10-Word Speech Recognition

Davide Giuseppe Griffon

Abstract

This document serves as the report for the first task in the "Natural Language Processing" course, instructed by Professor Gražina Korvel, and completed by student Davide Giuseppe Griffon at Vilnius University as part of the Master's program in Data Science.

Contents

1 Introduction

This report provides a comprehensive overview of developing a speech recognition system capable of identifying ten distinct spoken words using a neural network model. The content is organized to guide the reader through each phase of the project, from dataset creation to model evaluation.

An informal tone is employed throughout the report to enhance readability and engagement, directly addressing the reader to foster a shared learning experience.

This document offers a complete account of the project, complementing the accompanying codebase. Together, they provide full documentation of the assignment.

The code is available on GitHub: https://github.com/Griffosx/nlp.

Project Objectives

The objective of this project is to develop a system that recognizes ten distinct spoken words using a neural network model. The project involves five sequential tasks, each building upon the previous one:

1. **Creating a Dataset of Spoken Words:** Collecting and organizing audio recordings of 10 target words to form a comprehensive dataset.

2. **Extracting Relevant Features from Audio Files:** Processing the audio data to extract meaningful features that will serve as inputs for the neural network.

3. **Selecting an Appropriate Neural Network Model for Classification:** Choosing a suitable neural network architecture that can effectively classify the extracted features into the corresponding spoken words.

4. **Training the Model on the Dataset:** Feeding the prepared dataset into the neural network to train it to recognize and differentiate between the ten spoken words.

5. **Evaluating the Model's Performance on Test Data:** Assessing the trained model's accuracy and effectiveness using a separate set of test data to ensure its reliability and generalizability.

Each task is essential and must be completed in sequence to ensure the successful development and implementation of the speech recognition system.

2 Dataset Creation

2.1 Synthesizing Audio Data

Instead of relying on existing datasets, I decided to synthesize the audio data using an online tool called Lovo.ai. Lovo.ai provides APIs that allow the programmatic generation of audio files using different speakers and parameters. It offers more than 100 English speakers with various pronunciations (American, British, Australian, etc.), genders, and ages.

This approach allowed me to select the ten words my model is designed to recognize. Given this freedom, I chose ten words related to natural language processing because they are fundamental concepts in the field and are commonly used, making them relevant for training a robust speech recognition system: *analyze*, *phonetics*, *recognize*, *accents*, *detect*, *emotions*, *transcribe*, *audio*, *extract*, *features*.

By synthesizing the audio, I could control various aspects such as the speaker's voice and the speed of speech, which added diversity to the dataset. In Lovo.ai, it's possible to adjust the speed of speech by setting a float value where 1.0 represents normal speed. To make the model more robust to variations in speaking speed, I randomly selected speeds from the set {0.8, 1.0, 1.2} for each word and speaker. This variability simulates real-world scenarios where speakers may talk faster or slower, thereby improving the model's ability to generalize.

2.2 Dataset Generation Code

The following Python code snippet illustrates how the dataset was generated. (Note: The actual code used is slightly different, but this version is provided for comprehensibility.)

Listing 1: Dataset Generation Script

```python
speeds = [0.8, 1.0, 1.2]
WORDS = ['analyze', 'phonetics', 'recognize', 'accents', 'detect',
         'emotions', 'transcribe', 'audio', 'extract', 'features']
SPEAKERS = get_speakers()  # Function to retrieve speaker list

for word in WORDS:
    for speaker in SPEAKERS:
        speed = random.choice(speeds)
        get_audio(word, speaker, speed)
```

The function `get_audio` performs the API call to Lovo.ai and saves the WAV file to the appropriate folder. For further information, refer to the Lovo.ai API documentation.

The folder containing the actual code used to generate audio is available on GitHub: https://github.com/Griffosx/nlp/tree/main/src/lovoai.

2.3 Dataset Statistics

Using this method, I generated a total of 1,042 audio files, meaning that for each word there are approximately 104 utterances. Some files were removed due to poor quality, which is why the total number is not a multiple of ten. Each speaker recorded each word only once to ensure diversity, as generating the same audio multiple times would result in identical recordings, especially when the speed is also the same. Although Lovo.ai offers the capability to modify the tone of certain speakers (e.g., surprise, excitement, anger), I opted not to add additional audios in this aspect since the model already performed well with the existing dataset.

All audio files generated have the following specifications:

- Sample rate: 44.1 kHz
- Bit depth: 16 bits per sample
- Channels: Mono

By incorporating multiple speakers and varying speech speeds, the dataset captures a wide range of pronunciations and temporal variations. This diversity helps the neural network learn more generalized patterns, making it better equipped to handle new, unseen data. It simulates real-world conditions where users may have different accents and speaking habits. As a result, this diversity contributes to the high accuracy achieved by the developed model.

