Optical Speech Recognizer (OSR)

Robert Valencia, valencra, 001131844
Sicheng Xu, xus6, 001218605
Michael Tang, tangc2, 001147873
Wenjie Zhao, zhaow32, 001308084

EE 4BI6 Biomedical Capstone Design Project
Submission date: April 25, 2017
Supervisor: Dr. Hubert de Bruin

McMaster University
Faculty of Engineering
Department of Electrical and Computer Engineering

TABLE OF CONTENTS:

1. ABSTRACT

1.1 Background

Dysarthria is a type of motor speech disorder. It results from impaired movement of the muscles used in speech production, including but not limited to the lips, tongue, vocal folds, and/or diaphragm. [1] The severity and type of dysarthria depends specifically on which area of the nervous system is affected.[1] A person with dysarthria may demonstrate the following speech characteristics: slurred, choppy, mumbled, slow speech with an abnormal pitch and rhythm that may be difficult to understand.[2]

Dysarthria is mainly caused by damage to the brain. This may occur at birth, as is the case for cerebral palsy or muscular dystrophy, but it may also occur later in life due to one of many different conditions that involve the nervous system, including stroke, brain injury, tumors, Parkinson's disease, Lou Gehrig's disease/amyotrophic lateral sclerosis (ALS), Huntington's disease, and multiple sclerosis.[2]

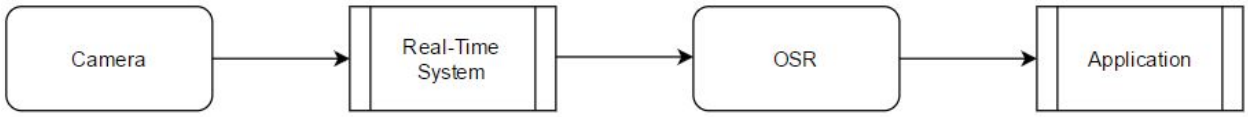
Currently, there are no known data about the incidence of dysarthria in the general population because of the wide variety of possible causes. However, to gauge the sample population, take into consideration the fact that 452 people suffer serious brain injury every day in Canada. This means that, annually, acquiring a brain injury in Canada is 44 times more common than spinal cord injuries, 30 times more common than breast cancer, and 400 times more common than HIV/AIDS.[3]

Currently, treatment for dysarthria is dependent on the cause, type, and severity of the symptoms. A speech-language pathologist works with the individual to improve communication abilities. Treatment usually aims to accomplish some, if not all, of the following goals: slow the rate of speech, improve breathing support to enable the person to speak louder, strengthen muscles, increase tongue and lip movement etc.[4] Often, teaching caregivers, family members, and teachers strategies to better communicate with the person with dysarthria is the most viable solution. In severe cases, a patient must learn to use alternative means of communication (e.g., simple gestures, alphabet boards, or electronic or computer-based equipment).

1.2 Scope

The purpose of this research project is to design a system which serves as a universal speech articulation tool to all forms of dysarthria. Our design utilizes machine learning algorithms in conjunction with real time acquisition in order to predict words being said. Current approaches and devices place a lot of physical demand on the patient, and we aim to flip that narrative by creating a system that can predict speech from visual oral cues.

2. SYSTEM ARCHITECTURE



The overall structure of the system, as shown in the block diagram above, consists of a series of subsystems, each with their own important desired task. The first is the real-time implementation component, which takes a raw video input feed and converts it into a suitable format for the OSR system. Then, the OSR system utilizes machine learning algorithms to make a prediction on what word was said. Finally, we can relay the output of the OSR system to an external control system in order to achieve a desired task.

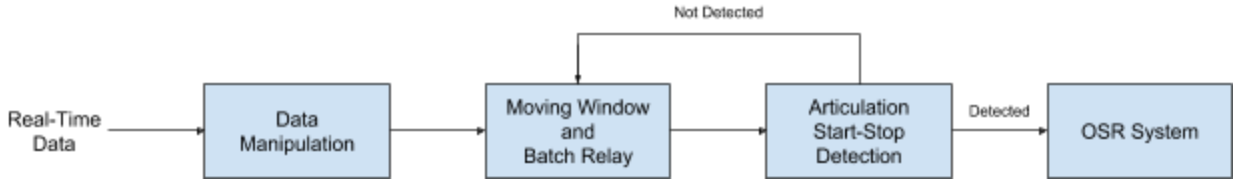
To implement the system, the following technology stack was utilized:

Python was used for general-purpose programming. Theano and SciPy were used for scientific computing. OpenCV was used for computer vision. Finally, Keras was used for implementing neural networks.

3. REAL-TIME DATA ACQUISITION

The real-time processing ability is designed to work in conjunction with the OSR's offline capability to provide a more practical solution to all aforementioned scenarios. Theoretically, a camera is running on idle all the time, able to capture and successfully identify the word articulated by the user.

To ensure a proper workflow of the real-time data acquisition, we decided to segregate the problem into individual parts and work on them separately. The plan is to implement two different algorithms. The first algorithm is responsible for automatically identifying the start and stop of an articulation. Then, the second algorithm is designed to be able to efficiently and selectively choose only one batch of frames which has the highest possibility of containing an articulation cycle. Finally, these pieces are to be integrated together with the OSR system to produce a functional real-time system. This is shown in the following flow chart:



3.1 Preprocessing of Real-Time Data

The format of the input data from the camera is different from what the OSR uses. In order to make sure that everything can run seamlessly, it is important to first manipulate the data so that the the format of the output of the proposed data acquisition process is consistent with that of the samples used during the training phase of the OSR model.

The first step of this stage is colour conversion. The individual frames captured by the camera are all in RGB colours. This means the format of each frame is a 3-D matrix. This matrix is composed of 3 layers, with each representing one of the 3 primary colours and each one of the

elements representing an intensity level. The results of this step are grey scaled images of each frame.

The next step is to localize the mouth region. This step is done prior to the detection on the start and stop of the articulation because the mouth region contains the most statistically relevant data, which is what the entire OSR system is based on. On top of that, we also want to minimize the amount of data being fed into the system by removing the relatively invariant pixels in each frame e.g. eyes, nose, and hair. The outputs of this step would be images of the mouth area. The dimension of each picture could vary significantly depending on the distance between the camera and the user.

In the last step, each frame obtained in the previous step is compressed and the dimension is normalized to 150 x100 pixels. This normalization maintains the consistency of the data while further reducing the memory consumption.

By the end of this preprocessing stage, we would expect the output to be a continuous sequence of grey scaled frames. All the frames are of the same size and each one contains a picture of the mouth region at a different time slot.

3.2 Articulation Start-Stop Detection

The problem of detecting the start and stop of an articulation can be simplified by accurately capturing the opening and closing of user's mouth. However, we are assuming that the lips do not move unless the person starts the articulation to avoid any ambiguities. Also, the user would only pronounce the word defined by one of the three classes: yes, no or null. We initially had two ideas on how to tackle this problem. The first idea utilizes OpenCV and Haar cascades to further localize the exact positions of the upper and lower lips. The computer can then decide whether the the user has started talking by checking if the displacement between upper and lower lips has passed a pre-defined threshold. The second idea focuses on the colour change within a sequence of frames. Theoretically, when the mouth is open, there would be a region that has a darker colour intensity in contrast to the surroundings. Again, by comparing the ratio of the darker and lighter region with a predefined threshold value, the computer is able to identify the start of an articulation.

Both ideas use a comparison approach. However, the displacement method provides a more accurate result than the second one, because it is more immune to errors that are introduced by a low colour contrast, which could be a potential problem when the user has a darker skin colour. An insufficient ambient lighting could also affect the results of such implementation. Thus, we decided the displacement approach is a better solution for detecting the start and stop of an articulation.

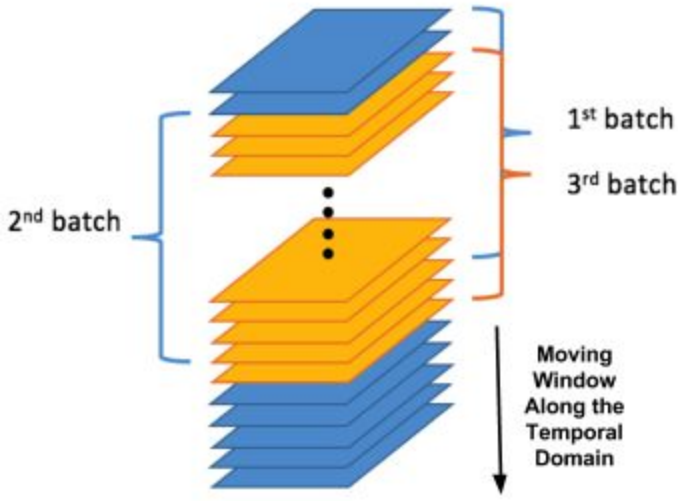
The displacement method starts by locating the lips. This introduces 3 pairs of markers which are put at the centre and two ends of the upper and lower lips. The vertical displacement between each pair of markers are calculated by the computer. The displacements are then summed up and divided by 3 to obtain an average value. Finally, this value is compared to a threshold value of 15 pixels.

