딥러닝을 활용한 지문자 인식 및 음성화 (TTS)

1118 이현수, 1314 이건희
지도 교사 : 송석리

(초록) 초록은 15줄을 넘지 않도록 한다. 논문은 12장 이내로 작성을 권장한다. 국문 서체는 바탕체, 영문 서체는 Times New Roman, 글자크기 10 point로 작성한다.

1. 서론

1.1. 연구 동기

예전부터 장애인들의 언어인 '수어'에 관심을 많이 가지고 있었는데, 연구원들은 이를 배울 기회를 가지게 되어 수어를 배운 적이 있다. 그 중에서도 지문자는 다른 수어들보다 보다 간단하였고, 한글의 모든 자음 (쌍자음 포함)과 모음을 표현할 수 있기 때문에 매우 인상적이었다.

지문자란, 수화보다 비교적 간단하고 단순한, 손을 이용한 문자로, 한글 자음과 모음을 표현할 수 있다. 아래는 그림.1. 은 지문자로 한글을 표현하는 방법이다. 수화로 표현할 수 없는 말에 사용되며, 주로 고유명사나 외래어를 표현할 때 사용된다.

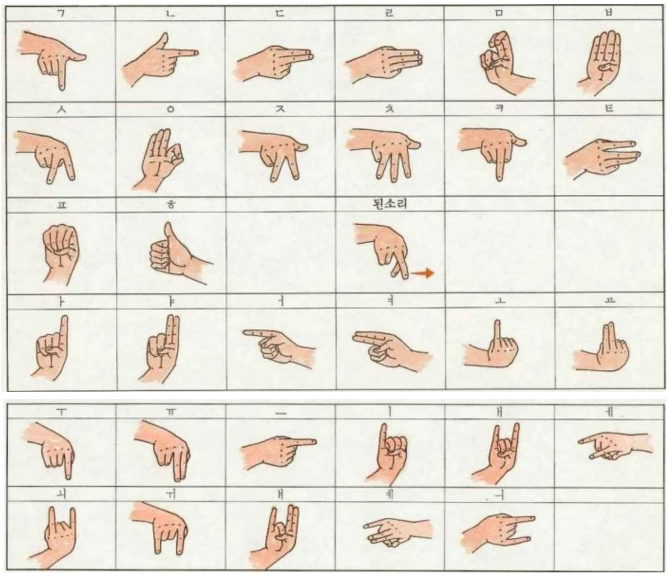

그림. 1. 한글 지문자 자음, 모음 https://goo.gl/413ZuH

대부분의 농인 (청각 장애인) 이 지문자를 애용한다고 한다. 그 이유는 청각 장애인들의 경우 일반 문자보다, 수어(지문자) 가 훨씬 익숙하게 느끼기 때문이다.

따라서, 청각 장애인이 수어 (지문자) 로 표현하는 말을 비장애인들이 쓰는 말로 번역해 농인들의 의사소통을 보다 원활하게 하고 싶었다. 마침 딥러닝을 접할 기회가 생겨 집중적으로 공부하게 되었고, 이 과정에서 CNN 알고리즘을 학습하며 이미지 인식을 조금 더 간단하게 할 수 있다는 것을 알게 되었다. 결론적으로 CNN을 이용하면 지문자 인식을 실현 가능하다는 것을 깨닫게 되었다. 기존 선행 논문들의 경우. 한글 지문자 인식은 거의 구현되어 있지 않았기에, 이런 동기까지 더해져, 연구를 시작하게 되었다.

1.2. 연구의 필요성

이 연구를 통해서, 청각 장애인들의 의사소통을 조금 더 원활하게 만들 수 있다. 연구를 통해 개발한 프로그램은 농인들이 지문자로 자신이 하고 싶은 말을 표현하면, 이를 자동으로 인식하여서 문장으로 변환하고, 비장애인들이 사용하는 언어(음성) 으로 자동 변환해주는 과정을 모두 포함하고 있다. 결론적으로, 장애인과 비장애인과의 거리감 역시 줄어들 수 있다.

