

IMPLEMENTING SPEECH PROCESSING USING NEURAL NETWORK IN ARTIFICIAL INTELLIGENCE ON PYTHON

BY
SONOI OGBUEFI*

Abstract

The main goal of this article is to implement the process of the Convolutional Neural Network (CNN) on speech processing in detail. This will be done on Python using certain datasets and with the demonstration using trained data. Another goal for CNNs is to learn the model.

Introduction

The major aim of this paper is the transcription of human speech into spoken words for speech recognition and processing. Human speech signals vary due to certain factors like different speaking styles, environmental noises, etc. and so ASR needs to map each variable-length speech signal into a variable-length sequence of words or phonetic symbols. In handling variable-length sequences alongside modeling, the temporal behavior, Hidden Markov model (HMM) has been proven successful which helps in modeling the temporal behavior of speech signals using a sequence of states in which each is associated with a probability distribution of observations.

Convolution of Neural network and their use in ASR:

Convolution of a neural network can be seen as variation of the standard neural network. CNN enables special network structures, consisting of alternating convolution and pooling layers against using fully connected hidden layers.

Organization of input data to the CNN:

While using CNN for pattern recognition, the input data that will be fed to the CNN needs to be organized as a member of feature maps. The term is similar to the one used in image processing applications, whereby intuitive to organize the input as a two-dimensional (2-D) array, which are pixel values x and y (i.e. the horizontal and the vertical) coordinate indices.

In organizing our speech feature vectors into feature maps which will be suitable for CNN processing, we will use the input image as our reference which can be thought of as a spectrogram with Static, Delta, and Delta-Delta features (i.e. first and second temporal derivatives). In order to reserve this metaphor, we need to apply the inputs that preserve locality in both axes of frequency and time whereby time does not present immediate problems from the standpoint of the locality. CNN consists of a wide amount of context (9 – 15) frames in a single window of input like DNN for speech.

Fig1:

*sogbuefi2019@my.fit.edu

Florida Institute of Technology

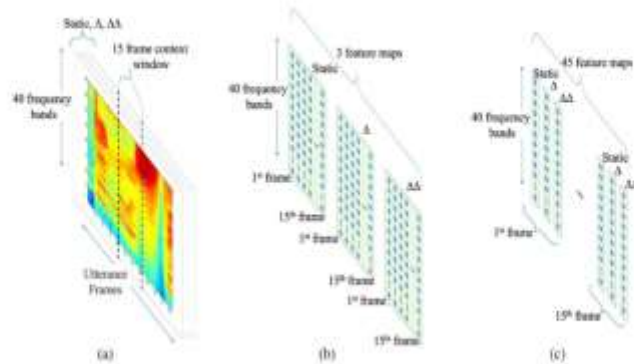


Fig. 1. Two different ways can be used to organize speech input features to a CNN. The above example assumes 40 MFCC features plus first and second derivatives with a context window of 15 frames for each speech frame.

In Fig 1, it shows several different alternatives to organizing the MFCC features into maps for CNN. These can be arranged as three 2-D feature maps, each of which represents MFCC features (Static, Delta, and Delta-Delta) distributed along with both frequency (by using frequency hand index) and time (using the frame number within each context window). Two-dimensional convolution is performed in order to normalize both frequency and temporal variations simultaneously.

Steps taken for implementation:

For the implementation of the speech recognition system in Tensor flow, we need to design the convolution of a neural network, compile, train, test, and reuse it.

NB: using Adam as our optimizer Goal,

Build the architecture of a convolutional neural network (with the tensor flow and Keras)

1. Compiling it
2. Train it and then Test it and save it so that we can reuse it during inference

Steps:

1) Load train/validation/test data splits

- a. `X_train, X_validation, X_test, y_train, y_validation, y_test = get_data_splits(DATA_PATH)`

2) Build the CNN model; the build model builds the CNN by using the function build_model which takes input_shape and learning_rate

- a. `Model = build_model(input_shape, Learning_rate)`
- b. `Learning_rate = 0.0001`; learning rate is the rate at which the model used to learn, as the grading distance. (more like trying to minimize the error).
- c. Input_shape is the shape of the input data that are being fed to the CNN
 - i. Since we are dealing with a CNN that takes a 3D input (1st dimension is the number of segments: In fact, we extracted MFCC at equally space segments and the length of the segment was given by the sample rate divided by the hop length, 2nd is the number of coefficients that we

extracted=13 MFCCs, 3rd is a 1 which is very important because it carries the information about the depth or the channel of an image

3) Train the model

- a. `Model.fit(X_train, y_train, epochs) = EPOCHS = 40` (epoch is the number of time it takes to train a data), `Batch_size, validation_data: (batch_size try to predict`

how the output is going to be at this point running the same data up to 40 times before predicting the output) Which is just `X_valiation` and `Y_Validation`):
Epochs

is the number of times the network is going to see the whole data for training purposes. `Batch_Size` is the number a sample a network we will see before running a backpropagation algorithm

4) Evaluate the model (Evaluate its performance on the test set)

- a. This function return both a `test_error` and a `test_accuracy`
- b. `Model.evaluate()` is another method coming from the Karis API. Takes as parameter `x_test` and `y_test`

5) Save the model

- a. Another great model from Karis which is just `model.save`

Preparing the dataset:

Extract an audio feature for each of the audio feature we have in our dataset and save that in a jsonfile "which is a file that stores data structures and objectives in JavaScript object notation (JSON) format, which is a standard interchange format."