With this algorithm, the real-time system is able to analyze each of the incoming frames and automatically make decisions on which batch of data should be used.

3.3 Moving Window and Frame Batch Relay

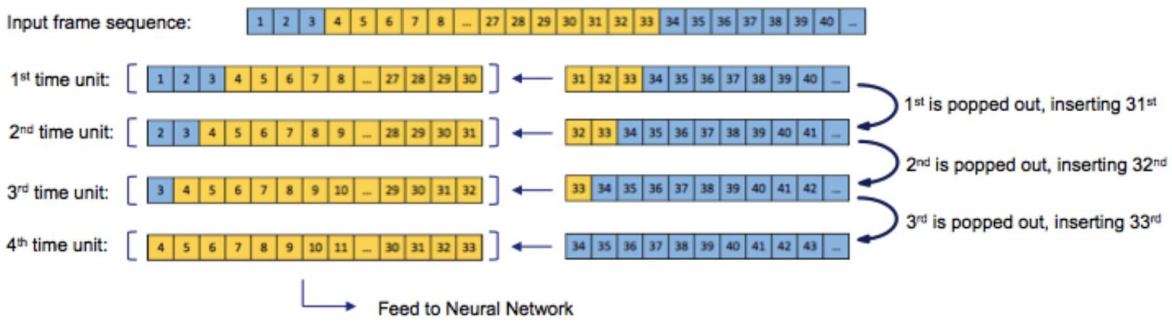
The main challenge with batch relay is memory management. Since we are dealing with real time data, we are getting a constant influx of data. This means we need an algorithm that can efficiently manage the memory consumption. More specifically, it needs to be able to accurately identify the data that we do not need and remove them before everything starts to accumulate and the computer runs out of memory storage. This also brings up another important aspect, which is time delay, since a more efficient algorithm would also reduce the lags that occur during the data acquisition process.

As mentioned earlier, the real-time processing ability works in a similar fashion as the training phase of the OSR model, in the sense that a window slides across the temporal domain and tries to capture the articulation cycle. Here, we are making an assumption that the user would always be able to finish articulating the word within a 30-frame time window. This is equal to one second with a camera that records at 30fps (frames per second). Thus, it is intuitive to use a batch size of 30 frames when analyzing the real-time data. Using a time difference of 1 frame between each frame, the algorithm is illustrated below:



The program starts running once the camera activates. It continuously collects frames until the first batch is formed. The program then analyzes the data contained in this batch to decide whether an articulation has been made within this time window. Because it would be too time-consuming to go through each frame and introduce unnecessary time delay, the program utilizes the “Articulation Start-Stop Detection” algorithm on the first frame only. This idea is motivated by the fact that there has to be a batch in which its first frame indicates the beginning of an articulation. Note that because we assume that the maximum length of an articulation is 30 frames, we do not care about when the user ends the articulation, and that it will always be less or equal to the full length of a window (batch).

The above process repeats itself for every single batch that forms until the target batch is found. One thing to keep in mind is that the analogy of “Moving Window” is used solely for a better visualization of this algorithm. The window actually follows the behaviour of a queue FIFO (first in first out) operation, as illustrated in the figure below:

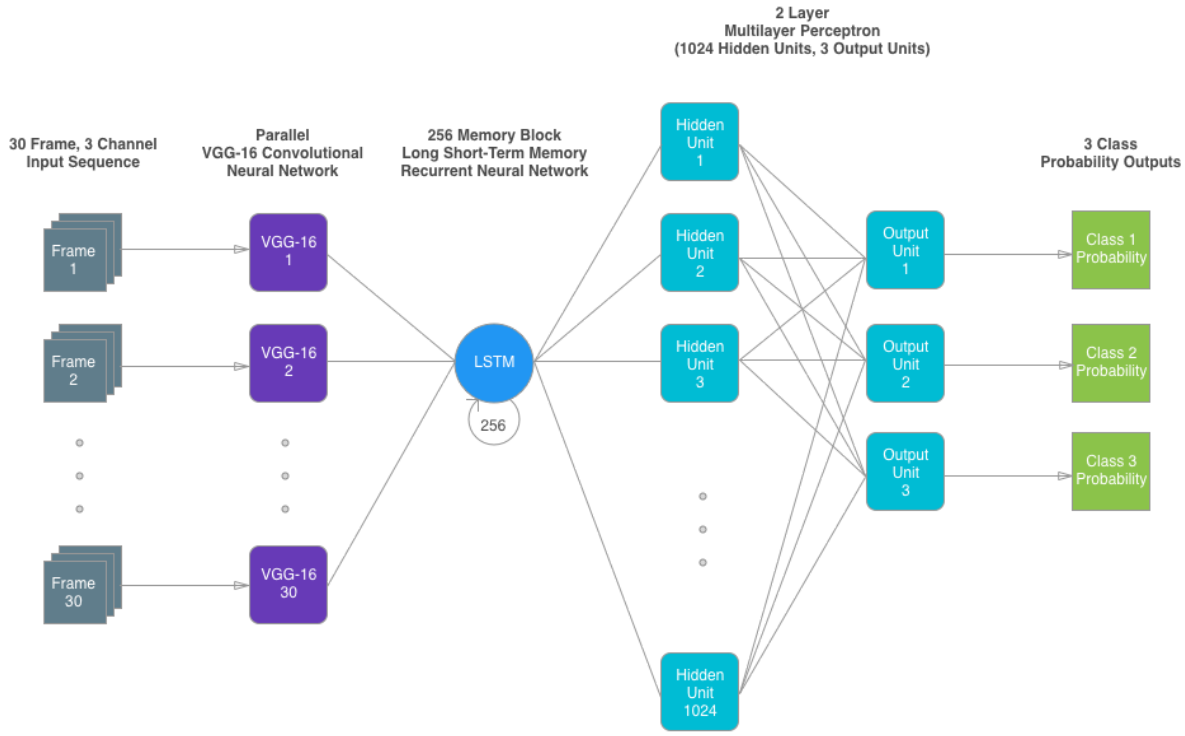


The window is fixed in the temporal space, and a sequence of frames is feed into this window. If the “Articulation Start-stop Detection” algorithm decides that the first frame of current batch does not have a clear indication on the start of an articulation, this frame is popped out and the next available frame is inserted to the end.

Once the system captures the target batch, it will be sent to the OSR model to be processed by the trained system.

4. OPTICAL SPEECH RECOGNITION (OSR)

4.1. OSR System Architecture



The OSR system has been configured to expect inputs, training labels, and outputs of particular dimensionalities. Inputs are preprocessed to have 30 frames, 3 colour channels, 100 row pixels, and 150 column pixels. Training labels are designated as one-hot encoded representations of one out of all possible class, e.g. [0, 0, 1] for the class 1 in a 3-class system. Finally, outputs are returned as arrays containing predicted probabilities for each class, e.g. [0.75, 0.25, 0.25] denotes 75% probability for class 1, and 25% probability for both classes 2 and 3.