2. 이론적 배경

2.1. 딥러닝 기본이론

가. Deeplearning

딥러닝이란 주어진 데이터들을 스스로 학습하고, 그 학습 내용을 바탕으로 컴퓨터 스스로 생각 및

사고 (판단)을 하여서 알맞은 결론을 만들어 내는 것이다. 대표적으로 딥러닝을 구현한 예로는 구글의 '알파고' 가 있으며, 바둑 기수를 지속적으로 학습하여서 자신이 이기기 위해서 바둑을 어디에 두어야 할지 판단한다.

즉, 딥러닝은 머신러닝을 실현하기 위한 하나의 방법이다. 딥러닝을 활용한 연구들로는 사진 인식, 음성 인식, 글자 인식, 자동 번역 및 단어완성 등이 있다.

나. 신경망 설계

딥러닝 알고리즘의 구조는 기본적으로 '인공 신경망 (NeuralNetwork)'로 이루어져 있다. 신경망은 입력 노드 (계층)로부터 데이터들이 들어온 후, 히든 노드 (계층)에서 학습을 진행하고 판단을 세운 후, 출력 노드 (계층)에서 최종적인 값을 출력한다.

신경망 내에서 딥러닝이 이루어지는 과정은 다음과 같다. 최종적으로 학습을 통해 신경망이 만들어낸 추축값과 정답의 일치율을 높여야 하므로, 그 차이 (오차)를 최소화 시키는 과정을 따라 딥러닝이 진행된다. 오차 최소화를 위해 사용하는 방법으로는 '최소 제곱법' 이 가장 널리 쓰이는데, 이는 주로 비교적 적은 양의, 단순한 데이터들을 이용해서 그 관계를 찾을 때 사용된다. 예시로 주어져 있는 아래 데이터에서 x와 y는 선형 관계를 가지고 있다고 가정하고, 그 관계식을 $y = ax + b$라고 해 보자.

표1. 선형 관계를 가지는 데이터 예시

x	y
1	2
2	5
3	7
4	9
5	13
6	15
7	17
8	20

먼저 아래와 같은 수식을 통해서 기울기와 절편 a, b의 값을 정할 수 있다.

$$a = \frac{\sum_{k=1}^{n}(x_k - x_{평균})(y_k - y_{평균})}{\sum_{k=1}^{n}(x_k - x_{평균})} \cdots (1)$$

$$b = y_{평균} - a \times x_{평균} \cdots (2)$$

이때 기울기 a의 값을 '가중치(W)' 라고 하고, 절편의 값 b를 '편향' 이라고 한다. 가중치와 편향이 변수별로 다르다면 가중치 행렬과 편향 행렬을 생각할 수 있는데, 이때의 선형 관계식은 아래와 같이 나타낼 수 있다.

$$y = x \cdot W + b \cdots (3)$$

즉, 행렬곱으로 표현할 수 있다.

다음으로는 오차를 구해서 그 오차를 최소화 시키는 과정에 대한 설명이다. 데이터 양이 많거나 그 관계가 복잡할 경우 (3) 만으로는 가중치와 편향을 결정할 수 없다. 따라서 오차를 계산해야 하는데, 오차로는 '평균 제곱근 오차(RMSE)'를 가장 많이 사용한다.

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(X_i - Y_i)^2} \cdots (4)$$

이때 X_i는 예측 값을, Y_i는 실제 값 (정답)을 의미한다.

오차를 줄여가는 방법으로는 '경사 하강법' 을 이용한다.

다. 경사 하강법

경사 하강법이란, 매우 조금씩 그 값 (W, b)을 바꿔 가면서 오차가 최소가 되는 최적의 점을 찾는 방법이다. 그래프에서 보자면, x좌표를 조금씩 바꿔 가면서 최솟값을 가지는 점을 찾는 과정이다. 해당 점에서 미분 계수를 구해서, 그 기울기 부호의 반대 방향으로 이동한다. 예를 들어, 해당 점에서의 미분계수가 양수라면, 음의 방향으로 ($-x$ 방향) 조금 더 이동하고, 이동한 점에서 다시 미분을 해서 이 과정을 반복하는 것이다.

