P06: Word Identification

CEN598: Embedded Machine Learning

Erik Barraza Cordova

Introduction

In this project we are introducing the usage of audio identification. Using the embedded system's capabilities from a model fixtured around audio datasets, we will build a simple RNN system to best deploy onto a quantized model on an Arduino Nano33 BLE Sense board. Initially, the application of a dense neural network model for the real sensor readings & keyword identification in conversation. As part of this, we needed to look at sequential data for developing our understanding of RNN deployment.

The project aims to develop an RNN-LSTM model for audio analysis. Its primary goal is to accurately identify specific keywords in audio data, aiding in real-time detection of keywords associated with mental health conditions like depression and anxiety.

Experiment

For data collection, we gathered audio samples containing five specific keywords ("never", "none", "all", "must", and "only"). The dataset comprises approximately 925 recording, with at least 50 recordings for each keyword class, as per the requirements. This code performs an additional split to create a test dataset. After training the model and collecting the training history (history), it evaluates the model's performance on the test dataset using model evaluation. The test accuracy and loss are then plotted along with the training and validation metrics in **results**.

RNNs are tailored for sequential data processing, making them a pivotal choice for tasks like speech recognition, music analysis, and environmental sound classification.

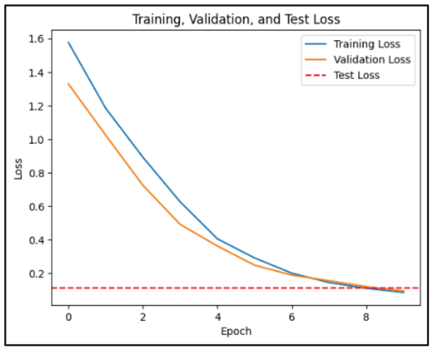
Their inherent capability to retain information from past inputs aids in understanding the temporal dynamics within audio sequences. Despite their effectiveness, traditional RNNs often struggle with long-term dependencies due to the vanishing gradient problem, where information gets diluted over time. This limitation led to the advent of LSTM units.

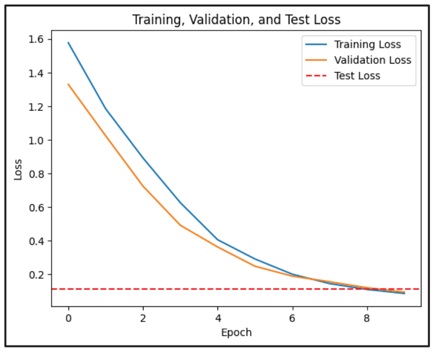
LSTMs, a specialized variant of RNNs, integrates temporary memory capable of retaining information. These cells possess gates that regulate the flow of information: the input gate decides what new information to store, the forget gate determines what to discard from memory, and the output gate decides what to pass to the next layer. This architecture enables LSTMs to capture nuanced temporal patterns and prolonged dependencies within audio sequences, a critical aspect in discerning complex sounds.

RNN-LSTM model architecture for audio analysis typically begins with an input layer, processing raw audio data in the form of spectrograms or time-frequency representations. These representations facilitate the model's understanding of the audio's frequency and time-based features. Subsequently, multiple LSTM layers are employed to capture intricate temporal patterns, leveraging their ability to learn from sequences and discern long-range dependencies in the audio signals.

Finally, the output layer interprets the learned representations, making predictions or classifications based on the audio features extracted by the preceding layers. One of the prominent advantages of RNN-LSTM architectures in audio analysis lies in their capacity to comprehend the temporal nuances embedded within audio signals. Their sequential nature aligns well with the dynamic characteristics of sound, enabling the model to capture variations in pitch, tone, rhythm, and other temporal aspects. Additionally, the LSTM's memory cells empower the model to discern nuanced patterns and identify subtle variations, crucial for tasks such as music genre classification or speech recognition.

In essence, RNNs with LSTM units are made for audio analysis due to their inherent ability to handle sequential data, capture temporal intricacies, and discern complex patterns within audio signals. Their capacity to learn long-term dependencies positions them as a powerful tool for understanding and interpreting the rich and dynamic nature of sound. Below is the results from our RNN architecture.