What audio feature are we going to extract?

- 1) The MFCC (Mel Frequency Cepstrum Coefficient) which is commonly used for audio classification. Can be used for;
 - a. Speech recognition
 - b. Music instrument classification
 - c. Detecting different music genres
- 2) The reason we use MFCC is because all the coefficients tell us about the timbre (how the sound is being modelled or represented) of an audio signal (3:17)
- 3) When we extract an MFCC, we take snapshots at different segments of an audio file and these snapshots are the MFCCs coefficient. The type of data we extract is 2 dimensions
 - a. E.g. MFCC (#number of time steps or segment, at each segment we are going to take 13
- 4) We did not build a MFCC from scratch, we used a library (processing library)
 - a. Import librosa

- 5) When we get our audio files, we need to determine the Samples_To_Consider = 22050 which happens to be 1 second worth of sound
- 6) To prepare our data set, we have a function prepare_dataset which takes 4 arguments, the path where the audio files are, the path where we will store our prepare_dataset which is a json file, the number of MFCC coefficient = 13, and the hop length which is measured in number of frames (512) and the last argument is the number of fast Fourier transform. Def prepare_dataset (dataset_path, json_path, n_mfcc, hop_length, n_fft
 - a. Definition of hop_length. When we extract MFCCs, we divide the audio signal equal segment, and then we take snapshots with MFCCs at each of those segments. The hop length tells us how big the segment should be in number of frames.
 - b. Regarding the n_FFT, in fact, when we do the MFCC extraction, we perform a FFT and the n_FFT tells us how big the window of the FFT that we want to consider. N_fft = 2048
- 7) After that, we need to do some mapping. In fact, we can pass a word into a neural network, therefore we need to map keywords to numbers

Codes:

```
import json
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from tensorflow.keras import optimizers

DATA_PATH = "data.json"
SAVED_MODEL_PATH = "model.h5"
EPOCHS = 40
BATCH_SIZE = 32
PATIENCE = 5
LEARNING_RATE = 0.0001

def load_data(data_path):
    """Loads training dataset from json file.

    :param data_path (str): Path to json file containing data
    :return X (ndarray): Inputs
    :return y (ndarray): Targets

    """
    with open(data_path, "r") as fp:
        data = json.load(fp)
```

```

X = np.array(data["MFCCs"])
y = np.array(data["labels"])
print("Training sets loaded!")
return X, y

def prepare_dataset(data_path, test_size=0.2, validation_size=0.2):
    """Creates train, validation and test sets.

    :param data_path (str): Path to json file containing data
    :param test_size (float): Percentage of dataset used for testing
    :param validation_size (float): Percentage of train set used for cross-validation

    :return X_train (ndarray): Inputs for the train set
    :return y_train (ndarray): Targets for the train set
    :return X_validation (ndarray): Inputs for the validation set
    :return y_validation (ndarray): Targets for the validation set
    :return X_test (ndarray): Inputs for the test set
    :return y_test (ndarray): Targets for the test set
    """

    # load dataset
    X, y = load_data(data_path)

    # create train, validation, test split
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test_size)
    X_train, X_validation, y_train, y_validation = train_test_split(X_train, y_train,
test_size=validation_size)

    # add an axis to nd array
    X_train = X_train[..., np.newaxis]
    X_test = X_test[..., np.newaxis]
    X_validation = X_validation[..., np.newaxis]

    return X_train, y_train, X_validation, y_validation, X_test, y_test

def build_model(input_shape, loss='sparse_categorical_crossentropy', learning_rate=0.0001):
    """Build neural network using keras.

    :param input_shape (tuple): Shape of array representing a sample train. E.g.: (44, 13, 1)
    :param loss (str): Loss function to use
    :param learning_rate (float):

    return model: TensorFlow model