3 Feature Extraction

Feature extraction is a critical step in developing any machine learning model for audio classification. Raw audio signals are high-dimensional and contain redundant information that may not be directly useful for classification tasks. Therefore, it is necessary to transform these signals into a set of features that effectively capture the essential information required for distinguishing between different classes—in this case, spoken words.

3.1 Spectrograms

For this project, I chose to use spectrograms as the primary feature representation of the audio data. A spectrogram is a visual representation of the frequency spectrum of an audio signal as it varies with time. It is a two-dimensional plot where:

- The **x-axis** represents time,
- The **y-axis** represents frequency, and
- The **color intensity** at each point indicates the magnitude (amplitude) of a particular frequency at a given time.

Spectrograms are particularly useful in speech recognition tasks because they effectively capture both temporal and spectral information of the audio signal, which are crucial for distinguishing between different phonetic elements in spoken words.

3.1.1 Spectrogram Generation Process

To generate spectrograms from the audio data, the following steps were performed:

1. **Signal Framing:** The continuous audio signal was divided into short frames of equal length.
2. **Windowing:** A window function, specifically the Hamming window, was applied to each frame to minimize spectral leakage.
3. **Fourier Transform:** The Discrete Fourier Transform (DFT) was computed on each windowed frame using the Fast Fourier Transform (FFT) algorithm.
4. **Magnitude Spectrum:** The magnitude of the Fourier Transform was calculated to obtain the amplitude of each frequency component.
5. **Power Spectrum:** The magnitude spectrum was squared to obtain the power spectrum, representing the power of each frequency component.
6. **Logarithmic Scaling:** The power values were converted to a logarithmic scale (e.g., decibels) to mimic human auditory perception.
7. **Spectrogram Plotting:** The resulting values were plotted to generate the spectrogram image.

3.1.2 Implementation Details

Instead of relying on existing libraries like LibROSA to perform this transformation, I decided to implement the spectrogram generation process using standard NumPy functions. This approach allowed me to gain a deeper understanding of each step involved in creating spectrograms and provided greater control over the parameters used.

The code used to generate all the spectrograms is available in the file `preprocessing.py`, which can be found at https://github.com/Griffosx/nlp/blob/main/src/model/preprocessing.py.

While the full code is available in the repository, I would like to highlight some key aspects of the implementation:

- **Window Function:** The Hamming window was generated using `numpy.hamming`, which helps in reducing spectral leakage by tapering the beginning and end of each frame.
- **Fast Fourier Transform:** The FFT was computed using `numpy.fft.fft`, an efficient algorithm for computing the DFT.
- **Frame Length:** The number of samples per frame was set to 1024. Given the audio sample rate of 44.1kHz, this corresponds to a frame duration of approximately 23 milliseconds, which is a typical value for speech processing.
- **Silence Removal:** Since the generated audio files had moments of silence at the beginning and end due to their exact durations (either 1 or 2 seconds), a silence removal function (`remove_silence`) was applied before framing and windowing. This function detects and removes periods of silence, focusing the analysis on the actual speech content.
- **Spectrogram Dimensions:** The spectrograms were plotted as square images with dimensions of 700x700 pixels, which provided a good balance between image quality and computational efficiency.

Choosing a frame length of 1024 samples ensures that the FFT is efficient (as 1024 is a power of two) and provides sufficient frequency resolution for the speech signals. Removing silence from the audio files helps focus the model on the actual speech content and reduces unnecessary computation.

3.1.3 Noise Augmentation

In addition to the spectrogram generation, the `preprocessing.py` file contains functions for generating noise-added audio samples and their corresponding spectrograms. These were used during the testing phase to assess the model's robustness to noisy inputs, simulating real-world scenarios where background noise is present.

3.2 Spectrogram Examples

Figure 1 shows examples of spectrograms created for the words *accents* and *phonetics*, with different levels of noise added.

By transforming the audio data into spectrograms, the model can leverage both temporal and frequency information present in the speech signals. This representation serves as an effective input for convolutional neural networks (CNNs), which are adept at extracting spatial hierarchies and patterns from image data.