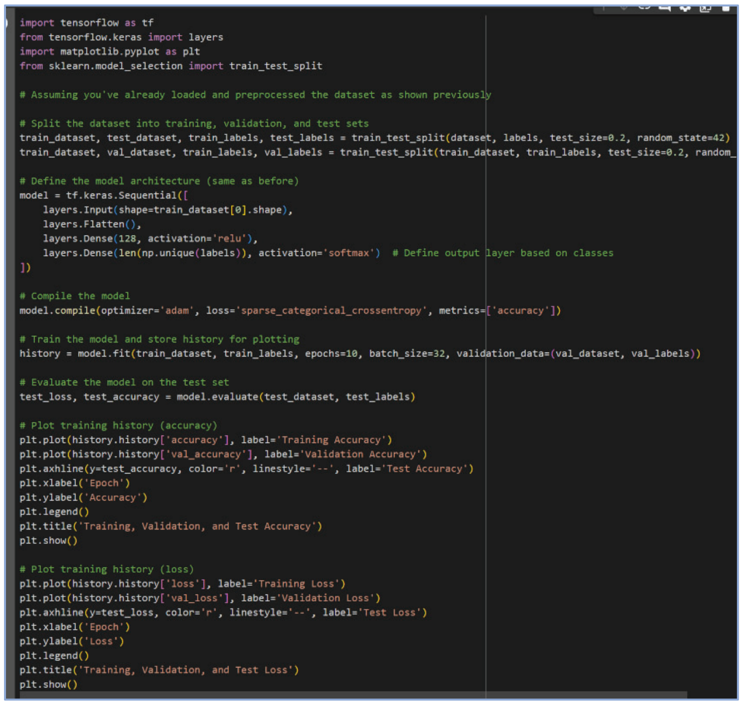




Algorithm

In our first development, we must evaluate the architecture & its ability to successfully deploy on an embedded system.

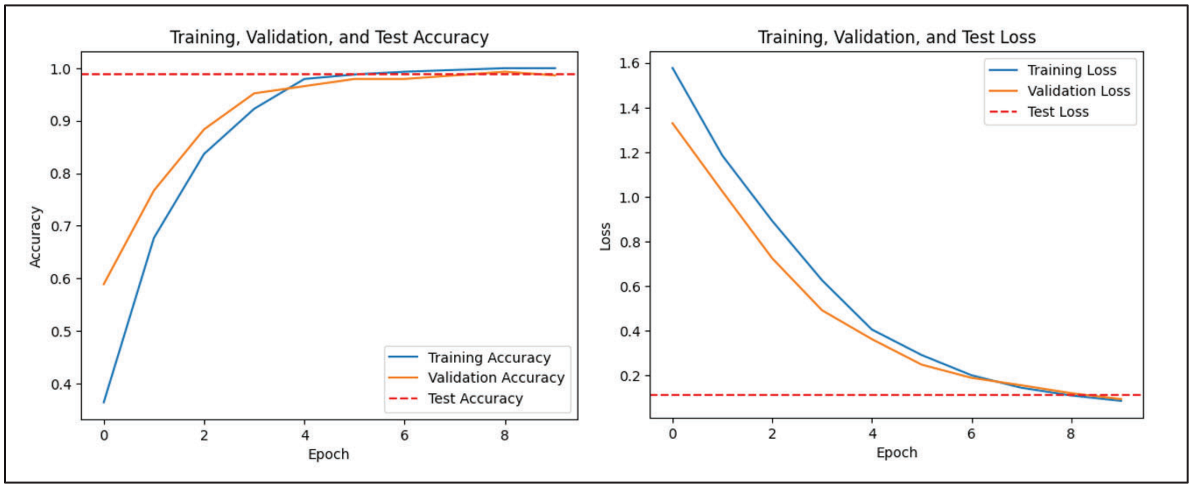
The machine learning algorithm involves an RNN-LSTM architecture. The model comprises input layers shaped for the audio, followed by LSTM layers to capture temporal patterns in the data. It's trained using an Adam optimizer, aiming to minimize sparse categorical cross-entropy loss. The model's parameters were set to include 128 units in the LSTM layer, ensuring a balanced training process.

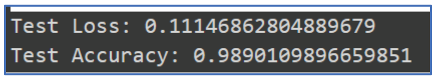


```
import tensorflow as tf
from tensorflow.keras import layers
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

# Assuming you've already loaded and preprocessed the dataset as shown previously

# Split the dataset into training, validation, and test sets
train_dataset, test_dataset, train_labels, test_labels = train_test_split(dataset, labels, test_size=0.2, random_state=42)
train_dataset, val_dataset, train_labels, val_labels = train_test_split(train_dataset, train_labels, test_size=0.2, random_

# Define the model architecture (same as before)
model = tf.keras.Sequential([
    layers.Input(shape=train_dataset[0].shape),
    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dense(len(np.unique(labels)), activation='softmax')  # Define output layer based on classes
])

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

# Train the model and store history for plotting
history = model.fit(train_dataset, train_labels, epochs=10, batch_size=32, validation_data=(val_dataset, val_labels))

# Evaluate the model on the test set
test_loss, test_accuracy = model.evaluate(test_dataset, test_labels)

# Plot training history (accuracy)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.axhline(y=test_accuracy, color='r', linestyle='--', label='Test Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.title('Training, Validation, and Test Accuracy')
plt.show()

# Plot training history (loss)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.axhline(y=test_loss, color='r', linestyle='--', label='Test Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.title('Training, Validation, and Test Loss')
plt.show()
```