Optical speech recognition is a multi-class classification problem, where each unique word is a class. Inputs are sequences of images, so classification depends on spatial and temporal patterns. Given these conditions, machine learning via neural networks is a viable approach. A 3-stage neural network architecture was implemented to address the task, which contain the following components:

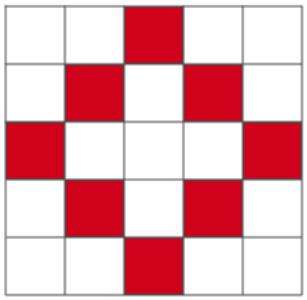
1. **Convolutional Neural Networks (CNNs)**, which utilize trainable filter units to find spatial patterns across images.
2. **Recurrent Neural Networks (RNNs)**, which utilize trainable memory units to find temporal patterns across sequences.
3. **Multilayer Perceptrons (MLPs)**, which utilize trainable artificial neurons to find correlations between features and classes

4.2. Convolutional Neural Networks (CNNs)

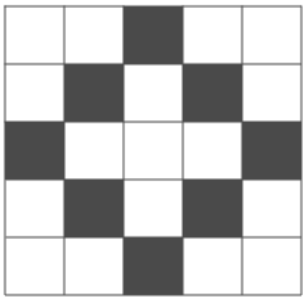
Convolutional neural networks, or CNNs, are biologically-inspired multilayer perceptrons (MLPs, *discussed in section 4.5*) that mimic the visual cortex. In a biological visual cortex, a complex arrangement of cells are sensitive to stimuli within a restricted region called a receptive field. These regions tile across the entire visual field. These cells act as localized filters for detecting spatial patterns, whose response can be approximated by a convolution operation:

$$\begin{aligned}(f * g)[n] &\stackrel{\text{def}}{=} \sum_{m=-\infty}^{\infty} f[m]\, g[n-m] \\ &= \sum_{m=-\infty}^{\infty} f[n-m]\, g[m].\end{aligned}$$

To illustrate how CNNs operate, a sample 5x5 image is shown below. To simplify analysis, the image is converted to grayscale, reducing the colour channels from 3 to 1, and its simplified digital representation is shown, where 1 is the maximum pixel value instead of 255, and 0 remains as the minimum value:



Sample Image

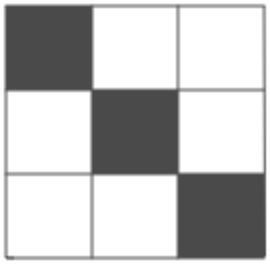


Grayscale Representation

0	0	1	0	0
0	1	0	1	0
1	0	0	0	1
0	1	0	1	0
0	0	1	0	0

Simplified Digital Representation

Below, a sample 3x3 filter is shown, as well as its simplified digital representation:

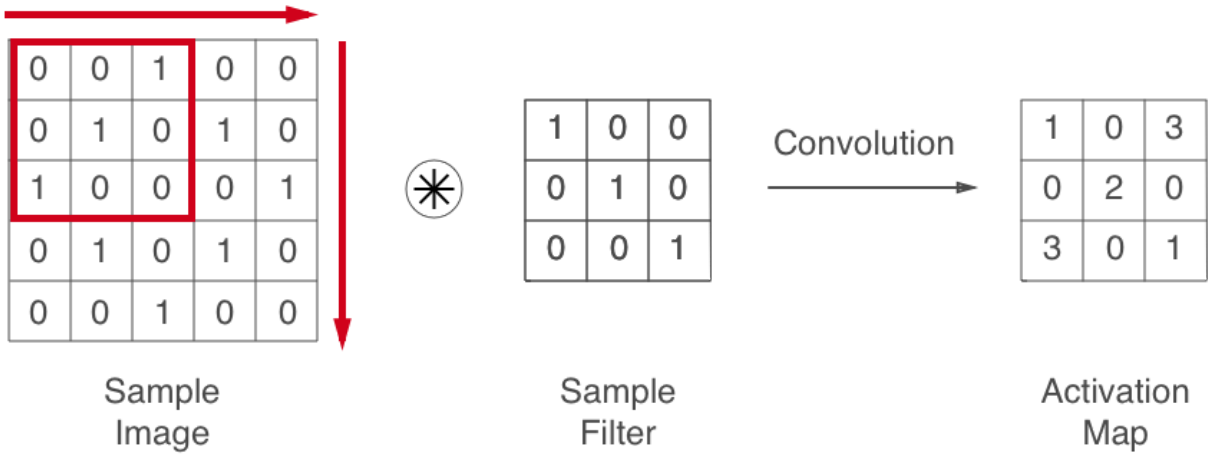


Sample Filter

1	0	0
0	1	0
0	0	1

Simplified Digital Representation

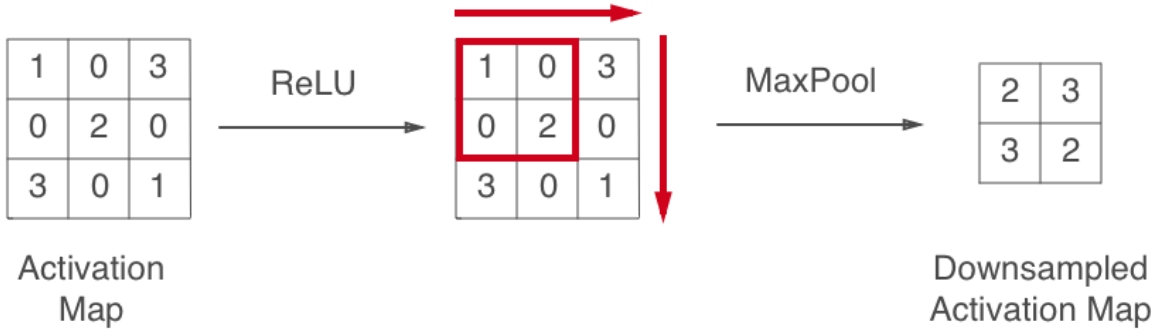
In CNNs, convolutional filters are tiled across an image, convolving receptive fields with corresponding regions of the visual field, until an activation map is generated. In the activation map, regions with spatial patterns that have a high correlation with the filter's pattern have high activation values, and vice versa. In the example below, the filter has a stride of 1 pixel, generating a 3x3 activation map:



After an activation map has been generated, the activation map passes through a layer of rectified linear units (ReLUs), which is an activation function that improves the nonlinearity of the neural network:

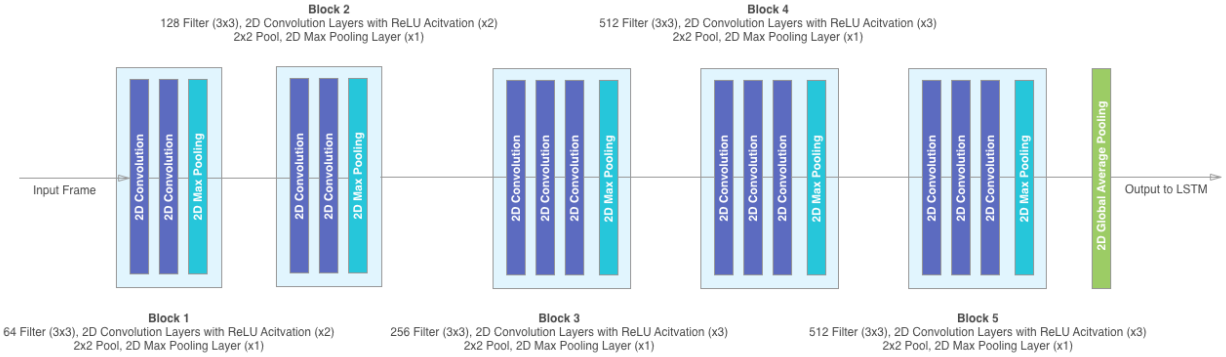
$$f(x) = \max(0, x)$$

After applying a layer of ReLUs, a pooling layer is used to downsample the activation map, reducing the number of parameters, which leads to less computation and less risk of overfitting. In the example below, a type of pooling layer called MaxPool, which replaces a pool of values with its maximum value, with a pool size of 2x2 and stride of 1, is used:



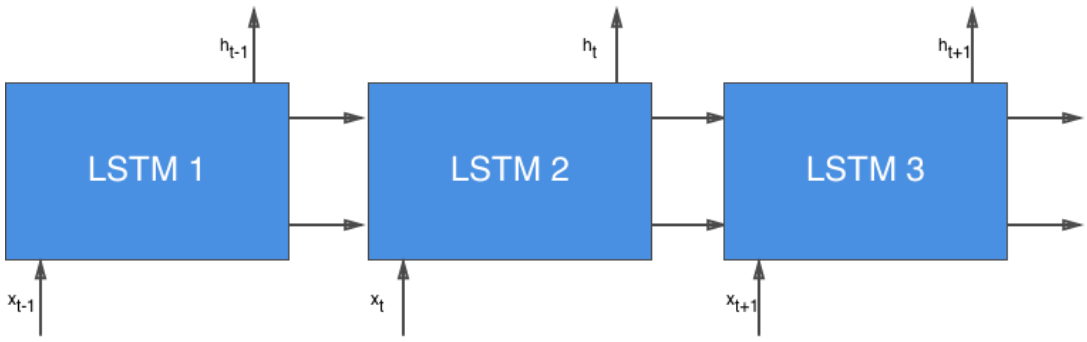
In practical CNN applications, a series of alternating Convolution, ReLU, and MaxPool layers are utilized, where the filter count, filter dimensions, pooling dimensions, and other hyperparameters are part of the design process. Also, the final layer is typically succeeded by fully connected layers and a loss layer, but for this application with inputs which posses spatio-temporal characteristics, the final layer is connected to LSTM units, a type of RNN, which is well-suited for finding temporal patterns.

