained during training.

3. **Training Configuration:**

- Information about how the model was trained (loss function, optimizer, etc.).

4. **Optimizer State (if any):**

- The state of the optimizer (e.g., momentum, learning rate) at the end of training.

You can later load this saved model using `loaded_model = keras.models.load_model('finalmodel.h5')` to make predictions or continue training from where you left off.

```

1 # Step 1: Load the Keras model
2 from sklearn.metrics import multilabel_confusion_matrix, accuracy_score, precision_score, recall_score, f1_score
3 Model = tf.keras.models.load_model('finalmodel.h5')
4 # Step 2: Make predictions on test data
5 # Assuming you have X_test and y_test generators already defined
6 y_pred = Model.predict(X_test)
7
8 # Convert predictions to binary predictions if needed
9 y_pred_binary = (y_pred > 0.5).astype(int)
10
11 # Step 3: Calculate metrics
12 # Since you're using generators, you need to loop through batches to get all predictions
13 # Initialize lists to store predictions and true labels
14 all_y_pred_binary = []
15 all_y_true = []
16
17 # Loop through batches of test data generator
18 for i in range(len(X_test)):
19     batch_x, batch_y = next(X_test)
20     batch_pred = Model.predict(batch_x)
21     batch_pred_binary = (batch_pred > 0.5).astype(int)
22
23     all_y_pred_binary.extend(batch_pred_binary)
24     all_y_true.extend(batch_y)
25
26 # Calculate multilabel confusion matrix
27 mcm = multilabel_confusion_matrix(all_y_true, all_y_pred_binary)
28
29 # Extract true positive, true negative, false positive, false negative values
30 tp = mcm[:, 1, 1]
31 tn = mcm[:, 0, 0]
32 fp = mcm[:, 0, 1]
33 fn = mcm[:, 1, 0]
34
35 # Calculate other metrics
36 accuracy = accuracy_score(all_y_true, all_y_pred_binary)
37 precision = precision_score(all_y_true, all_y_pred_binary, average='micro')
38 recall = recall_score(all_y_true, all_y_pred_binary, average='micro')
39 f1score = f1_score(all_y_true, all_y_pred_binary, average='micro')
40
41 print("Confusion Matrix:")
42 for i in range(len(tp)):
43     print(f"Class {i+1}:")
44     print("TN:", tn[i], "FP:", fp[i])
45     print("FN:", fn[i], "TP:", tp[i])
46
47 print("Accuracy:", accuracy)
48 print("Precision:", precision)
49 print("Recall:", recall)
50 print("F1 Score:", f1score)

```

Let's break down the code step by step:

### 1. Loading the Keras Model:

- The first step is to load a pre-trained Keras model from a file named 'finalmodel.h5'. This model is presumably trained for some specific task (e.g., image classification, text sentiment analysis, etc.).
- The tf.keras.models.load\_model('finalmodel.h5') line loads the model from the specified file.

### 2. Making Predictions on Test Data:

- Next, the code assumes that you already have test data (X\_test) and corresponding ground truth labels (y\_test).
- The Model.predict(X\_test) line generates predictions for the test data using the loaded model. The resulting y\_pred contains predicted probabilities for each class.

### 3. Converting Probabilities to Binary Predictions:

- To make binary predictions (class labels), the code threshold the predicted probabilities. If a probability is greater than 0.5, the corresponding class is considered positive; otherwise, it's negative.
- The line y\_pred\_binary = (y\_pred > 0.5).astype(int) converts the probabilities to binary predictions.

#### 4. Calculating Metrics:

- Since you're using data generators (`X_test` and `y_test`), the code loops through batches of test data to get all predictions.
- It initializes lists (`all_y_pred_binary` and `all_y_true`) to store predictions and true labels.
- For each batch, it predicts using the model (`batch_pred`) and converts the probabilities to binary (`batch_pred_binary`).
- The confusion matrix is calculated using `multilabel_confusion_matrix` for each class.
- Metrics such as accuracy, precision, recall, and F1-score are computed using `accuracy_score`, `precision_score`, `recall_score`, and `f1_score`, respectively.

