```
import json
import numpy as np
import tensorflow as tf
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Embedding, Dropout, Masking, TimeDistributed, Bidirectional  # Import Bidirectional
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from tensorflow.keras.callbacks import EarlyStopping # Import EarlyStopping



# 1. Load the JSON Data
with open("/content/finger_sequences.json", "r") as file:
    data = json.load(file)

# 2. Prepare Data & Labels
labels = []
sequences = []

for label, instances in data.items():
    for sequence in instances:
        finger_ids = [int(pair[0]) for pair in sequence]  # Convert finger ID to int
        durations = [pair[1] for pair in sequence]  # Extract duration times

        # Append sequence and corresponding label
        sequences.append(list(zip(finger_ids, durations)))
        labels.append(label)


# 3. Convert Labels to Numeric
label_encoder = LabelEncoder()
encoded_labels = label_encoder.fit_transform(labels)  # Convert labels to integers
```

Padding

```
Start coding or generate with AI.


durations = [dur for seq in sequences for _, dur in seq]  # Extract all durations
mean_duration = np.mean(durations)


def custom_pad_sequences(sequences, maxlen, padding='post', dtype='float32'):
  """Pads sequences with custom values based on label.

  Args:
    sequences: The list of sequences to pad.
    maxlen: The desired length for all sequences.
    padding: 'pre' or 'post' (default 'post').
    dtype: The data type of the padded sequences.

  Returns:
    A NumPy array of padded sequences.
  """
  padded_sequences = []
  for i, seq in enumerate(sequences):  # Use enumerate to get index i
    # Determine padding element based on corresponding label
    label = labels[i]  # Get label for the sequence
    if label == 'CallHarisVai':
      padding_element = ("2", mean_duration)  # Using finger_map dictionary
    elif label == 'Noise':
      padding_element = ("1", mean_duration)  # Using finger_map dictionary
    elif label == 'LikeMe':
      padding_element = ("3", mean_duration)  # Using finger_map dictionary
    elif label == 'ComeCLose':
      padding_element = ("8", mean_duration)  # Using finger_map dictionary
    else:
      padding_element = (int(-1), mean_duration)  # Default padding element

    # Pad the sequence to the desired length
    if len(seq) < maxlen:
      padding_elements = [padding_element] * (maxlen - len(seq))
      if padding == 'pre':
```

```
        padded_seq = padding_elements + seq
      else:  # padding == 'post'
        padded_seq = seq + padding_elements
    else:
      padded_seq = seq[:maxlen]  # Truncate if longer than maxlen

    padded_sequences.append(padded_seq)
  return np.array(padded_sequences, dtype=dtype)


# 4. Pad Sequences for Uniform Length (Modified)
max_seq_length = max(len(seq) for seq in sequences)
padded_sequences = custom_pad_sequences(sequences, maxlen=max_seq_length, padding='post', dtype='float32')
```

Start coding or generate with AI.

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 1s/step - accuracy: 0.8750 - loss: 0.5874
Test Accuracy: 0.8750
```

Start coding or generate with AI.

```
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is consi
```

output 1 sequence

```
# 5. Train-Test Split
X_train, X_test, y_train, y_test = train_test_split(padded_sequences, encoded_labels, test_size=0.2, random_state=42)

model = Sequential([
    Masking(mask_value=0.0, input_shape=(max_seq_length, 2)),  # Corrected input shape to (max_seq_length, 2)
    Bidirectional(LSTM(128, return_sequences=True)),
    Bidirectional(LSTM(64, return_sequences=True)),
    Bidirectional(LSTM(32)), #Added this LSTM layer without return_sequences=True.
    #This will return the output from the last timestep in the Bidirectional LSTM.
    Dropout(0.3),
    Dense(64, activation='relu'),
    Dense(len(set(labels)), activation='softmax')
])

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

# 8. Train with Early Stopping
early_stop = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)
model.fit(X_train, y_train, epochs=100, batch_size=16,
          validation_data=(X_test, y_test),
          callbacks=[early_stop])
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/masking.py:47: UserWarning: Do not pass an `input_shape`/`input_dim` argum
  super().__init__(**kwargs)
Epoch 1/100
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-39-0067e337334c> in <cell line: 0>()
     17 # 8. Train with Early Stopping
     18 early_stop = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)
---> 19 model.fit(X_train, y_train, epochs=100, batch_size=16,
     20           validation_data=(X_test, y_test),
     21           callbacks=[early_stop])

                                 1 frames
/usr/local/lib/python3.11/dist-packages/keras/src/backend/tensorflow/nn.py in sparse_categorical_crossentropy(target, output,
from_logits, axis)
    723         )
    724     if len(target.shape) != len(output.shape[:-1]):
--> 725         raise ValueError(
    726             "Argument `output` must have rank (ndim) `target.ndim - 1`. "
    727             "Received: "

ValueError: Argument `output` must have rank (ndim) `target.ndim - 1`. Received: target.shape=(16,), output.shape=(16, 33, 4)
```

label sequence output

```
# 5. Train-Test Split
X_train, X_test, y_train, y_test = train_test_split(padded_sequences, encoded_labels, test_size=0.2, random_state=42)

# Duplicate labels for each timestep in the sequence
y_train_repeated = np.repeat(y_train[:, np.newaxis], max_seq_length, axis=1)
y_test_repeated = np.repeat(y_test[:, np.newaxis], max_seq_length, axis=1)


model = Sequential([
    Masking(mask_value=0.0, input_shape=(max_seq_length, 2)),  # Corrected input shape to (max_seq_length, 2)
    Bidirectional(LSTM(128, return_sequences=True)),
    Bidirectional(LSTM(64, return_sequences=True)),
    Bidirectional(LSTM(32, return_sequences=True)),
    Dropout(0.3),
    Dense(64, activation='relu'),
    TimeDistributed(Dense(len(set(labels)), activation='softmax')) # Wrap the final Dense layer with TimeDistributed
])

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

# 8. Train with Early Stopping
early_stop = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)
model.fit(X_train, y_train_repeated, epochs=100, batch_size=16, # Use the repeated labels for training
          validation_data=(X_test, y_test_repeated), # Use the repeated labels for validation
          callbacks=[early_stop])
```