For the OSR system, a specific CNN architecture called VGG-16 was implemented. It consists of a series of blocks, each of which containing a cascade of 2D Convolution, ReLU, and MaxPooling layers, ending with a 2D Global Average Pooling layer before continuing on to further processing within LSTM units:



4.3. Recurrent Neural Networks (RNNs)

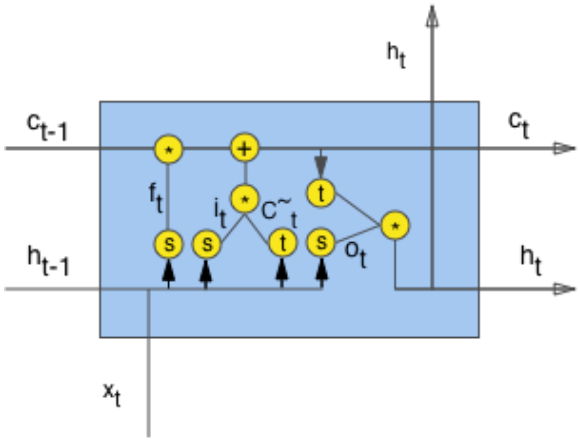
Recurrent neural networks (RNNs) are artificial neural networks, which consists of stateful memory units that are cyclically connected, enabling it to recognize temporal patterns within arbitrary sequences of data. A type of RNN, called Long short-term memory (LSTM), which, unlike regular RNNs, are well suited for data with variable gaps between important events, such as those observed in speech due to demographic and biological variability, is used for the OSR system. LSTMs consist of chains of repeated LSTM units:



Within each LSTM unit, a series of operations occur, represented by yellow circles, where *s* represents the sigmoid function,

$$S(t) = \frac{1}{1 + e^{-t}}$$

t represents the hyperbolic tangent function, * represents element-wise multiplication, and + represents element-wise addition:



To start, the LSTM unit calculates the values of the input gate, i_t, and the candidate value for the memory cell state at time *t*, $\tilde{C}_t$:

$$i_t = s(W_i x_t + U_i h_{t-1} + b_i)$$
$$\tilde{C}_t = t(W_c x_t + U_c h_{t-1} + b_c)$$

Then, the LSTM unit calculates the value of the memory cell forget gate at time *t*, f_t:

$$f_t = s(W_f x_t + U_f h_{t-1} + b_f)$$

Next, given the previously-calculated values, the LSTM unit calculates the memory cell's new state at time *t*, C_t:

$$C_t = i_t * \tilde{C}_t + f_t * \tilde{C}_{t-1}$$

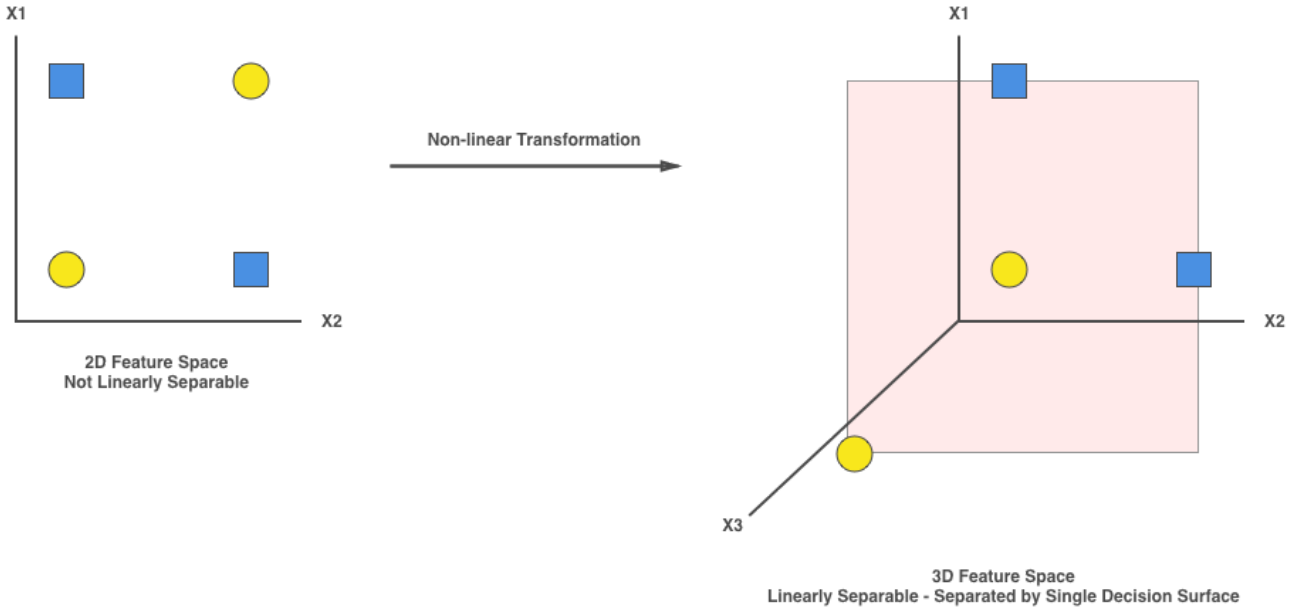
Finally, with the memory cell's new state, the LSTM unit calculates the output:

$$o_t = s(W_o x_t + U_o h_{t-1} + V_o C_t + b_o)$$
$$h_t = o_t * t(C_t)$$

In the previous equations, x_t is the input at time t, Ws, Us, and Vs are weight matrices, and bs are bias vectors.

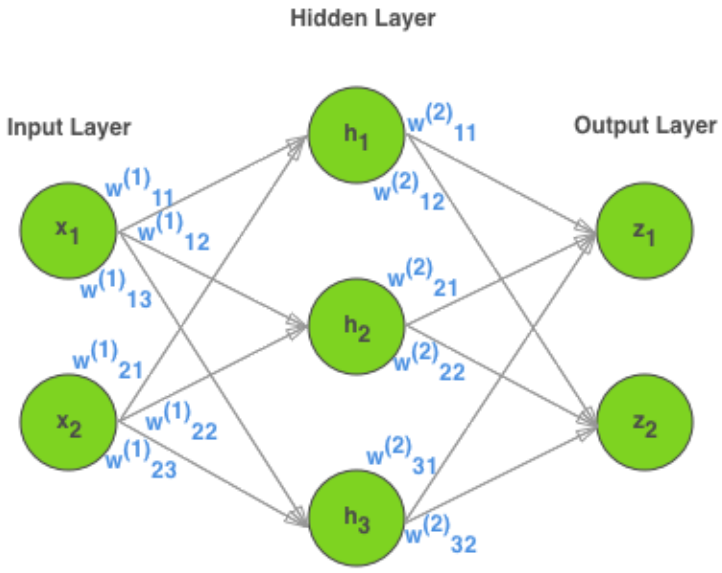
4.4. Multilayer Perceptrons (MLPs)

Multilayer perceptrons (MLPs) are artificial neural networks, which consist of fully connected layers of nodes. MLPs map input data into outputs via a learned non-linear transformation, which projects input data into a space where they become linearly separable, enabling classification. To illustrate this, shown below is a two-class classification problem, represented by blue squares and yellow circles, with a two-dimensional feature space, which is not linearly separable, is processed through a non-linear transformation, projecting the input data into a three-dimensional feature space, making it linearly separable by a single decision surface:



MLPs consists of three primary stages: an input layer, hidden layers, and an output layer. With even a single hidden layer, an MLP becomes a universal approximator, which guarantees that

any function, regardless of complexity, can be approximated by a certain configuration of the MLP, given an appropriate weight matrix learned via training. However, in practical deep learning applications, multiple hidden layers are utilized to generate more features. For illustration purpose, a sample MLP with a single hidden layer is shown:



In the example above, the input layer has 2 nodes, the hidden layer has 3 nodes, and the output layer has 2 nodes. Each input node represents input features. Each hidden node represents generated features. Each output node represents class probabilities. To make predictions with the network, forward propagation is performed, which can be described mathematically as:

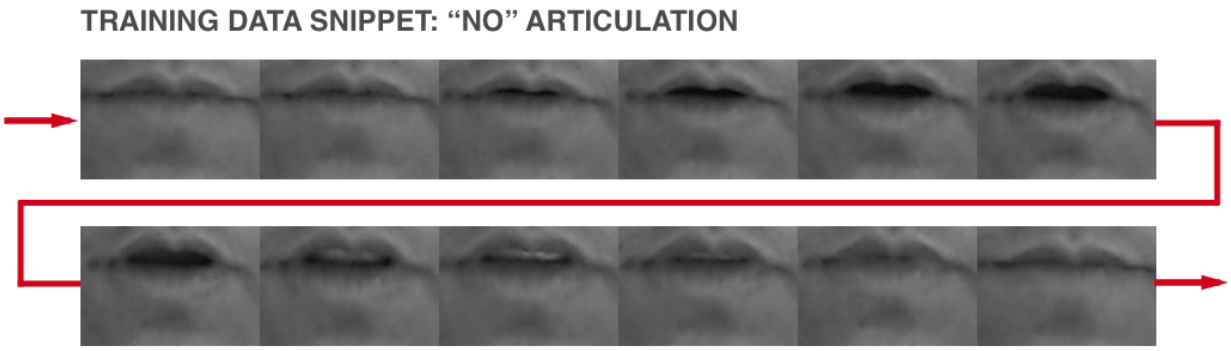
$$h(x) = s(b^{(1)} + w^{(1)}x)$$
$$z(h(x)) = G(b^{(2)} + w^{(2)}h(x))$$

Where h is the hidden layer vector, which contains generated features, and z is the output layer vector, which contains class probabilities. $b^{(1)}$ and $b^{(2)}$ are bias vectors. $w^{(1)}$ and $w^{(2)}$ are weight matrices, which comprise the set of parameters for the neural network to learn. s is the activation function for the hidden layer, which typically is either the *tanh* or *logistic* sigmoid function, but for the OSR model, the *ReLU* function was selected. G is the activation function for the output layer, which is set to the *softmax* function for multi-class classification, such as the task for the OSR model:

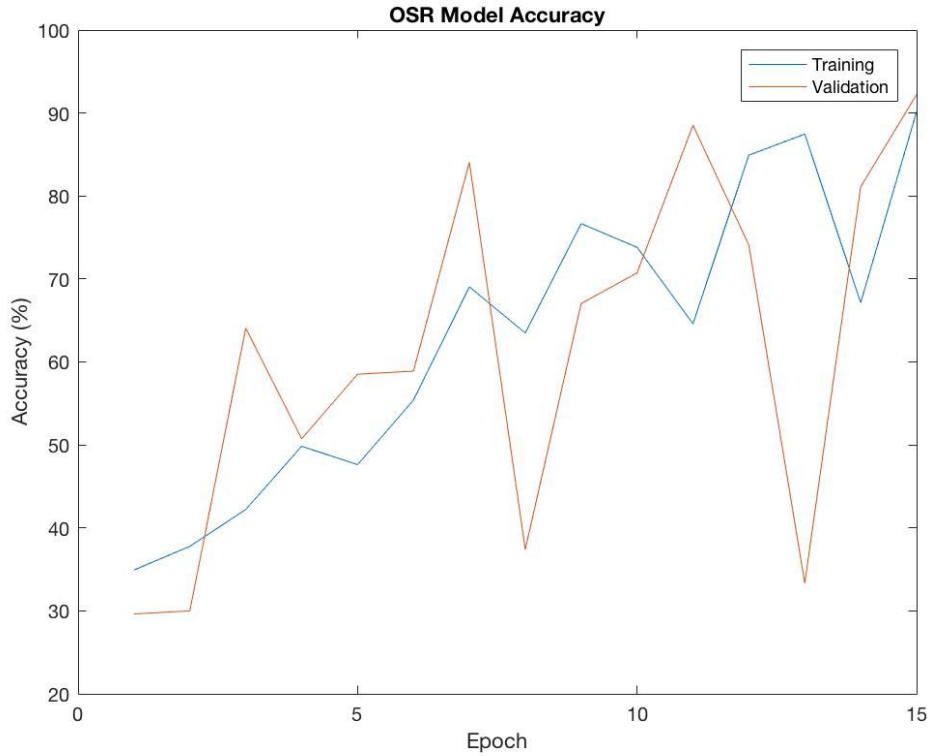
$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{for } j = 1, \ldots, K.$$

At the beginning, parameters are randomized, which would result in a low prediction accuracy. To improve the accuracy, these parameters have to be learned by training the model on labelled data via the backpropagation algorithm, which updates the weights after processing batches of samples. The discussion of the backpropagation algorithm in detail is out of the project's scope, and will be left out for supplementary reading.

4.5. Performance



The OSR model was trained to classify 3 classes: "yes", "no" and "null". 90 samples we collected, or 30 samples per class, which was increased to 900 samples with image augmentation, performed via random rotations, shifts, shears, and zooms. Each sample was preprocessed to grayscale, localized to the oral region, reshaped to a dimensionality of 30 frames, 3 colour channels (all gray), 100 row pixels, and 150 column pixels, and labeled with one-hot encoded expected outputs. Finally, the sample data was partitioned into 70% for training and 30% for validation.



The OSR model was trained over 15 epochs, each of which running through all 900 training and validation samples, with a batch size of 32 sequences. At the end, approximately 90% accuracy was attained for both training and validation phases.

5. APPLICATION

5.1 Future development and Improvement

Enable the possibility to classify more classes:

A possible option is to collect more data sets for various words or phrases and then follow the current implementation scheme in order to identify more class; The same strategy is used for training, which utilizes 70% of the sample data for training and 30% for validation. One important aspect to take into consideration is that it takes a significant amount of time for training the system and verifying the existing neural network in order to achieve a desired accuracy level. Another option is to construct another neural network, with more nodes and redefined parameters for each node.

Both methods are desirable in their own ways but a tradeoff must be made between the time efficiency and training difficulty in order to decide which one is more suitable. Namely, if given the same number of classes for classification, which approach would execute in a more efficient and accurate manner? Moreover, it is important to consider expansion compatibility for possible future development and upgrades**.**

Hardware improvement for higher efficiency:

Artificial neural networks show a promising capability for speech recognition and pattern classification, each of which consists of a number of layers and thousands of neurons. Although it is a powerful and efficient tool for pattern classification, it largely depends on the accessibility to a high performance computer to provide a reasonable run speed.

According to several authoritative researches on hardware development of neural networks, we could] overcome neural network implementation problems by utilizing a specially designed VLSI (very large-scale integration) processors,. [16]

Enhancement on the accuracy of the preprocessing stage

The priority of completing speech pattern recognition is to properly identify the faces and mouths from raw input images. Namely, the system is supposed to be capable of tracking the facial area and localizing the mouth with minimal time delay.

For the pre-processing steps on facial tracking, Haar cascades [17] is the powerful tool used in the system. Basically, the Haar Cascades utilizes a huge comparison database, which contains hundreds and thousands of both positive images (image with target feature) and negative images (image without target feature).

The accuracy of Haar Cascade depends on the variety and number of images used to train the classifiers. During the pre-processing step of the OSR system, the result of localizing mouth by utilizing the Haar Cascade mouth detection xml file is not optimal. Instead of localizing a single mouth with a rectangle frame, it identifies four non-mouth regions as mouths. To fix this problem, we operate a for-loop so as we would only define the lowest green rectangle as “mouth”.

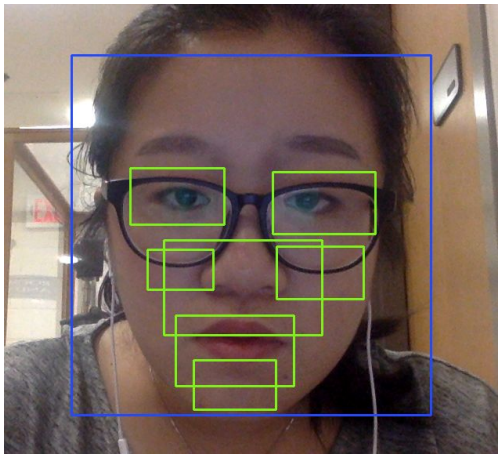


Figure 5.1.1: Incorrect Localization

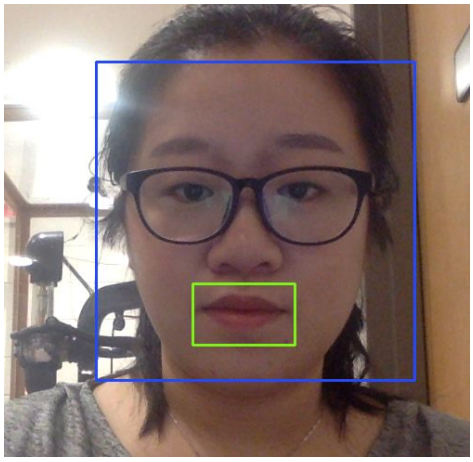


Figure 5.1.2: Correct Localization

Expansion of database:

The current stage of OSR system is able to achieve a 3-class classification with an accuracy of up to 95%. However, it is merely based on training samples from a small group of people (less than 3). Theoretically, the speech pattern varies dramatically among different people, which indicate the fact that in the future, OSR system may undergo a decrease in accuracy when being introduced to a larger testing group. In the future, it is appropriate and necessary to collect training phase data from a larger group of people so as the system would be more stable and obtain more reliable results.