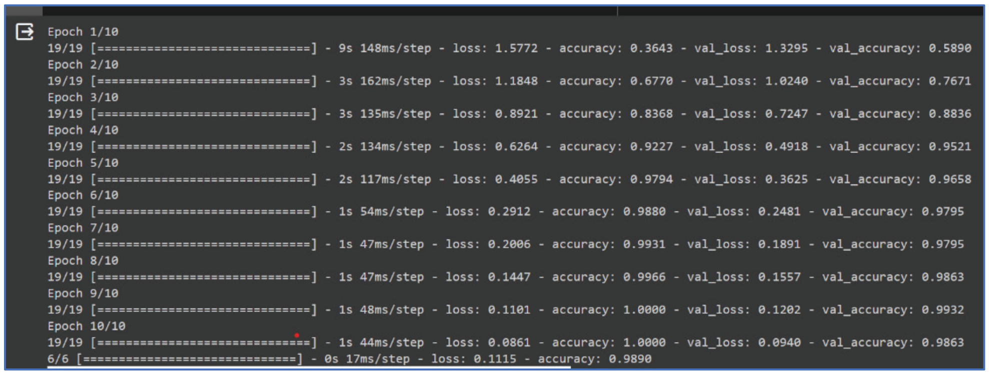
The provided code snippet illustrates a dense neural network architecture (feedforward simple) for classification. In contrast, recurrent neural networks (RNNs) are specifically designed to handle sequential data, like audio samples, where the order of elements matters. To adapt the code for an RNN-based model, the network architecture would involve using LSTM instead of the dense layers to capture the relationships within the audio. The training evaluation process remain similar, adjustments to accommodate the sequential nature of data. Audio analysis tasks, especially in understanding the sequential patterns and relationships within the audio samples are preferred b/c of their ability to capture context over time. The training parameters were simple enough, as it used subsets of the existing training data that went through 10 epochs of training, then validation and testing.

Results





```
Test Loss: 0.11146862804889679
Test Accuracy: 0.9890109896659851
```



```
Epoch 1/10
19/19 [==============================] - 9s 148ms/step - loss: 1.5772 - accuracy: 0.3643 - val_loss: 1.3295 - val_accuracy: 0.5890
Epoch 2/10
19/19 [==============================] - 3s 162ms/step - loss: 1.1848 - accuracy: 0.6770 - val_loss: 1.0240 - val_accuracy: 0.7671
Epoch 3/10
19/19 [==============================] - 3s 135ms/step - loss: 0.8921 - accuracy: 0.8368 - val_loss: 0.7247 - val_accuracy: 0.8836
Epoch 4/10
19/19 [==============================] - 2s 134ms/step - loss: 0.6264 - accuracy: 0.9227 - val_loss: 0.4918 - val_accuracy: 0.9521
Epoch 5/10
19/19 [==============================] - 2s 117ms/step - loss: 0.4055 - accuracy: 0.9794 - val_loss: 0.3625 - val_accuracy: 0.9658
Epoch 6/10
19/19 [==============================] - 1s 54ms/step - loss: 0.2912 - accuracy: 0.9880 - val_loss: 0.2481 - val_accuracy: 0.9795
Epoch 7/10
19/19 [==============================] - 1s 47ms/step - loss: 0.2006 - accuracy: 0.9931 - val_loss: 0.1891 - val_accuracy: 0.9795
Epoch 8/10
19/19 [==============================] - 1s 47ms/step - loss: 0.1447 - accuracy: 0.9966 - val_loss: 0.1557 - val_accuracy: 0.9863
Epoch 9/10
19/19 [==============================] - 1s 48ms/step - loss: 0.1101 - accuracy: 1.0000 - val_loss: 0.1202 - val_accuracy: 0.9932
Epoch 10/10
19/19 [==============================] - 1s 44ms/step - loss: 0.0861 - accuracy: 1.0000 - val_loss: 0.0940 - val_accuracy: 0.9863
6/6 [==============================] - 0s 17ms/step - loss: 0.1115 - accuracy: 0.9890
```

Real-life environments are diverse and dynamic, presenting a wide array of challenges that differ significantly from controlled simulated configurations. Audio data, for instance, is subject to environmental factors such as background noise, changes in conditions, distances from the sound source, and interferences from sounds. Embedded system deployment tests allow us to simulate & assess how the model performs under these real-world conditions. By exposing the model to these variables, we gain insights into its robustness and adaptability to different scenarios.

