O. Paszkiewicz, Krzysztof Ostrzycki

Project AI

Topic: Artificial neural network classifier trained by backpropagation for polish spoken words recognition (PSD dataset)

Project goal

The aim of this project is to develop an artificial neural network classifier that can recognize spoken Polish words from PSD datasets. The system will process the recorded audio, extract spectrogram features, and use backpropagation to train pseudo-neurons using backpropagation. By improving the performance of the model, the project aims to achieve greater accuracy in Polish word recognition.

Dataset

The PSD dataset used in this work contains recordings of Polish words spoken by different authors. Each sentence in the data set contains 500 sentences from a single author. For each sentence, there is an audio file in *.wav format, text in *.txt format, and a *.TextGrid file detailing the start and end times for each word and sound. The number of sentences ranges from 1 to 3000, and the data are arranged in blocks, 500 sentences from different authors in each. This framework provides the labeled audio data needed to train and test neural network classifiers.

Research

In this project, spectrograms are chosen for feature extraction as they provide a visual representation of the audio signal's frequency spectrum over time. Spectrograms effectively capture the temporal and spectral characteristics of the audio signal, making them suitable for speech recognition tasks. They enable the neural network to analyze patterns in the audio that are crucial for distinguishing between different spoken words.

Accuracy has been selected as the primary metric for evaluating the model's performance. This metric measures the proportion of correctly predicted word labels out of the total predictions, offering a clear and effective means to assess the model's effectiveness. Using accuracy ensures that the classifier's ability to correctly identify spoken words is quantified in a straightforward manner.

Libraries used

os: For file and directory operations.

tensorflow: For handling audio file operations and neural network training.

scikit-learn (sklearn): For data preprocessing, model evaluation, and splitting datasets.

keras: For building, training, and evaluating the neural network model.

keras.layers: For defining different layers of the neural network such as convolutional layers, pooling layers, dense layers, and normalization layers.

tensorflow.signal: For generating spectrograms from audio signals.

tensorflow.io: For reading audio files.

tensorflow.data: For efficient data pipeline creation and preprocessing using tf.data.Dataset.

tensorflow.keras.callbacks: For implementing callbacks like early stopping and model checkpointing to optimize the training process.

Designing

Data Preparation

Words preparation:

Initially, sentences from the dataset are split into individual words. The individual words are sorted alphabetically to facilitate organized processing. These alphabetically sorted words are then grouped and stored in the words file. This ensures that the words are readily accessible for further processing and model training. Then the words with more than 100 instances are chosen for training.

Audio File Processing:

Audio files are read using TensorFlow's tf.io.read_file and decoded with tf.audio.decode_wav. Each audio file is converted into a waveform, padded, or truncated to a consistent length (16,000 samples).

Feature Extraction:

```
def load_and_preprocess_audio(file_path, output_sequence_length=16000):
    audio_binary = tf.io.read_file(file_path)
    waveform, sample_rate = tf.audio.decode_wav(audio_binary)
    waveform = tf.squeeze(waveform, axis=-1)

    if tf.shape(waveform)[0] < output_sequence_length:
        padding = output_sequence_length - tf.shape(waveform)[0]
        waveform = tf.pad(waveform, paddings=[[0, padding]])
    else:
        waveform = waveform[:output_sequence_length]

    spectrogram = tf.signal.stft(waveform, frame_length=255, frame_step=128)
    spectrogram = tf.abs(spectrogram)
    spectrogram = spectrogram[..., tf.newaxis]

    return spectrogram
```

Spectrograms are generated from the audio waveforms using Short-Time Fourier Transform (STFT) through tf.signal.stft. The magnitude of the STFT results is taken to form the spectrograms, providing a visual representation of the audio signal's frequency spectrum over time. The spectrograms are then normalized for consistency.

Label Encoding:

```
label_encoder = LabelEncoder()
labels_encoded = label_encoder.fit_transform(labels)
labels_one_hot = to_categorical(labels_encoded)
```

The words (labels) are encoded into numerical values using LabelEncoder. The numerical labels are one-hot encoded to create a format suitable for model training.

Dataset Splitting:

```
X_train, X_temp, y_train, y_temp = train_test_split(file_paths, labels_one_hot, test_size=0.3, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42)
```

The data is initially split into a training set (70%) and a temporary set (30%) using train_test_split. The temporary set (30% of the original data) is then split into validation (15%) and testing (15%) sets. Such a split makes sure that the training set is sufficiently big enough to be able to learn from, and better generalize the features.

Model Architecture

```
model = Sequential()
model.add(Input(shape=(dummy_spectrogram.shape[1], dummy_spectrogram.shape[2], 1)))
model.add(norm_layer)
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', kernel_regularizer=l2(0.001)))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu', kernel_regularizer=l2(0.001)))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(128, kernel_size=(3, 3), activation='relu', kernel_regularizer=l2(0.001)))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(256, activation='relu', kernel_regularizer=l2(0.001)))
model.add(Dropout(0.5))
model.add(Dense(len(words), activation='softmax'))
```

Sequential Model:

The model is constructed using Keras' Sequential API, thanks to which sequential stacking of layers when building neural network is possible.

Convolutional Layers:

Three convolutional layers are added, each followed by a MaxPooling layer. These layers use ReLU activation functions to introduce non-linearity and enable the model to learn complex patterns from the spectrograms, additionally it helps with the vanishing gradient problem and improves training efficiency.

Convolutional help identify relevant audio features from spectrograms, such as phonemes or formants. MaxPooling layers reduce the spatial dimensions of the data, they help reducing the computational load and control overfitting.