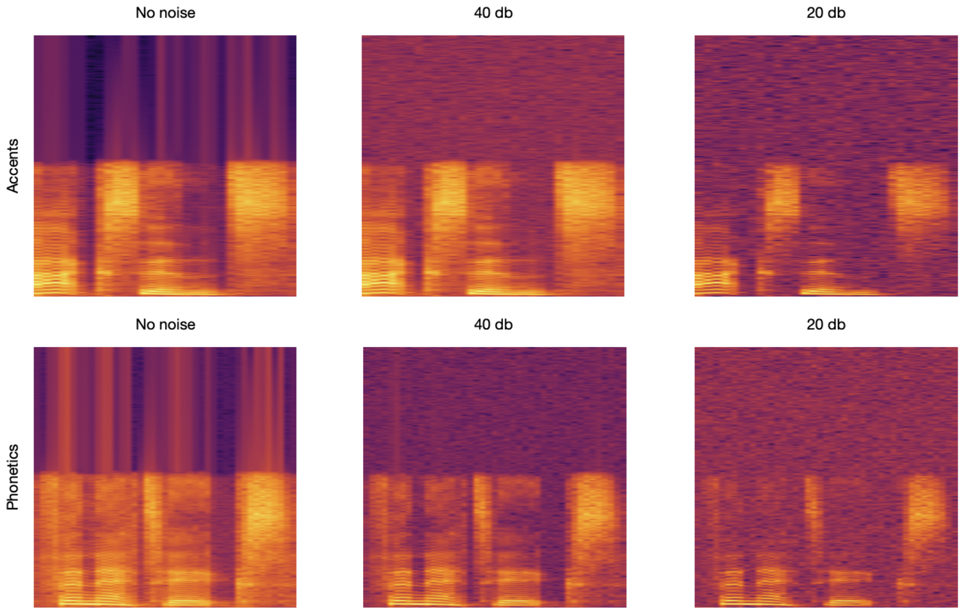

Figure 1: Spectrograms of the words *accents* and *phonetics* with varying levels of noise.

4 Model Selection

4.1 Testing Feasibility with ResNet-18

Since I generated the audio files using an AI tool, I was initially unsure whether, with these particular settings, I could create an effective model from scratch. To test this feasibility, I first conducted a quick experiment using a pre-trained model with the fastai library. I chose fastai because, as far as I know, it's one of the easiest and fastest ways to create a model. For this preliminary test, I selected the ResNet-18 model, as it offers a good balance between trainability and performance, making it a suitable candidate for our needs.

The code used for this test is available at `src/model/resnet/fastai.py` (https://github.com/Griffosx/nlp/blob/main/src/model/resnet/fastai.py). The code is fairly straightforward, so I will just present the results here:

Listing 2: Training Log for ResNet-18

```
epoch      train_loss   valid_loss   accuracy   time
0          2.153422     1.550741     0.432692   00:29
1          1.943995     1.141662     0.639423   00:30
2          1.697767     0.778126     0.778846   00:31
... other logs ...
14         0.176312     0.125081     0.956731   00:30

Classification Report:
              precision    recall  f1-score   support

     accents       1.00      0.85      0.92        13
     analyse       0.95      1.00      0.97        19
       audio       1.00      0.94      0.97        18
      detect       0.96      1.00      0.98        25
    emotions       0.96      1.00      0.98        26
     extract       0.94      0.84      0.89        19
    features       1.00      1.00      1.00        18
   phonetics       1.00      0.94      0.97        18
   recognise       0.90      0.96      0.93        28
  transcribe       0.92      0.96      0.94        24

    accuracy                           0.96       208
```

As shown in the training log, I trained the model for 15 epochs, achieving a final accuracy of approximately 96% on the validation set (without noise), which was promising. This quick test confirmed that it is possible to create a good model using the data I generated. And this is what we will explore in the next section.

4.2 Convolutional Neural Network

A Convolutional Neural Network (CNN) is a powerful and widely used architecture for image processing tasks due to its ability to automatically and adaptively learn spatial hierarchies of features. Building on the success of ResNet-18, which demonstrated strong performance in the previous section, I aimed to develop a simpler CNN model from scratch using only NumPy. This approach provides a deeper understanding of the underlying mechanics of CNNs and offers greater flexibility in customizing the architecture.