```

```
"""
```

```
# build network architecture using convolutional layers
```

```
model = tf.keras.models.Sequential()
```

```
# 1st conv layer
```

```
model.add(tf.keras.layers.Conv2D(64, (3, 3), activation='relu',
```

```
input_shape=input_shape,
```

```
kernel_regularizer=tf.keras.regularizers.l2(0.001)))
```

```
model.add(tf.keras.layers.BatchNormalization())
```

```
model.add(tf.keras.layers.MaxPooling2D((3, 3), strides=(2, 2),
```

```
padding='same'))
```

```
# 2nd conv layer
```

```
model.add(tf.keras.layers.Conv2D(32, (3, 3), activation='relu',
```

```
kernel_regularizer=tf.keras.regularizers.l2(0.001)))
```

```
model.add(tf.keras.layers.BatchNormalization())
```

```
model.add(tf.keras.layers.MaxPooling2D((3, 3), strides=(2, 2),
```

```
padding='same'))
```

```
# 3rd conv layer
```

```
model.add(tf.keras.layers.Conv2D(32, (2, 2), activation='relu',
```

```
kernel_regularizer=tf.keras.regularizers.l2(0.001)))
```

```
model.add(tf.keras.layers.BatchNormalization())
```

```
model.add(tf.keras.layers.MaxPooling2D((2, 2), strides=(2, 2),
```

```
padding='same'))
```

```
# flatten output and feed into dense layer
```

```
model.add(tf.keras.layers.Flatten())
```

```
model.add(tf.keras.layers.Dense(64, activation='relu'))
```

```
tf.keras.layers.Dropout(0.3)
```

```
# softmax output layer
```

```
model.add(tf.keras.layers.Dense(10, activation='softmax'))
```

```
optimiser = tf.keras.optimizers.Adam(learning_rate=learning_rate)
```

```
# compile model
```

```
model.compile(optimizer=optimiser,
```

```
loss=loss,
```

```
metrics=['accuracy'])
```

```
# print model parameters on console
```

```

model.summary()
return model
def train(model, epochs, batch_size, patience, X_train, y_train, X_validation, y_validation):
:param epochs (int): Num training epochs
:param batch_size (int): Samples per batch
:param patience (int): Num epochs to wait before early stop, if there isn't an
improvement on accuracy
:param X_train (ndarray): Inputs for the train set
:param y_train (ndarray): Targets for the train set
:param X_validation (ndarray): Inputs for the validation set
:param y_validation (ndarray): Targets for the validation set
:return history: Training history

    earllystop_callback = tf.keras.callbacks.EarlyStopping(monitor='acc', min_delta=0.001,
patience=patience)
    # train model
    history = model.fit(X_train,
                        y_train,
                        epochs=epochs,
                        verbose = 1,
                        batch_size=batch_size,
                        validation_data=(X_validation, y_validation),
                        callbacks=[earllystop_callback])
    return history

def plot_history(history):
    """Plots accuracy/loss for training/validation set as a function of the epochs

    :param history: Training history of model
    :return:
    """
    fig, axs = plt.subplots(2)

    # create accuracy subplot
    axs[0].plot(history.history['acc'], label="accuracy")
    axs[0].plot(history.history['val_acc'], label="val_accuracy")
    axs[0].set_ylabel("Accuracy")
    axs[0].legend(loc="lower right")
    axs[0].set_title("Accuracy evaluation")

    # create loss subplot
    axs[1].plot(history.history["loss"], label="loss")
    axs[1].plot(history.history["val_loss"], label="val_loss")
    axs[1].set_xlabel("Epoch")

```

```

    axs[1].set_ylabel("Loss")

    axs[1].legend(loc="upper right")
    axs[1].set_title("Loss evaluation")
    plt.show()

def main():
    # generate train, validation and test sets
    X_train, y_train, X_validation, y_validation, X_test, y_test = prepare_dataset(DATA_PATH)

    # create network
    input_shape = (X_train.shape[1], X_train.shape[2], 1)
    model = build_model(input_shape, learning_rate=LEARNING_RATE)

    # train network

    history = train(model, EPOCHS, BATCH_SIZE, PATIENCE, X_train, y_train, X_validation,
                    y_validation)
    # plot accuracy/loss for training/validation set as a function of the epochs
    plot_history(history)

    # evaluate network on test set
    test_loss, test_acc = model.evaluate(X_test, y_test)
    print("\nTest loss: {}, test accuracy: {}".format(test_loss, 100*test_acc))
    # save model
    model.save(SAVED_MODEL_PATH)

if __name__ == "__main__":
    main()

import json
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from tensorflow.keras import optimizers

DATA_PATH = "data.json"
SAVED_MODEL_PATH = "model.h5"
EPOCHS = 40
BATCH_SIZE = 32
PATIENCE = 5
LEARNING_RATE = 0.0001

```



```

def load_data(data_path):
    """Loads training dataset from json file.

    :param data_path (str): Path to json file containing data
    :return X (ndarray): Inputs
    :return y (ndarray): Targets

    """
    with open(data_path, "r") as fp:
        data = json.load(fp)

    X = np.array(data["MFCCs"])
    y = np.array(data["labels"])
    print("Training sets loaded!")
    return X, y

```

```

def prepare_dataset(data_path, test_size=0.1, validation_size=0.2):

```

References:

- O. Abdel-Hamid, A. Mohamed, H. Jiang, L. Deng, G. Penn and D. Yu, "Convolutional Neural Networks for Speech Recognition," in *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 22, no. 10, pp. 1533-1545, Oct. 2014, doi: 10.1109/TASLP.2014.2339736.

Brownlee J. How to Configure the Learning Rate When Training Deep Learning Neural Networks. <https://machinelearningmastery.com/>. [accessed 2020 Jul 8].

<https://machinelearningmastery.com/learning-rate-for-deep-learning-neural-networks/#:~:text=The%20amount%20that%20the%20weights,range%20between%200.0%20and%201.0.>