```
Epoch 1/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 17s 751ms/step - accuracy: 0.4316 - loss: 1.3737 - val_accuracy: 0.4413 - val_loss: 1.2762
Epoch 2/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 158ms/step - accuracy: 0.5225 - loss: 1.2074 - val_accuracy: 0.4962 - val_loss: 1.1187
Epoch 3/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 166ms/step - accuracy: 0.5184 - loss: 1.1082 - val_accuracy: 0.5114 - val_loss: 1.0299
Epoch 4/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 243ms/step - accuracy: 0.5513 - loss: 1.0023 - val_accuracy: 0.5227 - val_loss: 0.9749
Epoch 5/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 252ms/step - accuracy: 0.5631 - loss: 0.9225 - val_accuracy: 0.5246 - val_loss: 0.9257
Epoch 6/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 2s 377ms/step - accuracy: 0.5686 - loss: 0.8706 - val_accuracy: 0.5284 - val_loss: 0.8915
Epoch 7/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 316ms/step - accuracy: 0.6265 - loss: 0.7944 - val_accuracy: 0.5795 - val_loss: 0.8401
Epoch 8/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 157ms/step - accuracy: 0.6593 - loss: 0.7715 - val_accuracy: 0.6477 - val_loss: 0.8243
Epoch 9/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 156ms/step - accuracy: 0.6804 - loss: 0.7105 - val_accuracy: 0.6269 - val_loss: 0.7640
Epoch 10/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 178ms/step - accuracy: 0.8172 - loss: 0.5460 - val_accuracy: 0.7386 - val_loss: 0.6495
Epoch 11/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 198ms/step - accuracy: 0.8466 - loss: 0.4865 - val_accuracy: 0.7708 - val_loss: 0.6174
Epoch 12/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 162ms/step - accuracy: 0.9050 - loss: 0.4142 - val_accuracy: 0.8182 - val_loss: 0.5232
Epoch 13/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 147ms/step - accuracy: 0.9570 - loss: 0.2589 - val_accuracy: 0.8163 - val_loss: 0.4902
Epoch 14/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 170ms/step - accuracy: 0.9743 - loss: 0.1606 - val_accuracy: 0.8617 - val_loss: 0.4440
Epoch 15/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 161ms/step - accuracy: 0.9758 - loss: 0.1194 - val_accuracy: 0.8750 - val_loss: 0.5629
Epoch 16/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 2s 282ms/step - accuracy: 0.9239 - loss: 0.2594 - val_accuracy: 0.8750 - val_loss: 0.4108
Epoch 17/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 316ms/step - accuracy: 0.9699 - loss: 0.1047 - val_accuracy: 0.8769 - val_loss: 0.5039
Epoch 18/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 2s 490ms/step - accuracy: 0.9490 - loss: 0.1631 - val_accuracy: 0.6989 - val_loss: 0.8668
Epoch 19/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 2s 176ms/step - accuracy: 0.9276 - loss: 0.2201 - val_accuracy: 0.7727 - val_loss: 0.7210
Epoch 20/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 144ms/step - accuracy: 0.9006 - loss: 0.2194 - val_accuracy: 0.8390 - val_loss: 0.4688
Epoch 21/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 237ms/step - accuracy: 0.9707 - loss: 0.1197 - val_accuracy: 0.9072 - val_loss: 0.3200
Epoch 22/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 149ms/step - accuracy: 0.9683 - loss: 0.1162 - val_accuracy: 0.8750 - val_loss: 0.4204
Epoch 23/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 151ms/step - accuracy: 0.9779 - loss: 0.0828 - val_accuracy: 0.8769 - val_loss: 0.3444
Epoch 24/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 155ms/step - accuracy: 0.9876 - loss: 0.0674 - val_accuracy: 0.8769 - val_loss: 0.4435
Epoch 25/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 144ms/step - accuracy: 0.9434 - loss: 0.2121 - val_accuracy: 0.7841 - val_loss: 0.4069
```

```
Epoch 26/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 150ms/step - accuracy: 0.9931 - loss: 0.0562 - val_accuracy: 0.8144 - val_loss: 0.5681
<keras.src.callbacks.history.History at 0x7c9f08584750>
```

```
# 5. Train-Test Split
X_train, X_test, y_train, y_test = train_test_split(padded_sequences, encoded_labels, test_size=0.2, random_state=42)

# Duplicate labels for each timestep in the sequence
y_train_repeated = np.repeat(y_train[:, np.newaxis], max_seq_length, axis=1)
y_test_repeated = np.repeat(y_test[:, np.newaxis], max_seq_length, axis=1)


model = Sequential([
    Masking(mask_value=0.0, input_shape=(max_seq_length, 2)),  # Corrected input shape to (max_seq_length, 2)
    Bidirectional(LSTM(128, return_sequences=True)),
    Bidirectional(LSTM(64, return_sequences=True)),
    Bidirectional(LSTM(32, return_sequences=True)),
    Dropout(0.3),
    Dense(64, activation='relu'),
    TimeDistributed(Dense(2, activation='linear'))  # Output layer for finger ID and duration
])

model.compile(loss='mse', optimizer='adam', metrics=['mae'])

# 8. Train with Early Stopping
early_stop = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)


model.fit(X_train, y_train, epochs=100, batch_size=16,
          validation_data=(X_test, y_test),
          callbacks=[early_stop])
```

```
Epoch 1/100
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/masking.py:47: UserWarning: Do not pass an `input_shape`/`input_dim` argum
  super().__init__(**kwargs)
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-48-c881d23dd8d9> in <cell line: 0>()
     23
     24
---> 25 model.fit(X_train, y_train, epochs=100, batch_size=16,
     26           validation_data=(X_test, y_test),
     27           callbacks=[early_stop])

 ──────────────────────────────────── 1 frames ────────────────────────────────────
/usr/local/lib/python3.11/dist-packages/keras/src/losses/losses.py in mean_squared_error(y_true, y_pred)
   1677     y_true = ops.convert_to_tensor(y_true, dtype=y_pred.dtype)
   1678     y_true, y_pred = squeeze_or_expand_to_same_rank(y_true, y_pred)
-> 1679     return ops.mean(ops.square(y_true - y_pred), axis=-1)
   1680
   1681

ValueError: Dimensions must be equal, but are 16 and 2 for '{{node compile_loss/mse/sub}} = Sub[T=DT_FLOAT](compile_loss/mse/Cast,
sequential_11_1/time_distributed_15_1/transpose_2)' with input shapes: [16], [16,33,2].
```

```
# 5. Train-Test Split
X_train, X_test, y_train, y_test = train_test_split(padded_sequences, encoded_labels, test_size=0.2, random_state=42)