5.2. Future Applications

Automobile Optical Command System

It is widely acknowledged that the voice command system in automobile is highly honorable and beneficial especially during driving on highway. The audio command recognition system works by distinguishing voice signals for accomplishing jobs without manually pressing buttons. This system positively improves the driving security levels. However, this audio input-depended system may fail to complete the command when being interfered by with by surrounding environment noise.

OSR is capable of receiving driver's command under noisy environment, performing as a powerful assistant to the original audio command system. Ideally, it can further improve the security and convenience level while driving.

Smart Home Automation

Smart home automation [18] aims to build a smart automation control system for households. It currently involves the control of lights, heating, air conditioning, ventilation as well as security. Besides, this intelligent system is responsible for controlling household appliances such as refrigerators, washing machines, radio and TV set. Theoretically, it will allow people to monitor and control via Wi-Fi or voice command. The visual input is designed to assist voice command when there is severe interference such as walls and noisy ambient environment. It is worth mentioning that based on a authoritative market investigation report, the home automation market was worth US$ 5.77 billion in 2015, predicted to have a market value over US$10 billion by the year 2020.[18]

6. DISCUSSION

6.1. OSR Machine Learning Model

For the OSR machine learning model, the greatest challenge was to improve the prediction accuracy. Several architectures were explored, such as support vector machines, pure multilayer perceptrons, three-dimensional convolutional neural networks, ending with a three-stage architecture consisting two-dimensional VGG-16 convolutional neural networks, long short-term memory recurrent neural networks, and fully-connected multilayer perceptrons, which yielded the highest prediction accuracy.

Another challenge for the OSR model was increasing the sample size. Neural networks improve their accuracy as they train on more sample data. To address this, image augmentation was performed, generating more samples from each sample via a series of random transformations of specific magnitudes. Initially, this resulted in introducing excessive spatial noise, blurring separable boundaries between classes, decreasing the prediction accuracy of the model. To

address this, various random transformation magnitudes were tested, until an empirically determined set of magnitudes, which were not too high to cause excessive spatial noise, but were not too low to fail to create viable new data.

In the future, the OSR model can be improved in several ways. One potential improvement can be achieved by modifying the model to enable it to accept input sequences of variable lengths, which would open the model to classifying articulated words regardless of length. Another potential improvement can be achieved by increasing the complexity of the neural networks, either by increasing the number and layers of CNN convolution filters, LSTM memory units, RNN neurons, which would open the model to classifying more unique articulated words, and improve its prediction accuracy.

7. REFERENCE

[1] Yorkston, K. M., Spencer, K. A., Duffy, J. R., Beukelman, D. R., Golper, L. A., Miller, R. M., Strand, E. A., & Sullivan, M. (2001). Evidence-based medicine and practice guidelines: Application to the field of speech-language pathology. Journal of Medical Speech-Language Pathology, 9(4), 243–256.

[2] Spencer, K. A., Yorkston, K. M., & Duffy, J. R. (in press, June 2003). Behavioral management of respiratory/phonatory dysfunction from dysarthria: A flowchart for guidance in clinical decision-making. Journal of Medical Speech-Language Pathology.

[3] National Institute of Neurological Disorders and Stroke. Traumatic brain injury: hope through research. Bethesda (MD): National Institutes of Health; 2002 Feb. NIH Publication No.: 02-158.

[4] "Treatment of Dysarthria", *Medical Speech Pathology*, 2017. [Online]. Available: https://medicalspeechpathology.wordpress.com/student-handbook/treatment-of-dysarthria/. [Accessed: 24- Apr- 2017].

[5] “Convolutional Neural Networks (LeNet),” Convolutional Neural Networks (LeNet) — DeepLearning 0.1 documentation. [Online]. Available: http://deeplearning.net/tutorial/lenet.html. [Accessed: 22-Apr-2017].

[6] S. B. Damelin and W. Miller, The mathematics of signal processing. Cambridge: Cambridge University Press, 2012.

[7] A. Sachan, “Tensorflow Tutorial 2: image classifier using convolutional neural network,” CV-Tricks.com, 09-Apr-2017. [Online]. Available:

http://cv-tricks.com/tensorflow-tutorial/training-convolutional-neural-network-for-image-classification/. [Accessed: 22-Apr-2017].
[8] R Hahnloser, R. Sarpeshkar, M A Mahowald, R. J. Douglas, H.S. Seung (2000). Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. Nature. 405. pp. 947–951.
[9] K. Simonyan and A. Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition,” [1409.1556] Very Deep Convolutional Networks for Large-Scale Image Recognition, 10-Apr-2015. [Online]. Available: https://arxiv.org/abs/1409.1556. [Accessed: 22-Apr-2017].
[10] “LSTM Networks for Sentiment Analysis,” LSTM Networks for Sentiment Analysis — DeepLearning 0.1 documentation. [Online]. Available: http://deeplearning.net/tutorial/lstm.html. [Accessed: 22-Apr-2017].
[11] C. Olah, “Understanding LSTM Networks,” Understanding LSTM Networks -- colah's blog. [Online]. Available: https://colah.github.io/posts/2015-08-Understanding-LSTMs/. [Accessed: 24-Apr-2017].
[12] Sepp Hochreiter; Jürgen Schmidhuber (1997). "Long short-term memory". Neural Computation. 9 (8): 1735–1780.
[13] J. Han and C. Moraga, “The influence of the sigmoid function parameters on the speed of backpropagation learning,” Lecture Notes in Computer Science From Natural to Artificial Neural Computation, pp. 195–201, 1995.
[14] “Multilayer Perceptron,” Multilayer Perceptron — DeepLearning 0.1 documentation. [Online]. Available: http://deeplearning.net/tutorial/mlp.html. [Accessed: 22-Apr-2017].
[15] Haykin, Simon (1998). Neural Networks: A Comprehensive Foundation (2 ed.). Prentice Hall.
[16] C. M. Bishop, Pattern recognition and machine learning. New Delhi: Springer, 2013.
[17] 2017. [Online]. Available: http://yann.lecun.com/exdb/publis/pdf/boser-92a.pdf. [Accessed: 24- Apr- 2017].
[18] "Haar-like features", En.wikipedia.org, 2017. [Online]. Available: https://en.wikipedia.org/wiki/Haar-like_features. [Accessed: 24- Apr- 2017].
[19] "Home automation", En.wikipedia.org, 2017. [Online]. Available: https://en.wikipedia.org/wiki/Home_automation. [Accessed: 24- Apr- 2017].