Kernel regularizers (L2) are applied to the convolutional layers to prevent overfitting. They do that by penalizing large weights, encouraging model to learn more general features.

Flatten and Dense Layers:

The output of the final convolutional layer is flattened and passed through a dense layer with 256 units (ensuring the model has enough capacity to learn from the data) and ReLU activation.

The Flatten layer transforms the 3D output of the convolutional layers into a 1D vector, which is necessary to connect convolutional layers (operating on spatial data) with dense layers (operating on flat vectors). The dense layer acts as a classifier that learns

to make predictions based on the features extracted by the convolutional layers. ReLU Activation introduces non-linearity and helps the model learn complex patterns.

A Dropout layer with a rate of 0.5 is added to reduce overfitting by randomly setting a fraction of input units to 0 at each update during training. Thanks to that, model will not become too reliant on any particular feature.

Output Layer:

A final dense layer with a softmax activation function is added. This layer converts the outputs into probability distributions over the word classes. Each unit corresponds to a specific word class and the softmax activation makes sure that the outputs are interpreted as probabilities that sum to 1, which makes it easy to interpret the model's prediction and choose the word.

Training

Data:

```
def preprocess_dataset(file_paths, labels, batch_size=64):
    def load_and_preprocess(file_path, label):
        spectrogram = load_and_preprocess_audio(file_path)
        return spectrogram, label

    file_paths_ds = tf.data.Dataset.from_tensor_slices(file_paths)
    labels_ds = tf.data.Dataset.from_tensor_slices(labels)

    dataset = tf.data.Dataset.zip((file_paths_ds, labels_ds))
    dataset = dataset.map(lambda file_path, label: (load_and_preprocess_audio(file_path), label), num_parallel_calls=tf.data.AUTOTUNE)
    dataset = dataset.batch(batch_size).prefetch(buffer_size=tf.data.AUTOTUNE)

    return dataset
```

```
train_dataset = preprocess_dataset(X_train, y_train)
val_dataset = preprocess_dataset(X_val, y_val)
test_dataset = preprocess_dataset(X_test, y_test)
```

The training, validation, and test datasets are created using tf.data.Dataset. The datasets are batched and prefetched to optimize the training process (to make it easier to fit the data into memory during training). It pairs each file path with corresponding label combining them into single dataset. This function additionally allows the TensorFlow to determine the optimal number of parallel calls, speeding up the data preprocessing.

Normalization:

```
norm_layer = Normalization()
dummy_spectrogram = load_and_preprocess_audio(file_paths[0])
dummy_spectrogram = tf.expand_dims(dummy_spectrogram, axis=0)
norm_layer.adapt(dummy_spectrogram)
```

A normalization layer is adapted to the dataset, ensuring that the input data is normalized before being fed into the model.

Model Compilation:

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

The model is compiled with categorical cross-entropy loss, Adam optimizer, and accuracy as the evaluation metric.

Categorical cross-entropy loss is well-suited for multi-class classification problems, our work recognition model being such. It helps in penalizing incorrect classifications, encouraging the model to output probabilities closer to actual class labels.

The Adam optimizer is used because of its efficiency in handling sparse data, and its adaptivity. Adam is used with default parameters.

Accuracy is chosen as evaluation metric because it provides a clear indication how often the model's predictions match the true label, which is import for a word recognition.

Callbacks:

```
early_stopping = EarlyStopping(monitor='val_loss', patience=4, restore_best_weights=True)
model_checkpoint = ModelCheckpoint('best_model.keras', monitor='val_accuracy', save_best_only=True)
```

EarlyStopping and ModelCheckpoint callbacks are implemented to monitor validation loss and save the best model during training. Such implementation is prevention of overfitting (the training process stops when the model's performance on the validation set stops improving). Additionally by stopping the training process early, it reduces the computational resources and time required for training.

For our model we set the patience to four, meaning that after four failed improvements the training will stop.

Training:

The model is trained for 40 epochs with the training dataset, and validation is performed on the validation dataset. While it was defined to train on many epochs thanks to the EarlyStopping the training stops before the 40 epochs.

Testing

The trained model is evaluated on the test dataset to determine its accuracy in recognizing spoken Polish words. The test accuracy is recorded for each training loop (10 loops), providing insight into the model's performance across multiple training sessions.

Results

```
All test accuracies:
Loop 1: 0.7776603102684021
Loop 2: 0.7956307530403137
Loop 3: 0.7980972528457642
Loop 4: 0.8016208410263062
Loop 5: 0.8040874004364014
Loop 6: 0.7988019585609436
Loop 7: 0.8118393421173096
Loop 8: 0.8136011362075806
Loop 9: 0.8114869594573975
Loop 10: 0.8083156943321228
```

The result was on average 80% accuracy while being trained on 50 words (18919 files containing one word, and one word having at least 100 examples), indicating that it is performing quite well for such. The training was additionally performed on a lower number of words, resulting in similar results. The model correctly identifies 8 out of 10 words on average, for language models, achieving such accuracy with a relatively small vocabulary set is promising.

Summary

This project implements a spoken word recognition system for Polish using an artificial neural network classifier trained by backpropagation. The neural network model is trained using a structured pipeline, incorporating convolutional layers to learn complex patterns from spectrograms. The overall success of the project was that it:

- effectively processes the spoken word dataset, extracts spectrogram features, and normalizes the data,
- with the repeated accuracy close to 80%, the model demonstrates effectiveness in recognizing Polish spoken words,
- the clear and modular design of the training pipeline ensures that the system can be further refined and adapted for similar tasks in speech recognition.