# Reshape y_train and y_test to (num_samples, 1)
y_train = y_train.reshape(-1, 1)
y_test = y_test.reshape(-1, 1)

# Duplicate labels for each timestep in the sequence
y_train_repeated = np.repeat(y_train, max_seq_length, axis=1)
y_test_repeated = np.repeat(y_test, max_seq_length, axis=1)

# Reshape to (num_samples, sequence_length, num_features)
y_train_repeated = y_train_repeated[:, :, np.newaxis]  # Add a new axis for features
y_test_repeated = y_test_repeated[:, :, np.newaxis]  # Add a new axis for features

# Add duration information to y_train_repeated and y_test_repeated
# Assuming your padded_sequences have shape (num_samples, sequence_length, 2)
# where the last dimension contains (finger_id, duration)
# Use X_train and X_test instead of padded_sequences to match the shapes
y_train_repeated = np.concatenate([y_train_repeated, X_train[:, :, 1:]], axis=2)  # Concatenate duration
```

```
y_test_repeated = np.concatenate([y_test_repeated, X_test[:, :, 1:]], axis=2)  # Concatenate duration


model = Sequential([
    Masking(mask_value=0.0, input_shape=(max_seq_length, 2)),  # Corrected input shape to (max_seq_length, 2)
    Bidirectional(LSTM(128, return_sequences=True)),
    Bidirectional(LSTM(64, return_sequences=True)),
    Bidirectional(LSTM(32, return_sequences=True)),
    Dropout(0.3),
    Dense(64, activation='relu'),
    TimeDistributed(Dense(2, activation='linear'))  # Output layer for finger ID and duration
])

model.compile(loss='mse', optimizer='adam', metrics=['mae'])

# 8. Train with Early Stopping
early_stop = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)

model.fit(X_train, y_train_repeated, epochs=100, batch_size=16,
          validation_data=(X_test, y_test_repeated),
          callbacks=[early_stop])
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/masking.py:47: UserWarning: Do not pass an `input_shape`/`input_dim` ar
  super().__init__(**kwargs)
Epoch 1/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 17s 896ms/step - loss: 1.5458 - mae: 0.8494 - val_loss: 1.1528 - val_mae: 0.7930
Epoch 2/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 151ms/step - loss: 0.9575 - mae: 0.7128 - val_loss: 1.0872 - val_mae: 0.8273
Epoch 3/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 143ms/step - loss: 0.8817 - mae: 0.7105 - val_loss: 1.0579 - val_mae: 0.7519
Epoch 4/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 157ms/step - loss: 0.8282 - mae: 0.6525 - val_loss: 1.0406 - val_mae: 0.7591
Epoch 5/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 187ms/step - loss: 0.8245 - mae: 0.6557 - val_loss: 1.0032 - val_mae: 0.7388
Epoch 6/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 161ms/step - loss: 0.8956 - mae: 0.6863 - val_loss: 0.9897 - val_mae: 0.7492
Epoch 7/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 144ms/step - loss: 0.8404 - mae: 0.6783 - val_loss: 1.0281 - val_mae: 0.7500
Epoch 8/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 159ms/step - loss: 0.7595 - mae: 0.6222 - val_loss: 0.9702 - val_mae: 0.7245
Epoch 9/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 159ms/step - loss: 0.7573 - mae: 0.6158 - val_loss: 0.9367 - val_mae: 0.7093
Epoch 10/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 154ms/step - loss: 0.6954 - mae: 0.5930 - val_loss: 0.8970 - val_mae: 0.7078
Epoch 11/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 159ms/step - loss: 0.6098 - mae: 0.5678 - val_loss: 0.7415 - val_mae: 0.6022
Epoch 12/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 154ms/step - loss: 0.5365 - mae: 0.5262 - val_loss: 0.5848 - val_mae: 0.5194
Epoch 13/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 173ms/step - loss: 0.6099 - mae: 0.5157 - val_loss: 0.6966 - val_mae: 0.5908
Epoch 14/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 191ms/step - loss: 0.5820 - mae: 0.5211 - val_loss: 0.9170 - val_mae: 0.5702
Epoch 15/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 2s 570ms/step - loss: 0.4783 - mae: 0.4446 - val_loss: 0.5533 - val_mae: 0.4974
Epoch 16/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 165ms/step - loss: 0.3881 - mae: 0.4213 - val_loss: 0.6139 - val_mae: 0.5389
Epoch 17/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 2s 328ms/step - loss: 0.4782 - mae: 0.4646 - val_loss: 0.5092 - val_mae: 0.4033
Epoch 18/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 2s 165ms/step - loss: 0.3395 - mae: 0.3788 - val_loss: 0.4872 - val_mae: 0.3756
Epoch 19/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 145ms/step - loss: 0.2937 - mae: 0.3375 - val_loss: 0.3478 - val_mae: 0.3403
Epoch 20/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 146ms/step - loss: 0.2709 - mae: 0.3219 - val_loss: 0.3625 - val_mae: 0.3397
Epoch 21/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 144ms/step - loss: 0.2377 - mae: 0.2915 - val_loss: 0.3310 - val_mae: 0.3314
Epoch 22/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 180ms/step - loss: 0.2860 - mae: 0.3166 - val_loss: 0.3413 - val_mae: 0.3344
Epoch 23/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 264ms/step - loss: 0.2763 - mae: 0.3148 - val_loss: 0.3093 - val_mae: 0.3151
Epoch 24/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 158ms/step - loss: 0.2777 - mae: 0.3143 - val_loss: 0.2784 - val_mae: 0.2818
Epoch 25/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 155ms/step - loss: 0.2236 - mae: 0.2695 - val_loss: 0.2852 - val_mae: 0.2890
Epoch 26/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 203ms/step - loss: 0.2640 - mae: 0.2975 - val_loss: 0.2959 - val_mae: 0.2941
Epoch 27/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 258ms/step - loss: 0.2343 - mae: 0.2726 - val_loss: 0.2812 - val_mae: 0.2780
Epoch 28/100
```

```
# 9. Evaluate Model
loss, accuracy = model.evaluate(X_test, y_test)
print(f"Test Accuracy: {accuracy:.4f}")
```

```
---------------------------------------------------------------------------
OperatorNotAllowedInGraphError            Traceback (most recent call last)
<ipython-input-42-61f911869286> in <cell line: 0>()
      1 # 9. Evaluate Model
----> 2 loss, accuracy = model.evaluate(X_test, y_test)
      3 print(f"Test Accuracy: {accuracy:.4f}")

                                 1 frames
/usr/local/lib/python3.11/dist-packages/keras/src/utils/traceback_utils.py in error_handler(*args, **kwargs)
    122             raise e.with_traceback(filtered_tb) from None
    123         finally:
--> 124             del filtered_tb
    125
    126     return error_handler

OperatorNotAllowedInGraphError: Exception encountered when calling TimeDistributed.call().