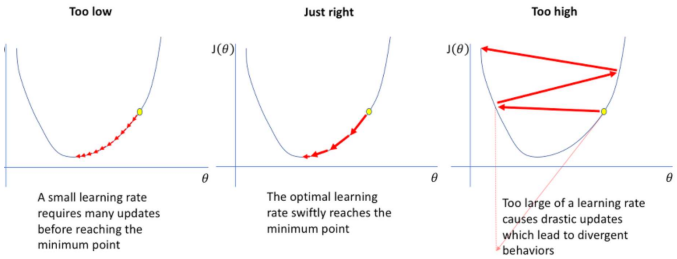

그림. 2. 경사 하강법의 원리
출처 머신러닝단기집중과정

이때 그래프 상에서 '얼마만큼' 움직일지를 정하는 변수가 바로 '학습률(learning rate)' 이다. 학습률이 크다는 것은 이동 폭이 크다는 것이고, 학습률이 작다는 것은 그 반대이다. 적절한 값의 학습률을 설정해야 오차를 최소화 할 수 있으며, 너무 크거나 작은 경우 최솟값에 도달하지 못하거나 시간이 너무 오래 걸릴 수 있다.

2.2. CNN

CNN이란, Convolution Neural Network의 약자로, 이미지 인식 등을 보다 효율적으로 도와주는 딥러닝의 한 방법 중 하나이다. 기존 신경망 알고리즘을 이용하면 사진 데이터 (3차원)을 1차원 평면배열로 변환시키는 과정에서 데이터 손실이 일어나게 되는데, 이를 보완해서 그 정확도를 높이는 것이 바로 CNN의 핵심 원리이다. CNN 알고리즘의 크게 컨볼루션 계층, 풀링 계층, 그리고 판단 및 출력계층으로 구성된다.

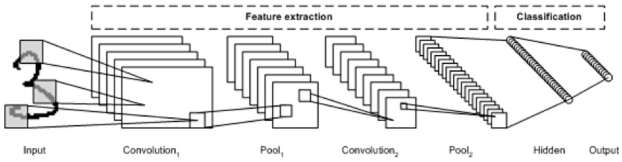

그림. 3. CNN의 원리
출처 https://goo.gl/fkphjL

먼저 하나의 이미지가 입력되면, '컨볼루션 계층'에서 이 이미지의 특징적 정보들을 추출해 내어서 압축시킨다. 이후 한번 더 압축 과정을 거치는데, 이것이 바로 '풀링' 이다. 주로 '맥스 풀링' 기법을 이용해서 풀링을 진행하며, 이 연구 역시 '맥스 풀링' 기법을 사용하였다. 이 과정을 설정한 노드 개수만큼 반복하여서 최종적으로 판단 (결론) 을 내린다.

이 연구에서는 딥러닝(CNN)을 통해 위 지문자들을 인식하는 알고리즘을 개발하였다.

2.3. Dropout

적은 데이터로 학습을 시도하면 그 정확도가 매우 낮다. 데이터 양이 부족하기 때문에 신경망이 충분한 학습을 진행하지 못하였기 때문이다.

하지만, 그렇다고 해서 너무 많은 데이터가 있어도 '과적합 (Overshooting)' 이 발생하여서 정확도가 낮아진다. 학습 횟수 역시 마찬가지다. 너무 적은 학습은 '학습량 부족' 이라는 결론을 야기하지만, 너무 많은 학습 역시 과적합을 발생시킨다. 즉, 적절한 양의 데이터와 학습 횟수를 설정해야 최적의 정답률을 얻을 수 있다.

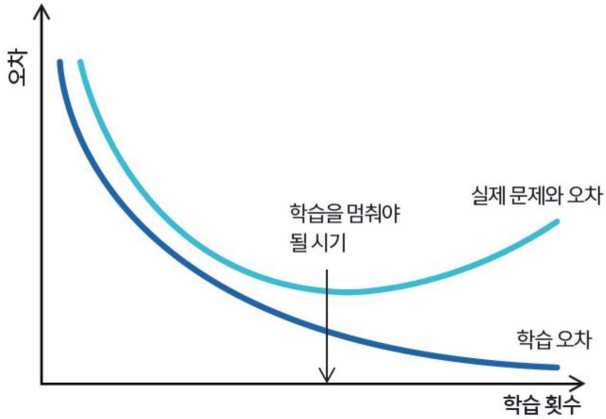

그림. 4. 학습 횟수가 너무 많아지면 오히려 오차가 증가하는 과적합 현상
출처 http://dongascience.donga.com/news.php?idx=11225

하지만 이 '적절한' 횟수를 알아내는 것은 힘들다. 따라서 '드롭아웃' 기법이 사용되는데, 이 드롭아웃 기법은 통해서 데이터나 학습 횟수가 과적합을 불러올 정도로 많아도 자동적으로 줄여준다. 이 연구에서도 드롭아웃 기법을 이용해서 과적합을 방지하였다.