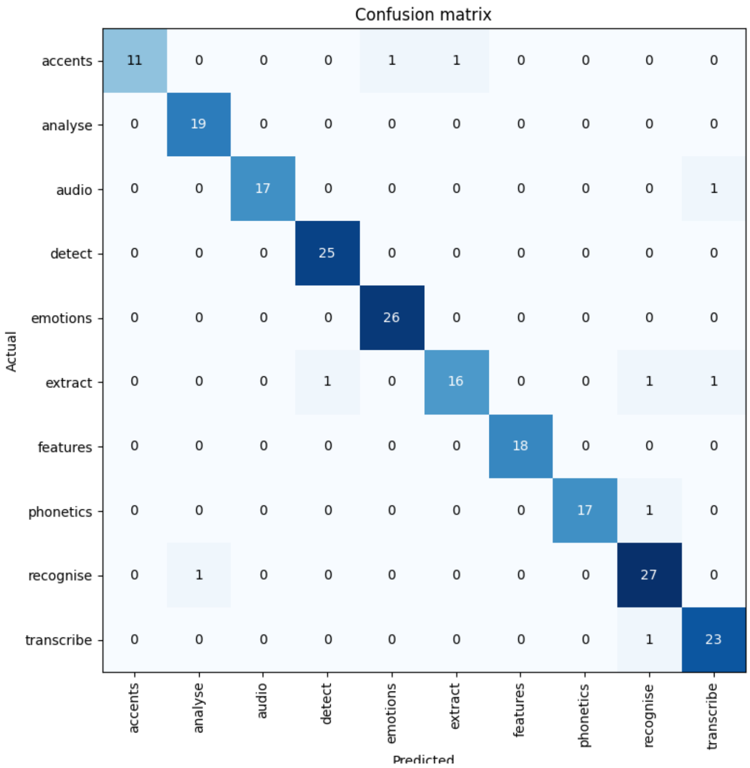

Figure 2: Confusion matrix for the ResNet-18 model.

The CNN model I developed is significantly simpler than ResNet-18, focusing on essential convolutional and fully connected layers without the residual connections that characterize ResNet architectures. The simplicity of the model makes it easier to train and faster to evaluate, while still maintaining competitive performance for the task at hand, as we will see. The architecture is detailed in the following listing, and the complete code is available on GitHub: https://github.com/Griffosx/nlp/blob/main/src/model/cnn/model.py.

Listing 3: CNN Model Architecture

```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 72 * 72, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, num_classes)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 72 * 72)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

Model Architecture Overview

- **Convolutional Layers:** The model begins with two convolutional layers. The first layer takes an input with 3 channels (corresponding to RGB spectrogram images) and applies 6 filters of size 5×5. The second convolutional layer takes the 6 feature maps from the first layer and applies 16 filters of the same size. These layers are responsible for extracting spatial features from the input images.

- **Activation and Pooling:** After each convolutional layer, a ReLU activation function introduces non-linearity, allowing the network to learn more complex patterns. Following the activation, a max pooling layer with a 2×2 window reduces the spatial dimensions of the feature maps. Although only one pooling layer (`self.pool`) is defined in the model, it is applied after both convolutional layers in the forward pass, effectively performing two separate pooling operations. This helps decrease computational complexity and control overfitting by reducing the size of the feature maps.

- **Fully Connected Layers:** After flattening the feature maps, the model includes three fully connected layers. The first two layers (`fc1` and `fc2`) reduce the dimensionality of the data, while the final layer (`fc3`) maps the features to the number of target classes, producing the final class scores.

Supporting Scripts

The `src/model/cnn` directory contains additional scripts that facilitate the training and evaluation of the CNN model:

- `dataset.py`: This script defines the `SpectrogramDataset` class, which handles the loading and preprocessing of spectrogram images. The `get_data_loaders` function is responsible for creating data loaders that efficiently feed data in batches during training and validation.

- `train.py`: This script includes functions for training the CNN model. It manages the training loop, computes loss, updates model weights, and saves the trained model to disk for future use.

- `test.py`: This script contains functions to evaluate the trained model's performance on test data. It includes procedures for testing the model under various noise levels to assess robustness and generates confusion matrices to visualize classification performance across different classes.

By implementing a CNN from scratch and organizing the supporting scripts effectively, this approach ensures a clear and modular workflow for training, evaluating, and refining the speech recognition model.

5 Model Training

6 Model Testing

7 Conclusion

The project successfully developed a speech recognition system capable of recognizing 10 distinct spoken words with high accuracy. By synthesizing a diverse dataset and using a CNN model, the system demonstrated robustness to variations in speakers and speech speeds. Future work could involve expanding the vocabulary, incorporating real-world noisy data, and exploring more sophisticated models like Recurrent Neural Networks or Transformers.

References

1. Lovo.ai API Documentation: `https://docs.genny.lovo.ai/reference/intro/getting-started`

2. Librosa Documentation: `https://librosa.org/doc/latest/index.html`

3. TensorFlow Documentation: `https://www.tensorflow.org/`

4. S. Haykin, *Neural Networks and Learning Machines*, 3rd ed., Prentice Hall, 2008.