Using a symbolic `tf.Tensor` as a Python `bool` is not allowed. You can attempt the following resolutions to the problem: If you are
running in Graph mode, use Eager execution mode or decorate this function with @tf.function. If you are using AutoGraph, you can try
decorating this function with @tf.function. If that does not work, then you may be using an unsupported feature or your source code may
not be visible to AutoGraph. See
https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/autograph/g3doc/reference/limitations.md#access-to-source-code
for more information.

Arguments received by TimeDistributed.call():
  • inputs=tf.Tensor(shape=(None, 33, 64), dtype=float32)
  • training=False
```

```
# 10. Save Model
model.save("lstm_finger_Sq_model.h5")
```

```
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is consi
```

Start coding or generate with AI.

This give linear output

```
# 5. Train-Test Split
X_train, X_test, y_train, y_test = train_test_split(padded_sequences, encoded_labels, test_size=0.2, random_state=42)

# Reshape y_train and y_test to (num_samples, 1)
y_train = y_train.reshape(-1, 1)
y_test = y_test.reshape(-1, 1)

# Duplicate labels for each timestep in the sequence
y_train_repeated = np.repeat(y_train, max_seq_length, axis=1)
y_test_repeated = np.repeat(y_test, max_seq_length, axis=1)

# Reshape to (num_samples, sequence_length, num_features)
y_train_repeated = y_train_repeated[:, :, np.newaxis]  # Add a new axis for features
y_test_repeated = y_test_repeated[:, :, np.newaxis]  # Add a new axis for features

# Add duration information to y_train_repeated and y_test_repeated
# Assuming your padded_sequences have shape (num_samples, sequence_length, 2)
# where the last dimension contains (finger_id, duration)
# Use X_train and X_test instead of padded_sequences to match the shapes
y_train_repeated = np.concatenate([y_train_repeated, X_train[:, :, 1:]], axis=2)  # Concatenate duration
y_test_repeated = np.concatenate([y_test_repeated, X_test[:, :, 1:]], axis=2)  # Concatenate duration


model = Sequential([
    Masking(mask_value=0.0, input_shape=(max_seq_length, 2)),  # Corrected input shape to (max_seq_length, 2)
    Bidirectional(LSTM(128, return_sequences=True)),
    Bidirectional(LSTM(64, return_sequences=True)),
    Bidirectional(LSTM(32, return_sequences=True)),
    Dropout(0.3),
    Dense(64, activation='relu'),
    TimeDistributed(Dense(2, activation='linear'))  # Output layer for finger ID and duration
])

model.compile(loss='mse', optimizer='adam', metrics=['mse'])
```

```
# 8. Train with Early Stopping
early_stop = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)

model.fit(X_train, y_train_repeated, epochs=100, batch_size=16,
          validation_data=(X_test, y_test_repeated),
          callbacks=[early_stop])
```

```
Epoch 1/100
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/masking.py:47: UserWarning: Do not pass an `input_shape`/`input_dim` ar
  super().__init__(**kwargs)
4/4 ━━━━━━━━━━━━━━━━━━━━ 17s 680ms/step - loss: 1.9941 - mse: 1.9941 - val_loss: 1.2536 - val_mse: 1.2536
Epoch 2/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 267ms/step - loss: 1.0505 - mse: 1.0505 - val_loss: 1.1309 - val_mse: 1.1309
Epoch 3/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 256ms/step - loss: 0.9406 - mse: 0.9406 - val_loss: 1.0770 - val_mse: 1.0770
Epoch 4/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 157ms/step - loss: 0.9341 - mse: 0.9341 - val_loss: 1.1047 - val_mse: 1.1047
Epoch 5/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 159ms/step - loss: 0.8882 - mse: 0.8882 - val_loss: 1.0687 - val_mse: 1.0687
Epoch 6/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 160ms/step - loss: 0.7702 - mse: 0.7702 - val_loss: 1.0241 - val_mse: 1.0241
Epoch 7/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 295ms/step - loss: 0.8532 - mse: 0.8532 - val_loss: 1.0212 - val_mse: 1.0212
Epoch 8/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 264ms/step - loss: 0.8005 - mse: 0.8005 - val_loss: 1.0210 - val_mse: 1.0210
Epoch 9/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 157ms/step - loss: 0.8202 - mse: 0.8202 - val_loss: 1.0572 - val_mse: 1.0572
Epoch 10/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 161ms/step - loss: 0.8570 - mse: 0.8570 - val_loss: 1.0393 - val_mse: 1.0393
Epoch 11/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 157ms/step - loss: 0.8480 - mse: 0.8480 - val_loss: 1.0209 - val_mse: 1.0209
Epoch 12/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 320ms/step - loss: 0.8925 - mse: 0.8925 - val_loss: 1.0121 - val_mse: 1.0121
Epoch 13/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 149ms/step - loss: 0.8134 - mse: 0.8134 - val_loss: 1.0240 - val_mse: 1.0240
Epoch 14/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 275ms/step - loss: 0.8019 - mse: 0.8019 - val_loss: 1.0175 - val_mse: 1.0175
Epoch 15/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 140ms/step - loss: 0.7702 - mse: 0.7702 - val_loss: 1.0003 - val_mse: 1.0003
Epoch 16/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 254ms/step - loss: 0.8087 - mse: 0.8087 - val_loss: 1.0392 - val_mse: 1.0392
Epoch 17/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 271ms/step - loss: 0.7379 - mse: 0.7379 - val_loss: 0.9063 - val_mse: 0.9063
Epoch 18/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 169ms/step - loss: 0.6124 - mse: 0.6124 - val_loss: 0.9362 - val_mse: 0.9362
Epoch 19/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 147ms/step - loss: 0.5574 - mse: 0.5574 - val_loss: 0.8737 - val_mse: 0.8737
Epoch 20/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 147ms/step - loss: 0.5801 - mse: 0.5801 - val_loss: 0.5575 - val_mse: 0.5575
Epoch 21/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 156ms/step - loss: 0.3773 - mse: 0.3773 - val_loss: 0.6634 - val_mse: 0.6634
Epoch 22/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 154ms/step - loss: 0.3463 - mse: 0.3463 - val_loss: 0.4719 - val_mse: 0.4719
Epoch 23/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 164ms/step - loss: 0.2971 - mse: 0.2971 - val_loss: 0.4318 - val_mse: 0.4318
Epoch 24/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 159ms/step - loss: 0.4943 - mse: 0.4943 - val_loss: 1.2792 - val_mse: 1.2792
Epoch 25/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 166ms/step - loss: 0.5370 - mse: 0.5370 - val_loss: 0.4482 - val_mse: 0.4482
Epoch 26/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 150ms/step - loss: 0.3966 - mse: 0.3966 - val_loss: 0.3891 - val_mse: 0.3891
Epoch 27/100
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 161ms/step - loss: 0.3167 - mse: 0.3167 - val_loss: 0.4144 - val_mse: 0.4144
Epoch 28/100
```