2.4. Activation Function (활성화 함수)

신경망에서 입력 노드에서 입력된 데이터들은 특정한 함수를 거쳐 처리된 후 출력된다. 이때 이 처리를 담당하는 '특정한 함수' 를 활성화 함수라고 한다. 주로 비선형 함수로 분류되는데, 그 이유는 신경망에 입력되는 노드는 비선형이며 아날로그 식이기 때문이다. 이 과정을 간단히 나타내면 아래와 같다.

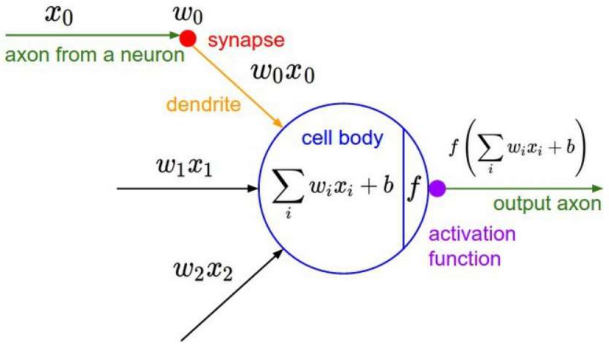

그림. . 활성화 함수의 작용과정
위 그림에서 activation function 이 활성화 함수가 작용하는 부위이다. 출처 https://goo.gl/9MiP4R

활성화 함수는 여러 가지 종류가 있는데, 대표적인 종류는 아래와 같다.

① 계단 함수

$$f(x) = \begin{cases} 0 & (x < 0) \\ 1 & (x \geq 0) \end{cases} \cdots (5)$$

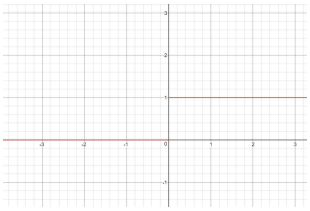

그림. . 계단 함수

② 시그모이드 함수

$$f(x) = \frac{1}{1+e^{-x}} \cdots (6)$$

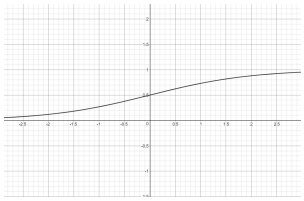

그림. . 시그모이드 함수

③ Relu 함수 (임계논리 함수)

$$f(x) = \max(0,x) = \begin{cases} x & (x > 0) \\ 0 & (x \leq 0) \end{cases} \cdots (7)$$

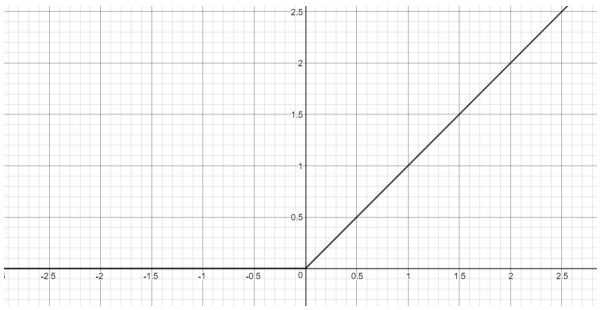

그림. . 임계논리 함수

이 연구에서는 활성화 함수로 Relu 함수를 사용하였다.

3. 연구의 목적

이 연구는 최종적으로 청각 장애인들의 지문자를 인식해서 문장으로 출력해 주고, 이를 음성으로 읽어 주어서 (Text To Speech, TTS) 비장애인들과의 소통을 원활하게 하려는 목표를 가지고 있다. 딥러닝과 CNN, Tensorflow를 이용해서 이를 구현하였으며, 결론적으로 장애인, 비장애인 간의 언어 장벽을 없앨 수 있을 것이다.