8. APPENDIX

8.1. osr.py - main optical speech recognition program

```
from keras import backend as K
from keras.applications.vgg16 import VGG16
from keras.callbacks import Callback
from keras.constraints import maxnorm
from keras.models import Model, model_from_json
from keras.layers import Dense, Dropout, Flatten, Input
from keras.layers.pooling import GlobalAveragePooling2D
from keras.layers.recurrent import LSTM
from keras.layers.wrappers import TimeDistributed
from keras.optimizers import Nadam
from keras.preprocessing.image import random_rotation, random_shift, random_shear, random_zoom
from keras.utils import np_utils
from keras.utils.io_utils import HDF5Matrix
from pprint import pprint
K.set_image_dim_ordering("th")

import cv2
import h5py
import json
import os
import sys
import numpy as np


class OpticalSpeechRecognizer(object):
    def __init__(self, samples_generated_per_sample, frames_per_sequence, rows, columns, config_file, training_save_fn, osr_save_fn, osr_weights_save_fn):
        self.samples_generated_per_sample = samples_generated_per_sample
        self.frames_per_sequence = frames_per_sequence
        self.rows = rows
```

```
        self.columns = columns
        self.config_file = config_file
        self.training_save_fn = training_save_fn
        self.osr_save_fn = osr_save_fn
        self.osr_weights_save_fn = osr_weights_save_fn
        self.osr = None

    def save_osr_model(self):
        """ Save the OSR model to an HDF5 file
        """
        # delete save files, if they already exist
        try:
            print "\nOSR save file \"{0}\" already exists! Overwriting previous saved file.".format(self.osr_save_fn)
            os.remove(self.osr_save_fn)
        except OSError:
            pass
        try:
            print "OSR weights save file \"{0}\" already exists! Overwriting previous saved file.\n".format(self.osr_weights_save_fn)
            os.remove(self.osr_weights_save_fn)
        except OSError:
            pass

        # save OSR model
        print "\nSaving OSR model to \"{0}\"".format(self.osr_save_fn)
        with open(self.osr_save_fn, "w") as osr_save_file:
            osr_model_json = self.osr.to_json()
            osr_save_file.write(osr_model_json)

        # save OSR model weights
        print "Saving OSR model weights to \"{0}\"".format(self.osr_weights_save_fn)
        self.osr.save_weights(self.osr_weights_save_fn)
```

```
            print "Saved OSR model and weights to disk\n"

    def load_osr_model(self):
            """ Load the OSR model from an HDF5 file
            """
            print "\nLoading OSR model from \"{0}\"".format(self.osr_save_fn)
            with open(self.osr_save_fn, "r") as osr_save_file:
                    osr_model_json = osr_save_file.read()
                    self.osr = model_from_json(osr_model_json)

            print "Loading OSR model weights from \"{0}\"".format(self.osr_weights_save_fn)
            with open(self.osr_weights_save_fn, "r") as osr_weights_save_file:
                    self.osr.load_weights(self.osr_weights_save_fn)

            print "Loaded OSR model and weights from disk\n"

    def predict_words(self, sequences):
            """ Predicts the word pronounced in each sequence within the given list of sequences
            """
            with h5py.File(self.training_save_fn, "r") as training_save_file:
                    training_classes = training_save_file.attrs["training_classes"].split(",")
                    predictions = self.osr.predict(np.array(sequences)).argmax(axis=-1)
                    predictions = [training_classes[class_prediction] for class_prediction in predictions]
            return predictions

    def train_osr_model(self):
            """ Train the optical speech recognizer
            """
            print "\nTraining OSR"
            validation_ratio = 0.3
```

```
                batch_size = 32
                with h5py.File(self.training_save_fn, "r") as training_save_file:
                        sample_count = int(training_save_file.attrs["sample_count"])
                        sample_idxs = range(0, sample_count)
                        sample_idxs = np.random.permutation(sample_idxs)
                        training_sample_idxs                                    =
sample_idxs[0:int((1-validation_ratio)*sample_count)]
                        validation_sample_idxs                                  =
sample_idxs[int((1-validation_ratio)*sample_count):]
                        training_sequence_generator                             =
self.generate_training_sequences(batch_size=batch_size,

                                                training_save_file=training_save_file,

                                                training_sample_idxs=training_sample_idxs)
                        validation_sequence_generator                           =
self.generate_validation_sequences(batch_size=batch_size,

                                                        training_save_file=training_save_file,

                                                        validation_sample_idxs=validation_sample_idxs)
                        pbi = ProgressDisplay()
                        self.osr.fit_generator(generator=training_sequence_generator,

validation_data=validation_sequence_generator,

samples_per_epoch=len(training_sample_idxs),

nb_val_samples=len(validation_sample_idxs),
                                                                nb_epoch=15,
                                                                max_q_size=1,
                                                                verbose=2,
                                                                callbacks=[pbi],
```

```
                                                    class_weight=None,
                                                    nb_worker=1)

    def generate_training_sequences(self, batch_size, training_save_file,
training_sample_idxs):
        """ Generates training sequences from HDF5 file on demand
        """
        while True:
            # generate sequences for training
            training_sample_count = len(training_sample_idxs)
            batches = int(training_sample_count/batch_size)
            remainder_samples = training_sample_count%batch_size
            if remainder_samples:
                batches = batches + 1
            # generate batches of samples
            for idx in xrange(0, batches):
                if idx == batches - 1:
                    batch_idxs = training_sample_idxs[idx*batch_size:]
                else:
                    batch_idxs                                          =
training_sample_idxs[idx*batch_size:idx*batch_size+batch_size]
                batch_idxs = sorted(batch_idxs)

                X = training_save_file["X"][batch_idxs]
                Y = training_save_file["Y"][batch_idxs]

                yield (np.array(X), np.array(Y))

    def generate_validation_sequences(self, batch_size, training_save_file,
validation_sample_idxs):
        while True:
            # generate sequences for validation
            validation_sample_count = len(validation_sample_idxs)
```

```
                batches = int(validation_sample_count/batch_size)
                remainder_samples = validation_sample_count%batch_size
                if remainder_samples:
                        batches = batches + 1
                # generate batches of samples
                for idx in xrange(0, batches):
                        if idx == batches - 1:
                                batch_idxs = validation_sample_idxs[idx*batch_size:]
                        else:
                                batch_idxs                                                      =
validation_sample_idxs[idx*batch_size:idx*batch_size+batch_size]
                        batch_idxs = sorted(batch_idxs)

                        X = training_save_file["X"][batch_idxs]
                        Y = training_save_file["Y"][batch_idxs]

                        yield (np.array(X), np.array(Y))

        def print_osr_summary(self):
                """ Prints a summary representation of the OSR model
                """
                print "\n*** MODEL SUMMARY ***"
                self.osr.summary()

        def generate_osr_model(self):
                """ Builds the optical speech recognizer model
                """
                print "".join(["\nGenerating OSR model\n",
                                        "-"*40])
                with h5py.File(self.training_save_fn, "r") as training_save_file:
                        class_count = len(training_save_file.attrs["training_classes"].split(","))
                video = Input(shape=(self.frames_per_sequence,
                                                3,
```

```
                                                self.rows,
                                                self.columns))
            cnn_base = VGG16(input_shape=(3,
                                                        self.rows,
                                                        self.columns),
                                        weights="imagenet",
                                        include_top=False)
            cnn_out = GlobalAveragePooling2D()(cnn_base.output)
            cnn = Model(input=cnn_base.input, output=cnn_out)
            cnn.trainable = False
            encoded_frames = TimeDistributed(cnn)(video)
            encoded_vid = LSTM(256)(encoded_frames)
            hidden_layer = Dense(output_dim=1024, activation="relu")(encoded_vid)
            outputs = Dense(output_dim=class_count, activation="softmax")(hidden_layer)
            osr = Model([video], outputs)
            optimizer = Nadam(lr=0.002,
                                        beta_1=0.9,
                                        beta_2=0.999,
                                        epsilon=1e-08,
                                        schedule_decay=0.004)
            osr.compile(loss="categorical_crossentropy",
                                    optimizer=optimizer,
                                    metrics=["categorical_accuracy"])
            self.osr = osr
            print " * OSR MODEL GENERATED * "

    def process_training_data(self):
            """ Preprocesses training data and saves them into an HDF5 file
            """
            # load training metadata from config file
            training_metadata = {}
            training_classes = []
            with open(self.config_file) as training_config:
```

```
                training_metadata = json.load(training_config)
                training_classes = sorted(list(training_metadata.keys()))

                print "".join(["\n",
                                                        "Found {0} training
classes!\n".format(len(training_classes)),
                                                "-"*40])
                for class_label, training_class in enumerate(training_classes):
                        print "{0:<4d} {1:<10s} {2:<30s}".format(class_label, training_class,
training_metadata[training_class])
                print ""

        # count number of samples
        sample_count = 0
        sample_count_by_class = [0]*len(training_classes)
        for class_label, training_class in enumerate(training_classes):
                # get training class sequeunce paths
                training_class_data_path = training_metadata[training_class]
                training_class_sequence_paths  =  [os.path.join(training_class_data_path,
file_name)
                                                        for  file_name
in os.listdir(training_class_data_path)
                                                                if
(os.path.isfile(os.path.join(training_class_data_path, file_name))