```
# 10. Save Model
model.save("lstm_finger_Sqmse_model.h5")
```

```
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is consi
```

Double-click (or enter) to edit

```
import numpy as np
from tensorflow.keras.models import load_model
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import load_model
from tensorflow.keras.losses import MeanSquaredError
```

```
# Load the model and explicitly define the loss function
model = load_model("lstm_finger_Sqmse_model.h5", custom_objects={'mse': MeanSquaredError()})

# 2. Function to preprocess the input sequence (same as before)
def preprocess_input(sequence):
    """Preprocesses a raw sequence for prediction.

    Args:
        sequence: A list of (finger_id, duration) tuples representing the sequence.

    Returns:
        A padded NumPy array representing the sequence.
    """
    finger_ids = [int(pair[0]) for pair in sequence]
    durations = [pair[1] for pair in sequence]
    input_sequence = list(zip(finger_ids, durations))

    # Pad the sequence to match the model's input shape
    padded_sequence = pad_sequences([input_sequence], maxlen=max_seq_length, padding='post', dtype='float32')
    return padded_sequence

# 3. Example new input sequence
new_sequence = [(1, 0.5), (2, 0.3), (3, 0.2)]

# 4. Preprocess the input
processed_sequence = preprocess_input(new_sequence)

# 5. Get model predictions
predictions = model.predict(processed_sequence)

# 6. Interpret the predictions
# predictions will have shape (1, max_seq_length, 2)
# - predictions[0, i, 0] represents the predicted finger ID for timestep i
# - predictions[0, i, 1] represents the predicted duration for timestep i

# Example: Print predictions for each timestep
for i in range(max_seq_length):
    finger_id = predictions[0, i, 0]
    duration = predictions[0, i, 1]
    print(f"Timestep {i + 1}: Predicted Finger ID: {finger_id:.2f}, Predicted Duration: {duration:.2f}")
```

```
WARNING:absl:Compiled the loaded model, but the compiled metrics have yet to be built. `model.compile_metrics` will be empty until you t
1/1 ━━━━━━━━━━━━━━━━━━━━ 2s 2s/step
Timestep 1: Predicted Finger ID: 0.61, Predicted Duration: 0.04
Timestep 2: Predicted Finger ID: 0.67, Predicted Duration: 0.03
Timestep 3: Predicted Finger ID: 0.69, Predicted Duration: 0.01
Timestep 4: Predicted Finger ID: 0.20, Predicted Duration: -0.01
Timestep 5: Predicted Finger ID: 0.20, Predicted Duration: -0.01
Timestep 6: Predicted Finger ID: 0.20, Predicted Duration: -0.01
Timestep 7: Predicted Finger ID: 0.20, Predicted Duration: -0.01
Timestep 8: Predicted Finger ID: 0.20, Predicted Duration: -0.01
Timestep 9: Predicted Finger ID: 0.20, Predicted Duration: -0.01
Timestep 10: Predicted Finger ID: 0.20, Predicted Duration: -0.01
Timestep 11: Predicted Finger ID: 0.20, Predicted Duration: -0.01
Timestep 12: Predicted Finger ID: 0.20, Predicted Duration: -0.01
Timestep 13: Predicted Finger ID: 0.20, Predicted Duration: -0.01
Timestep 14: Predicted Finger ID: 0.20, Predicted Duration: -0.01
Timestep 15: Predicted Finger ID: 0.20, Predicted Duration: -0.01
Timestep 16: Predicted Finger ID: 0.20, Predicted Duration: -0.01
Timestep 17: Predicted Finger ID: 0.20, Predicted Duration: -0.01
Timestep 18: Predicted Finger ID: 0.20, Predicted Duration: -0.01
Timestep 19: Predicted Finger ID: 0.20, Predicted Duration: -0.01
Timestep 20: Predicted Finger ID: 0.20, Predicted Duration: -0.01
Timestep 21: Predicted Finger ID: 0.20, Predicted Duration: -0.01
Timestep 22: Predicted Finger ID: 0.20, Predicted Duration: -0.01
Timestep 23: Predicted Finger ID: 0.20, Predicted Duration: -0.01
Timestep 24: Predicted Finger ID: 0.20, Predicted Duration: -0.01
Timestep 25: Predicted Finger ID: 0.20, Predicted Duration: -0.01
Timestep 26: Predicted Finger ID: 0.20, Predicted Duration: -0.01
Timestep 27: Predicted Finger ID: 0.20, Predicted Duration: -0.01
Timestep 28: Predicted Finger ID: 0.20, Predicted Duration: -0.01
Timestep 29: Predicted Finger ID: 0.20, Predicted Duration: -0.01
Timestep 30: Predicted Finger ID: 0.20, Predicted Duration: -0.01
Timestep 31: Predicted Finger ID: 0.20, Predicted Duration: -0.01
Timestep 32: Predicted Finger ID: 0.20, Predicted Duration: -0.01
Timestep 33: Predicted Finger ID: 0.20, Predicted Duration: -0.01
```

```
import numpy as np
from tensorflow.keras.models import load_model
import pickle

# 1. Load the saved model
model = load_model("lstm_finger_model.h5")

# 2. Load the label encoder
with open('label_encoder_model.pkl', 'rb') as file:
    label_encoder = pickle.load(file)

# 3. Function to preprocess the input sequence
def preprocess_input(sequence):
  """Preprocesses a raw sequence for prediction.