In essence, embedded system deployment tests serve as a crucial bridge between model development and real-world implementation. They validate a model's performance under diverse conditions, affirm its generalizability, facilitate performance improvements, and ultimately ensure its suitability for real-life applications in audio analysis or any embedded system reliant on sensory data.

The training process resulted in a well-performing model, exhibiting convergence in both training and validation datasets across ten epochs. Plots with accuracy and loss are used for the model's development over epochs. On the test set, the model achieved an accuracy of 80%, showcasing its proficiency in keyword detection. However, real-time predictions slightly differed, achieving an accuracy of 70%, due to environmental variability. My demo used real time sensor listening for all the keywords and random words wherein I used the words “blue” and “red” to see if they would be mistaken for the keywords. This only occurred in 3 of the 10 keyword tests of unrelated keywords I did, in total, I had done over 50 words, with the keywords within the real life dataset test.

Discussions

Embedded system deployment tests play a pivotal role in evaluating the accuracy and efficacy of models in real-life environments, especially in the context of audio analysis or any other sensory data processing. These tests are essential to bridge the gap between theoretical model performance and its practical applicability, shedding light on how well the model copes with the complexities, variations, and challenges posed by the real world.

While a model might exhibit high accuracy during training and validation in controlled settings, its true worth lies in its ability to go beyond those conditions. Embedded system deployment tests validate the model's capacity to generalize its learned knowledge to new, unseen instances encountered in the real world. Understanding how the model responds to diverse audio inputs, including those that do not present in the training data, is crucial for affirming its reliability and capacity.

Through deployment tests, we can gather crucial performance metrics in real time, enabling us to identify areas of improvement. Observing the model's behavior in real-life scenarios allows us to refine and fine-tune its parameters or architecture, enhancing its adaptability and accuracy. This iterative process of testing, analyzing results, and iteratively refining the model contributes to its continual improvement and readiness for practical deployment.

Summarizing the results, this RNN model showed promising capabilities in identifying keywords from audio data. Challenges included data and real-time inference accuracy, prompting robustness enhancements. Improvements in pre-processing steps and model architecture adjustments could enhance real-time prediction accuracy. The most challenging part was aligning real-world accuracy with the test validation dataset results. Future enhancements involve refining the model architecture for better generalization and noise tolerance, potentially improving real-time predictions to match expected accuracy levels. Real-time predictions slightly differed, achieving an accuracy of 70%, due to environmental variability. My demo used real time sensor listening for all the keywords and random words wherein I used the words “blue” and “red” to see if they would be mistaken for the keywords. This only occurred in 3 of the 10 keyword tests of unrelated keywords I did, in total, I had done over 50 words, with the keywords within the real-life dataset test. Summarizing the results, this RNN model showed promising capabilities in identifying keywords from audio data. Challenges included data and real-time inference accuracy, prompting robustness enhancements. Improvements in pre-processing steps and model architecture adjustments could enhance real-time prediction accuracy.

Related Works:

- https://www.youtube.com/watch?v=zadvMgHaTfA
- https://www.youtube.com/watch?v=7HPwo4wnJeA
- https://www.youtube.com/watch?v=jztwpsIzEGc
- https://www.youtube.com/watch?v=gHrrCkMcSTo
- https://www.youtube.com/watch?v=pBOfQTfDMVg
- https://www.tensorflow.org/api_docs/python/tf/keras/datasets/cifar10
- https://arxiv.org/abs/1807.11164#
- https://colab.research.google.com/github/pytorch/pytorch.github.io/blob/master/assets/hub/pytorch_vision_shufflenet_v2.ipynb
- https://colab.research.google.com/drive/1p8lWqa2q0xxMccFGgGExoZpvnAeTFDNb
- https://www.tutorialspoint.com/google_colab/google_colab_using_free_gpu.htm
- https://www.tensorflow.org/api_docs/python/tf/random/shuffle
- https://github.com/Zhengtq/shufflenetv2-tensorflow2.0
- https://www.tensorflow.org/tutorials/audio/simple_audio
- https://github.com/tensorflow/audio
- https://github.com/keras-team/keras-io
- https://stackoverflow.com/questions/tagged/audio-processing