                                                                and
".mov" in file_name)]
                # update sample count
                sample_count += len(training_class_sequence_paths)
                sample_count_by_class[class_label]                              =
len(training_class_sequence_paths)

        print "".join(["\n",
                                "Found {0} training samples!\n".format(sample_count),
```

```
                                        "-"*40])
                for class_label, training_class in enumerate(training_classes):
                        print "{0:<4d} {1:<10s} {2:<6d}".format(class_label, training_class,
sample_count_by_class[class_label])
                print ""

                # initialize HDF5 save file, but clear older duplicate first if it exists
                try:
                        print "Saved file \"{0}\" already exists! Overwriting previous saved
file.\n".format(self.training_save_fn)
                        os.remove(self.training_save_fn)
                except OSError:
                        pass

                # process and save training data into HDF5 file
                print "Generating {0} samples from {1} samples via data
augmentation\n".format(sample_count*self.samples_generated_per_sample,

                                                        sample_count)
                sample_count = sample_count*self.samples_generated_per_sample
                with h5py.File(self.training_save_fn, "w") as training_save_file:
                        training_save_file.attrs["training_classes"]                    =
np.string_(",".join(training_classes))
                        training_save_file.attrs["sample_count"] = sample_count
                        x_training_dataset = training_save_file.create_dataset("X",

                                shape=(sample_count, self.frames_per_sequence, 3, self.rows,
self.columns),

                          dtype="f")
                        y_training_dataset = training_save_file.create_dataset("Y",

                          shape=(sample_count, len(training_classes)),
```

```
                dtype="i")

            # iterate through each class data
            sample_idx = 0
            for class_label, training_class in enumerate(training_classes):
                # get training class sequeunce paths
                training_class_data_path = training_metadata[training_class]
                training_class_sequence_paths =
[os.path.join(training_class_data_path, file_name)
                                                for
file_name in os.listdir(training_class_data_path)
                                                if
(os.path.isfile(os.path.join(training_class_data_path, file_name))

and ".mov" in file_name)]
                # iterate through each sequence
                for idx, training_class_sequence_path in
enumerate(training_class_sequence_paths):
                    sys.stdout.write("Processing training data for class \"{0}\":
{1}/{2} sequences\r"
                                     .format(training_class, idx+1,
len(training_class_sequence_paths)))
                    sys.stdout.flush()

                    # accumulate samples and labels
                    samples_batch =
self.process_frames(training_class_sequence_path)
                    label = [0]*len(training_classes)
                    label[class_label] = 1
                    label = np.array(label).astype("int32")

                    for sample in samples_batch:
```

```
                                        x_training_dataset[sample_idx] = sample
                                        y_training_dataset[sample_idx] = label

                                        # update sample index
                                        sample_idx += 1

                                print "\n"

                        training_save_file.close()

                        print "Training data processed and saved to {0}".format(self.training_save_fn)

        def process_testing_data(self, testing_data_dir):
                """ Preprocesses testing data
                """
                # acquire testing data sequence paths
                testing_data_sequence_paths = [os.path.join(testing_data_dir, file_name)
                                                for file_name in os.listdir(testing_data_dir)
                                                if (os.path.isfile(os.path.join(testing_data_dir, file_name))
                                                and ".mov" in file_name)]

                # process testing data
                testing_data = []
                print "Processing {0} testing samples from {1}\n".format(len(testing_data_sequence_paths),

                        testing_data_dir)
                for testing_data_idx, testing_data_sequence_path in enumerate(testing_data_sequence_paths):
```

```
                    sys.stdout.write("Processing testing data: {0}/{1} sequences\r"
                                                                    .format(testing_data_idx+1,
len(testing_data_sequence_paths)))
                    sys.stdout.flush()
                    testing_data.append(self.process_frames(testing_data_sequence_path,
augment=False))

            return np.array(testing_data), testing_data_sequence_paths

        def process_frames(self, video_file_path, augment=True):
            """ Preprocesses sequence frames
            """
            # haar cascades for localizing oral region
            face_cascade = cv2.CascadeClassifier('haarcascade_frontalface_default.xml')
            mouth_cascade = cv2.CascadeClassifier('haarcascade_mcs_mouth.xml')

            video = cv2.VideoCapture(video_file_path)
            success, frame = video.read()

            frames = []
            success = True

            # convert to grayscale, localize oral region, equalize frame dimensions, and
accumulate valid frames
            while success:
              success, frame = video.read()
              if success:
                    # convert to grayscale
                    frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

                    # localize single facial region
                    faces_coords = face_cascade.detectMultiScale(frame, 1.3, 5)
                    if len(faces_coords) == 1:
```

```
                        face_x, face_y, face_w, face_h = faces_coords[0]
                        frame = frame[face_y:face_y + face_h, face_x:face_x + face_w]

                        # localize oral region
                        mouth_coords = mouth_cascade.detectMultiScale(frame, 1.3, 5)
                        threshold = 0
                        for (mouth_x, mouth_y, mouth_w, mouth_h) in mouth_coords:
                                if (mouth_y > threshold):
                                        threshold = mouth_y
                                        valid_mouth_coords = (mouth_x, mouth_y, mouth_w,
mouth_h)
                                else:
                                        pass
                        mouth_x, mouth_y, mouth_w, mouth_h = valid_mouth_coords
                          frame = frame[mouth_y:mouth_y + mouth_h, mouth_x:mouth_x +
mouth_w]

                        # equalize frame dimensions
                        frame = cv2.resize(frame, (self.columns, self.rows)).astype('float32')

                        # accumulate frames
                        frames.append(frame)

                       # ignore multiple facial region detections
                       else:
                                pass

                # equalize sequence lengths
                if len(frames) < self.frames_per_sequence:
                        frames = [frames[0]]*(self.frames_per_sequence - len(frames)) + frames
                frames = np.array(frames[-self.frames_per_sequence:])

                # function to normalize and add channel dimension to each frame
```

```
                proc_frame = lambda frame: np.array([frame / 255.0]*3)

                if augment:
                        samples_batch = [np.array(map(proc_frame, frames))]
                        # random transformations for data augmentation
                        for _ in xrange(0, self.samples_generated_per_sample-1):
                                rotated_frames = random_rotation(frames, rg=4.5)
                                shifted_frames = random_shift(rotated_frames, wrg=0.05,
hrg=0.05)
                                sheared_frames = random_shear(shifted_frames, intensity=0.08)
                                zoomed_frames = random_zoom(sheared_frames,
zoom_range=(1.05, 1.05))
                                samples_batch.append(np.array(map(proc_frame,
zoomed_frames)))
                        return_data = samples_batch

                else:
                        return_data = np.array(map(proc_frame, frames))

                return return_data

class ProgressDisplay(Callback):
        """ Progress display callback
        """
        def on_batch_end(self, epoch, logs={}):
                print "        Batch {0:<4d} => Accuracy: {1:>8.4f} | Loss: {2:>8.4f} | Size:
{3:>4d}".format(int(logs["batch"])+1,


float(logs["categorical_accuracy"]),

                                                                float(logs["loss"]),
```

```
                                                        int(logs["size"]))

if __name__ == "__main__":
        osr = OpticalSpeechRecognizer(samples_generated_per_sample=10,
                                                        frames_per_sequence=30,
                                                        rows=100,
                                                        columns=150,
                                                        config_file="training_config.json",

training_save_fn="training_data.h5",
                                                        osr_save_fn="osr_model.json",

osr_weights_save_fn="osr_weights.h5")

        # Training workflow example
        osr.process_training_data()
        osr.generate_osr_model()
        osr.print_osr_summary()
        osr.train_osr_model()
        osr.save_osr_model()

        # Application workflow example. Requires a trained model. Do not use training data for
actual tests
        osr.load_osr_model()
        osr.print_osr_summary()
        test_sequences, test_sequence_file_names = osr.process_testing_data("./testing-data")
        test_predictions = osr.predict_words(test_sequences)
        print "".join(["Predictions for each test sequence\n",
                                "----------------------------------"])
        for file_name, prediction in zip(test_sequence_file_names, test_predictions):
                print "{0}: {1}".format(file_name, prediction)
```

8.2. training_config.json - training configuration file

```
{
        "yes": "./training-data/yes",
        "no": "./training-data/no",
        "null": "./training-data/null"
}
```

8.3. haarcascade_frontalface_default.xml - haar cascade for face detection; excluded due to extreme length; alternative can be found here: http://alereimondo.no-ip.org/OpenCV/34

8.4. haarcascade_mcs_mouth.xml - haar cascade for mouth detection; excluded due to extreme length; alternative can be found here: http://alereimondo.no-ip.org/OpenCV/34

8.5. Batch_Relay.py

```
class BatchQueue:
    def __init__(self):
        self.length = 0
        self.head = None
        self.last = None
    def is_empty(self):
        return self.is_empty.length == 0
    def insert(self, img):
        last = self.last
        last.next = img
        self.last = img
        self.length += 1
    def remove(self):
        self.head = self.head.next
        self.length -= 1

batch = BatchQueue()
```