  Args:
    sequence: A list of (finger_id, duration) tuples representing the sequence.

  Returns:
    A padded NumPy array representing the sequence.
  """
  finger_ids = [int(pair[0]) for pair in sequence]
  durations = [pair[1] for pair in sequence]
  input_sequence = list(zip(finger_ids, durations))

  # Pad the sequence to match the model's input shape
  # Assuming max_seq_length is defined or loaded from somewhere
  padded_sequence = pad_sequences([input_sequence], maxlen=max_seq_length, padding='post', dtype='float32')
  return padded_sequence

# 4. Example new input sequence
new_sequence = [(1, 0.5), (2, 0.3), (3, 0.2)]  # Replace with your actual sequence

# 5. Preprocess the input
processed_sequence = preprocess_input(new_sequence)

# 6. Make prediction
prediction = np.argmax(model.predict(processed_sequence), axis=1)

print(prediction)

# 7. Convert prediction back to label
predicted_label = label_encoder.inverse_transform(prediction)[0]  # Use index 0

# 8. Print the predicted gesture
print(f"Predicted Gesture: {predicted_label}")
```

```
WARNING:absl:Compiled the loaded model, but the compiled metrics have yet to be built. `model.compile_metrics` will be empty until you t
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 1s/step
[2]
Predicted Gesture: LikeMe
```

```
import numpy as np
from tensorflow.keras.models import load_model
import pickle
from tensorflow.keras.preprocessing.sequence import pad_sequences  # Import pad_sequences

# 1. Load the saved model
model = load_model("lstm_finger_model.h5")

# 2. Load the label encoder
with open('label_encoder_model.pkl', 'rb') as file:
    label_encoder = pickle.load(file)

# 3. Function to preprocess the input sequence
def preprocess_input(sequence):
  """Preprocesses a raw sequence for prediction.

  Args:
    sequence: A list of (finger_id, duration) tuples representing the sequence.

  Returns:
    A padded NumPy array representing the sequence.
  """
  finger_ids = [int(pair[0]) for pair in sequence]
  durations = [pair[1] for pair in sequence]
```

```
  input_sequence = list(zip(finger_ids, durations))

  # Pad the sequence to match the model's input shape
  # Assuming max_seq_length is defined or loaded from somewhere
  padded_sequence = pad_sequences([input_sequence], maxlen=max_seq_length, padding='post', dtype='float32')
  return padded_sequence

# 4. List of input sequences
sequences = [
    [(1, 0.5), (2, 0.3), (3, 0.2)],
    [(2, 0.4), (3, 0.6), (1, 0.3)],
    [(3, 0.2), (1, 0.5), (2, 0.4)]
    # Add more sequences as needed
]

# 5. Predict and print for each sequence
for sequence in sequences:
    processed_sequence = preprocess_input(sequence)
    prediction = np.argmax(model.predict(processed_sequence), axis=1)
    print(prediction)
    # predicted_label = label_encoder.inverse_transform(prediction)[0]  # Use index 0
    # print(f"Sequence: {sequence}, Predicted Gesture: {predicted_label}")
```

```
WARNING:absl:Compiled the loaded model, but the compiled metrics have yet to be built. `model.compile_metrics` will be empty until you t
1/1 ━━━━━━━━━━━━━━━━━━━━ 2s 2s/step
[[0 3 3 2]]
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 65ms/step
[[3 3 3 2]]
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 66ms/step
[[3 3 3 2]]
```

sample

```
Start coding or generate with AI.
```

```
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Masking, TimeDistributed
# ... (other imports)


# 1. Reshape target data for next-step prediction
y_train_shifted = np.zeros_like(X_train)  # Initialize with zeros
y_test_shifted = np.zeros_like(X_test)

for i in range(len(X_train)):
    y_train_shifted[i, :-5] = X_train[i, 5:]  # Shift target by 5 steps

for i in range(len(X_test)):
    y_test_shifted[i, :-5] = X_test[i, 5:]

# 2. Modify model architecture for sequence prediction
model = Sequential([
    Masking(mask_value=0.0, input_shape=(max_seq_length, 2)),
    LSTM(128, return_sequences=True),
    LSTM(64, return_sequences=True),
    LSTM(32, return_sequences=True),  # Return sequences for all timesteps
    Dropout(0.3),
    Dense(64, activation='relu'),
    TimeDistributed(Dense(2, activation='linear'))  # Predict finger ID and duration
])

# 3. Compile and train the model
model.compile(loss='mse', optimizer='adam', metrics=['mae'])
model.fit(X_train, y_train_shifted, epochs=50, batch_size=16,
          validation_data=(X_test, y_test_shifted), callbacks=[early_stop])
```

```
Epoch 1/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 9s 457ms/step - loss: 2.8193 - mae: 0.9732 - val_loss: 1.7813 - val_mae: 0.8518
Epoch 2/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 105ms/step - loss: 2.1383 - mae: 0.8814 - val_loss: 1.5088 - val_mae: 0.7899
```