#### 5. Printing Results:

- The code prints the confusion matrix for each class, including true negatives (TN), false positives (FP), false negatives (FN), and true positives (TP).
- It also displays overall accuracy, precision, recall, and F1-score.

```
1 import cv2
2 import numpy as np
3 from keras.models import load_model
4
5 # Load the trained model
6 model = load_model('finalmodel.h5')
7
8 # Function to preprocess a single image
9 def preprocess_image(image_path, size):
10     # Read the image
11     image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
12     # Resize the image
13     image = cv2.resize(image, (size, size))
14     # Reshape the image to match model input shape
15     image = image.reshape(-1, size, size, 1)
16     # Normalize the image
17     image = image / 255.0
18     return image
19
20 # Path to your single test image
21 test_image_path = 'output-onlinejpgtools.jpg'
22
23 # Preprocess the single test image
24 preprocessed_image = preprocess_image(test_image_path, size)
25
26 # Perform prediction
27 prediction = model.predict(preprocessed_image)
28
29 # Interpret the prediction
30 predicted_class_index = np.argmax(prediction)
31 print("Predicted class index:", predicted_class_index)
32 predicted_class = arabic_alphabet[predicted_class_index]
33
34 print("Predicted class:", predicted_class)
```

Let's break down the code step by step:

##### 1. Loading the Trained Model:

- The first line of code imports the necessary libraries (`cv2`, `numpy`, and `keras.models`) and loads a pre-trained Keras model from a file named `'finalmodel.h5'`.
- The loaded model is stored in the variable `model`.

## 2. Preprocessing a Single Image:

- The `preprocess_image` function is defined to preprocess a single image before making predictions.
- It takes two arguments:
  - `image_path`: The path to the image file.
  - `size`: The desired size for the image (assumed to be square).
- Inside the function:
  - The image is read using `cv2.imread` with grayscale mode (`cv2.IMREAD_GRAYSCALE`).
  - The image is resized to the specified size using `cv2.resize`.
  - The reshaped image is converted to a 4D array to match the model's input shape.
  - Finally, the image is normalized by dividing pixel values by 255.0.

## 3. Predicting the Class:

- The variable `test_image_path` contains the path to a single test image (e.g., `'output-onlinejpgtools.jpg'`).
- The image is preprocessed using the `preprocess_image` function.
- The `model.predict(preprocessed_image)` line generates predictions for the preprocessed image. The resulting `prediction` contains probabilities for each class.

## 4. Interpreting the Prediction:

- The index of the predicted class is obtained using `np.argmax(prediction)`.
- The corresponding class label is retrieved from an array (presumably named `arabic_alphabet`) using the predicted index.
- The predicted class label is printed.