4. 연구 방법

본 연구는 2018년 10월 22일부터 2018년 11월 5일까지 진행되었다.

4.1. 개발 환경

이 연구에서 최종적으로 목표하는 것은 사진 인식과 TTS 기능이다. CNN 알고리즘이 가장 적합하다 판단하였기에, Python에서 Tensorflow를 이용하여서 개발하기로 하였다. 아래는 정확한 개발 환경이다.

Windows - 64bit
Anaconda, Tensorflow (텐서플로) 기반
Python 3.7
Pycharm, Jupyter Notebook 동시 사용

아래는 본 연구 시 설치하거나 불러온 library 목록이다.

```
import tensorflow
import os
import cv2
import numpy

from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
from numpy import array
from gtts import gTTS
```

전체 코드는 논문 맨 뒤 <부록>에 첨부하였으며, 자세한 결과물들은 GitHub에 업로드 하였다. GitHub Url 역시 <부록> 에 첨부되어 있다.

4.2. 연구 절차 및 개발

우리는 최종적으로 원하는 결과물을 얻기 위해 다음 절차에 따라서 연구(개발)을 진행하였고 검토 역시 병행하였다.

① 데이터 생성하기

: 한글 지문자 데이터의 경우, 빅 데이터를 얻을 수 있는 사이트 (Kaggle) 에서도 찾을 수 없었다. 따라서 직접 촬영하기로 하였으며, 검정색 바탕을 배경으로 모든 지문자에 대해서 촬영 하였다.

또한, 사람마다 손 모양이나 색깔 등이 약간씩 다르므로 연구자 2명 외에도 주변인의 손 사진 역시 찍어서 학습 데이터에 추가함으로써 그 정확도를 높였다. 데이터 생성은 수시로 이루어졌다. (최종 개발이 끝난 후에도 후속 과제로 정확도 높이기를 진행하였는데, 이 기간 역시 데이터를 추가하였다).

② 이미지 읽어오기 및 처리

: 기존 CNN 알고리즘에서 손글씨를 인식할 때는, Tensorflow 에서 기본적으로 MNIST 데이터를 제공하였기 때문에 이미지를 변환할 필요가 없었다. 하지만 현재 인위적으로 데이터를 생성하였기 때문에 (직접 촬영), Python에서 이를 인식하기 위해서는 28pixel × 28pixel 사이즈로 바꾸어야 숫자 배열로 변환해야 한다.

㉠ 촬영한 사진을 28pixel × 28pixel 로 변환한다(Resize). 이때는 사진 편집 프로그램 (PhotoScape) 을 이용하여서 일괄적으로 변환하였다.

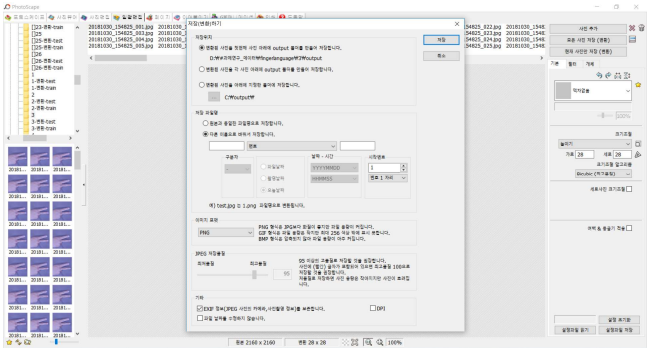

그림. . 이미지를 28 × 28 사이즈로 변환하는 과정

그 결과는 아래와 같다. 각 폴더 내에 들어있는 이미지들의 예시이다.

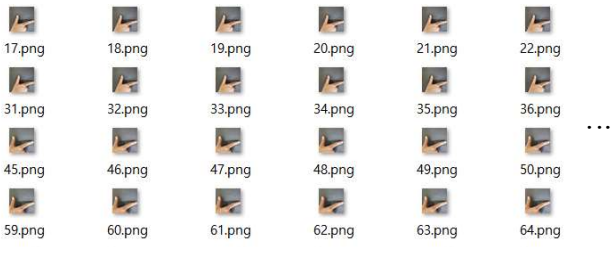

그림. . 28 × 28 사이즈로 변환된 이미지들 (ㄴ)

㉡ Resize 시킨 이미지를 숫자 배열로 변환하고, train data (학습 데이터) 와 test data (테스트 데이터)로 구분한 뒤, 이를 CNN 인식 데이터에 첨부시킨다.