```
Epoch 3/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 111ms/step - loss: 2.1504 - mae: 0.8788 - val_loss: 1.4344 - val_mae: 0.7455
Epoch 4/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 87ms/step - loss: 1.8564 - mae: 0.8156 - val_loss: 1.3060 - val_mae: 0.7429
Epoch 5/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 90ms/step - loss: 1.9352 - mae: 0.8282 - val_loss: 1.2549 - val_mae: 0.6866
Epoch 6/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 96ms/step - loss: 1.9016 - mae: 0.8081 - val_loss: 1.1901 - val_mae: 0.6596
Epoch 7/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 92ms/step - loss: 1.6289 - mae: 0.7392 - val_loss: 1.1446 - val_mae: 0.6425
Epoch 8/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 92ms/step - loss: 1.5416 - mae: 0.7131 - val_loss: 1.1211 - val_mae: 0.6276
Epoch 9/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 89ms/step - loss: 1.3676 - mae: 0.6660 - val_loss: 1.0985 - val_mae: 0.6249
Epoch 10/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 91ms/step - loss: 1.4997 - mae: 0.7029 - val_loss: 1.0776 - val_mae: 0.6199
Epoch 11/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 90ms/step - loss: 1.4947 - mae: 0.7037 - val_loss: 1.0527 - val_mae: 0.5986
Epoch 12/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 99ms/step - loss: 1.3407 - mae: 0.6547 - val_loss: 1.0268 - val_mae: 0.5845
Epoch 13/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 88ms/step - loss: 1.3912 - mae: 0.6471 - val_loss: 0.9983 - val_mae: 0.5727
Epoch 14/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 90ms/step - loss: 1.3058 - mae: 0.6282 - val_loss: 0.9553 - val_mae: 0.5481
Epoch 15/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 96ms/step - loss: 1.3063 - mae: 0.6106 - val_loss: 0.9214 - val_mae: 0.5324
Epoch 16/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 145ms/step - loss: 1.3497 - mae: 0.6119 - val_loss: 0.8982 - val_mae: 0.5220
Epoch 17/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 150ms/step - loss: 1.1342 - mae: 0.5561 - val_loss: 0.8877 - val_mae: 0.5162
Epoch 18/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 149ms/step - loss: 1.1376 - mae: 0.5528 - val_loss: 0.8979 - val_mae: 0.5183
Epoch 19/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 160ms/step - loss: 1.0197 - mae: 0.5145 - val_loss: 0.8959 - val_mae: 0.5131
Epoch 20/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 95ms/step - loss: 1.3177 - mae: 0.5955 - val_loss: 0.8827 - val_mae: 0.5110
Epoch 21/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 106ms/step - loss: 1.1418 - mae: 0.5452 - val_loss: 0.8986 - val_mae: 0.5171
Epoch 22/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 95ms/step - loss: 1.1996 - mae: 0.5622 - val_loss: 0.8741 - val_mae: 0.5088
Epoch 23/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 96ms/step - loss: 1.2351 - mae: 0.5717 - val_loss: 0.8619 - val_mae: 0.4993
Epoch 24/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 113ms/step - loss: 1.2382 - mae: 0.5628 - val_loss: 0.8605 - val_mae: 0.4955
Epoch 25/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 105ms/step - loss: 1.2290 - mae: 0.5630 - val_loss: 0.8589 - val_mae: 0.4942
Epoch 26/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 98ms/step - loss: 1.2831 - mae: 0.5665 - val_loss: 0.8695 - val_mae: 0.5005
Epoch 27/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 102ms/step - loss: 1.2331 - mae: 0.5527 - val_loss: 0.8825 - val_mae: 0.5131
Epoch 28/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 97ms/step - loss: 1.2356 - mae: 0.5580 - val_loss: 0.9233 - val_mae: 0.5307
Epoch 29/50
```

Double-click (or enter) to edit

```
# 3. Function to preprocess the input sequence
def preprocess_input(sequence):
  """Preprocesses a raw sequence for prediction.

  Args:
    sequence: A list of (finger_id, duration) tuples representing the sequence.

  Returns:
    A padded NumPy array representing the sequence.
  """
  finger_ids = [int(pair[0]) for pair in sequence]
  durations = [pair[1] for pair in sequence]
  input_sequence = list(zip(finger_ids, durations))

  # Pad the sequence to match the model's input shape
  # Assuming max_seq_length is defined or loaded from somewhere
  padded_sequence = pad_sequences([input_sequence], maxlen=max_seq_length, padding='post', dtype='float32')
  return padded_sequence


# 4. Prediction
def predict_next_5_steps(input_sequence):
    """Predicts the next 5 steps of a sequence."""
    processed_sequence = preprocess_input(input_sequence)
    predictions = model.predict(processed_sequence)
```

```
    next_5_steps = predictions[0, -5:]  # Get last 5 predicted steps
    return next_5_steps


input_sequence = [(1, 0.5), (2, 0.3), (3, 0.2)]  # Example input
predicted_steps = predict_next_5_steps(input_sequence)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-72-731de320fb82> in <cell line: 0>()
      1 input_sequence = [(1, 0.5), (2, 0.3), (3, 0.2)]  # Example input
----> 2 predicted_steps = predict_next_5_steps(input_sequence)

 2 frames
/usr/local/lib/python3.11/dist-packages/keras/src/trainers/data_adapters/__init__.py in get_data_adapter(x, y, sample_weight,
batch_size, steps_per_epoch, shuffle, class_weight)
    123         # )
    124     else:
--> 125         raise ValueError(f"Unrecognized data type: x={x} (of type {type(x)})")
    126
    127

ValueError: Unrecognized data type: x=[(1, 0.5), (2, 0.3), (3, 0.2)] (of type <class 'list'>)
```

Double-click (or enter) to edit

```
def predict_next_5_steps(input_sequence):
    """Predicts the next 5 steps of a sequence and prints them."""
    processed_sequence = preprocess_input(input_sequence)
    #predictions = model.predict(processed_sequence)

    predictions = model.predict(input_sequence)
    next_5_steps = predictions[0, -5:]  # Get last 5 predicted steps

    print("Predicted Next 5 Steps:")
    for i, step in enumerate(next_5_steps):
        finger_id = step[0]  # Assuming finger ID is in the first column
        duration = step[1]  # Assuming duration is in the second column
        print(f"Step {i + 1}: Finger ID: {finger_id:.2f}, Duration: {duration:.2f}")

    return next_5_steps


def predict_next_5_steps(input_sequence):
    """Predicts the next 5 steps of a sequence and prints them."""
    processed_sequence = np.preprocess_input(input_sequence)
    #predictions = model.predict(processed_sequence)

    predictions = model.predict(input_sequence)
    next_5_steps = predictions[0, -5:]  # Get last 5 predicted steps

    print("Predicted Next 5 Steps:")
    for i, step in enumerate(next_5_steps):
        finger_id = step[0]  # Assuming finger ID is in the first column
        duration = step[1]  # Assuming duration is in the second column
        print(f"Step {i + 1}: Finger ID: {finger_id:.2f}, Duration: {duration:.2f}")

    return next_5_steps


input_sequence = [(1, 0.5), (2, 0.3), (3, 0.2)]  # Example input
predicted_steps = predict_next_5_steps(input_sequence)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-81-731de320fb82> in <cell line: 0>()
      1 input_sequence = [(1, 0.5), (2, 0.3), (3, 0.2)]  # Example input
----> 2 predicted_steps = predict_next_5_steps(input_sequence)

 1 frames
/usr/local/lib/python3.11/dist-packages/numpy/__init__.py in __getattr__(attr)
    331             raise RuntimeError("Tester was removed in NumPy 1.25.")
    332
--> 333         raise AttributeError("module {!r} has no attribute "
    334                              "{!r}".format(__name__, attr))
    335