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from tensorflow.keras.models import load_model
4 class_names = ['ا', 'ب', 'ت', 'ث', 'ج', 'ح', 'خ', 'د', 'ذ', 'ر', 'ز', 'س', 'ش', 'ص', 'ط', 'ظ', 'ع', 'غ', 'ف', 'ق', 'ك', 'ل', 'م', 'ن', 'ه', 'و', 'ي']
5
6 # Load the saved model
7 loaded_model = load_model('finalmodel.h5')
8
9 # Choose random images from the test set
10 num_images_to_show = 5
11 random_indices = np.random.choice(len(df_test), num_images_to_show)
12
13 # Set up the grid layout
14 fig, axes = plt.subplots(1, num_images_to_show, figsize=(15, 5))
15
16 # Preprocess the images and make predictions
17 for i, idx in enumerate(random_indices):
18     image_path = df_test.iloc[idx]['img']
19     image = plt.imread(os.path.join(train_path, image_path))
20     image = image.reshape((1, size, size, channels)) / 255.0 # Reshape and normalize the image
21
22     # Make prediction
23     prediction = loaded_model.predict(image)
24     predicted_class = class_names[np.argmax(prediction)]
25
26     # Display the image and its predicted class
27     axes[i].imshow(image.reshape((size, size)), cmap='gray')
28     axes[i].set_title(f'Predicted Class: {predicted_class}')
29     axes[i].axis('off')
30
31 plt.tight_layout()
32 plt.show()
```

---

Let's break down the code snippet you provided:

### 1. Loading the Saved Model:

- The first step is to load a pre-trained Keras model from a file named `'finalmodel.h5'`.
- The line `loaded_model = load_model('finalmodel.h5')` loads the model.

### 2. Choosing Random Images from the Test Set:

- The variable `num_images_to_show` specifies how many random images you want to display.
- The line `random_indices = np.random.choice(len(df_test), num_images_to_show)` selects random indices from the test dataset.

### 3. Setting Up the Grid Layout:

- The `fig, axes = plt.subplots(1, num_images_to_show, figsize=(15, 5))` line creates a grid of subplots to display the images and their predicted classes.
- It sets up a single row with `num_images_to_show` columns.

### 4. Preprocessing Images and Making Predictions:

- For each randomly selected index:
  - The image path is retrieved from the test dataset (`df_test`) using `df_test.iloc[idx]['img']`.
  - The image is read using `plt.imread` and reshaped to match the model's input shape.
  - The image is normalized by dividing pixel values by 255.0.
  - The model predicts the class probabilities for the preprocessed image using `loaded_model.predict(image)`.
  - The predicted class index is obtained using `np.argmax(prediction)`.
  - The corresponding class label is retrieved from the `class_names` list.
  - The image and its predicted class are displayed in the subplot.

### 5. Displaying the Results:

- The `imshow` function displays the image.
- The `set_title` function sets the title of the subplot with the predicted class.
- The `axis('off')` line removes the axis ticks and labels.

### 6. Tightening the Layout and Showing the Plot:

- The `plt.tight_layout()` ensures that the subplots are properly spaced.
- Finally, `plt.show()` displays the entire plot.

```

1 import numpy as np
2 import tensorflow as tf
3
4 # Load the TensorFlow Lite model
5 interpreter = tf.lite.Interpreter(model_path='model.tflite')
6 interpreter.allocate_tensors()
7
8 # Get input and output tensors
9 input_details = interpreter.get_input_details()
10 output_details = interpreter.get_output_details()
11
12 # Prepare input data (replace this with your actual input data)
13 input_data = np.random.rand(1, 64, 64, 1).astype(np.float32) # Example input data (modify as needed)
14
15 # Set input tensor
16 interpreter.set_tensor(input_details[0]['index'], input_data)
17
18 # Run inference
19 interpreter.invoke()
20
21 # Get output tensor
22 output_data = interpreter.get_tensor(output_details[0]['index'])
23
24 # Process output (replace this with your post-processing code)
25 predicted_class_index = np.argmax(output_data)
26 predicted_class = class_names[predicted_class_index] # Example output processing (modify as needed)
27
28 print("Predicted class:", predicted_class)
29

```

Let's break down the code snippet you provided:

#### 1. Loading the TensorFlow Lite Model:

- The first step is to load a TensorFlow Lite model from a file named `'model.tflite'`.
- The line `interpreter = tf.lite.Interpreter(model_path='model.tflite')` initializes the interpreter.

#### 2. Allocating Tensors and Getting Input/Output Details:

- The `interpreter.allocate_tensors()` line allocates memory for the model's input and output tensors.
- The `input_details` and `output_details` are dictionaries containing information about the model's input and output tensors, respectively.

#### 3. Preparing Input Data:

- The variable `input_data` is an example input data (a random 64x64 grayscale image).
- You should replace this with your actual input data before running inference.

#### 4. Setting Input Tensor and Running Inference:

- The `interpreter.set_tensor(input_details[0]['index'], input_data)` line sets the input tensor with the provided data.
- The `interpreter.invoke()` line runs inference on the model.

#### 5. Getting Output Tensor and Processing Results:

- The `output_data` contains the model's predictions (output tensor).
- The `np.argmax(output_data)` finds the index of the predicted class.
- The `predicted_class` is obtained from the `class_names` list based on the predicted index.
- Finally, the predicted class label is printed.

# GUI