AttributeError: module 'numpy' has no attribute 'preprocess_input'
```

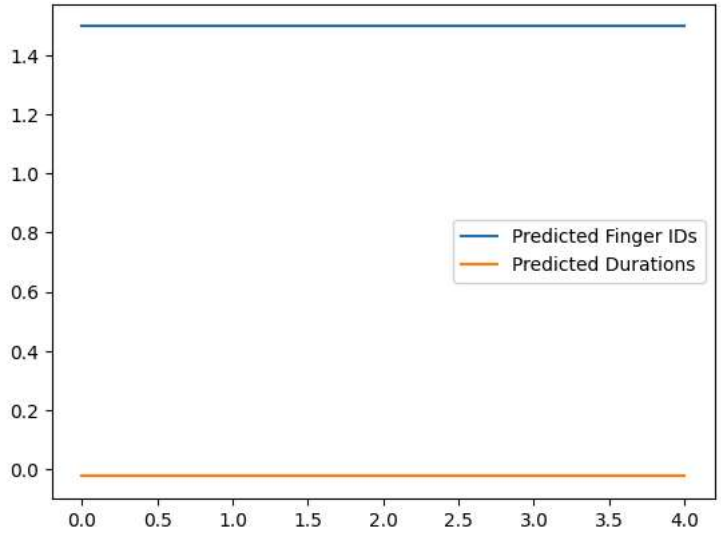
```
# 3. Function to preprocess the input sequence
def preprocess_input(sequence):
  """Preprocesses a raw sequence for prediction.

  Args:
    sequence: A list of (finger_id, duration) tuples representing the sequence.

  Returns:
    A padded NumPy array representing the sequence.
  """
  finger_ids = [int(pair[0]) for pair in sequence]
  durations = [pair[1] for pair in sequence]
  input_sequence = list(zip(finger_ids, durations))

  # Pad the sequence to match the model's input shape
  # Assuming max_seq_length is defined or loaded from somewhere
  padded_sequence = custom_pad_sequences([input_sequence], maxlen=max_seq_length, padding='post', dtype='float32')
  return padded_sequence


def custom_pad_sequences(sequences, maxlen, padding='post', dtype='float32'):
  """Pads sequences with custom values based on label.

  Args:
    sequences: The list of sequences to pad.
    maxlen: The desired length for all sequences.
    padding: 'pre' or 'post' (default 'post').
    dtype: The data type of the padded sequences.

  Returns:
    A NumPy array of padded sequences.
  """
  padded_sequences = []
  for i, seq in enumerate(sequences):  # Use enumerate to get index i
    # Determine padding element based on corresponding label
    label = labels[i]  # Get label for the sequence
    if label == 'CallHarisVai':
      padding_element = ("2", mean_duration)  # Using finger_map dictionary
    elif label == 'Noise':
      padding_element = ("1", mean_duration)  # Using finger_map dictionary
    elif label == 'LikeMe':
      padding_element = ("3", mean_duration)  # Using finger_map dictionary
    elif label == 'ComeCLose':
      padding_element = ("8", mean_duration)  # Using finger_map dictionary
    else:
      padding_element = (int(-1), mean_duration)  # Default padding element

    # Pad the sequence to the desired length
    if len(seq) < maxlen:
      padding_elements = [padding_element] * (maxlen - len(seq))
      if padding == 'pre':
        padded_seq = padding_elements + seq
      else:  # padding == 'post'
        padded_seq = seq + padding_elements
    else:
      padded_seq = seq[:maxlen]  # Truncate if longer than maxlen

    padded_sequences.append(padded_seq)
  return np.array(padded_sequences, dtype=dtype)
```

Double-click (or enter) to edit

```
Start coding or generate with AI.
```

```
import matplotlib.pyplot as plt

# ... (load your model and data)

input_sequence = [(1, 0.5), (2, 0.3), (3, 0.2)]
predicted_steps = predict_next_5_steps(input_sequence)

# Extract predicted finger IDs and durations
predicted_finger_ids = [step[0] for step in predicted_steps]
predicted_durations = [step[1] for step in predicted_steps]

# Plot the predictions
plt.plot(predicted_finger_ids, label="Predicted Finger IDs")
plt.plot(predicted_durations, label="Predicted Durations")
plt.legend()
plt.show()
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 69ms/step
Predicted Next 5 Steps:
Step 1: Finger ID: 1.50, Duration: -0.02
Step 2: Finger ID: 1.50, Duration: -0.02
Step 3: Finger ID: 1.50, Duration: -0.02
Step 4: Finger ID: 1.50, Duration: -0.02
Step 5: Finger ID: 1.50, Duration: -0.02
```



Double-click (or enter) to edit

label output

```
def preprocess_input(sequence):
  """Preprocesses a raw sequence for prediction.

  Args:
    sequence: A list of (finger_id, duration) tuples representing the sequence.

  Returns:
    A padded NumPy array representing the sequence.
  """
  finger_ids = [int(pair[0]) for pair in sequence]
  durations = [pair[1] for pair in sequence]
  input_sequence = list(zip(finger_ids, durations))

  # Pad the sequence to match the model's input shape
  padded_sequence = pad_sequences([input_sequence], maxlen=max_seq_length, padding='post', dtype='float32')
  return padded_sequence
```

```
# Example new input sequence
new_sequence = [(1, 0.5), (2, 0.3), (3, 0.2)]  # Replace with your actual sequence

# Preprocess the input
processed_sequence = preprocess_input(new_sequence)


# Make prediction using the trained model
prediction = np.argmax(model.predict(processed_sequence), axis=1)

# Convert prediction back to label
predicted_label = label_encoder.inverse_transform(prediction)[1]

print(f"Predicted Gesture: {predicted_label}")
```

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

label encoder

```
import json
import numpy as np
from sklearn.preprocessing import LabelEncoder
import pickle

# 1. Load the JSON Data (Assuming your data is in 'finger_sequences.json')
with open("/content/finger_sequences.json", "r") as file:
    data = json.load(file)

# 2. Prepare Labels
labels = []
for label, instances in data.items():
    for _ in instances:  # We only need the labels, not the sequences
        labels.append(label)

# 3. Create and Fit Label Encoder
label_encoder = LabelEncoder()
label_encoder.fit(labels)

# 4. Save the Label Encoder Model
with open('label_encoder_model.pkl', 'wb') as file:
    pickle.dump(label_encoder, file)

print("Label encoder model saved to 'label_encoder_model.pkl'")
```

```
Label encoder model saved to 'label_encoder_model.pkl'
```