㉢ 첨부된 사진들을 정답과 함께 배열에 최종적으로 저장하고, 인식을 시작한다.

위 ㉡, ㉢을 위한 코드는 아래와 같다. 자세한 코드 해설(주석)은 GitHub 에 업로드 하였다.

코드 1

```
#cnn이미지 변환은 김성훈 교수님의 코드를 참고함 (https://goo.gl/CjHapo)
#학습 데이터 가공
TRAIN_DIR = 'D:/deeplearning/programming/fingerlanguage/traindata'
train_folder_list = array(os.listdir(TRAIN_DIR))

train_input = []
train_label = []

output_nodes = 31
label_encoder = LabelEncoder()
#폴더들을 인식, 그것들의 이름을 순차적으로 숫자형 데이터로 전환하기
integer_encoded = label_encoder.fit_transform(train_folder_list)

onehot_encoder = OneHotEncoder(sparse=False)
integer_encoded = integer_encoded.reshape(len(integer_encoded), 1)

onehot_encoded = onehot_encoder.fit_transform(integer_encoded)

#img를 변환시키는 과정
for index in range(len(train_folder_list)):
    path = os.path.join(TRAIN_DIR, train_folder_list[index])
    path = path + '/'
    img_list = os.listdir(path)
    for img in img_list:
        img_path = os.path.join(path, img)
        img = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)  #이미지를 읽어온다.


#이미지를 numpy형 배열으로 변환
        train_input.append([np.array(img)])
        train_label.append([np.array(onehot_encoded[index])])

#input 데이터를 28x28 크기의 있는 배열으로 변환
train_input = np.reshape(train_input, (-1, 784))
train_label = np.reshape(train_label, (-1, output_nodes))

train_input = np.array(train_input).astype(np.float32)
train_label = np.array(train_label).astype(np.float32)
```

이미지를 배열로 변환시킨 결과는 다음과 같다.

```
[[72 74 70 59 61 64 78 164 190 202 180 182 197 186 188 193 194 188 185 177 178 151  56  46  51  54  56  56]
 [72 74 66 61 61 59 93 175 196 195 181 191 198 184 188 195 195 189 182 175 176 166  80  46  53  50  53  56]
 [74 74 66 66 65 59 111 184 199 190 187 199 197 185 190 196 194 190 180 175 175 182 117  43  56  53  53  56]
 [74 74 70 66 66 66 133 189 196 192 194 203 196 190 195 199 195 190 180 175 175 186 149  53  53  53  53  53]
 [74 74 72 66 64 68 158 198 192 194 199 205 197 197 199 201 195 188 183 174 173 182 173  82  46  55  56  56]
 [76 78 68 66 59 78 180 199 187 193 200 208 199 200 202 204 197 189 185 174 174 176 186 127  50  56  59  56]
 [78 78 72 68 59 114 194 192 183 190 200 210 199 204 209 206 200 194 188 179 178 178 192 170  66  53  59  59]
 [80 76 80 70 61 151 203 184 179 190 200 210 195 207 216 205 203 197 190 182 183 181 192 195  85  54  59  61]
 [78 78 82 78 68 158 206 177 170 202 203 206 195 209 212 210 200 198 194 194 189 197 203 206 127  50  61  64]
 [74 76 76 78 68 140 200 160 186 203 206 189 193 214 207 209 198 194 202 194 183 194 206 204 175  68  56  66]
 [76 78 80 80 68 111 189 190 210 206 196 186 199 210 213 207 203 199 194 210 124  84 195 199 197 109  56  61]
 [76 84 80 78 80 80 102 126 212 217 201 179 203 221 217 205 213 197 193 192  59  21 139 185 192 156  70  61]
 [78 76 85 89 84 82 74 66 98 144 162 189 226 223 218 212 209 205 213 146  39  21  53 173 192 180  82  66]
 [76 85 84 84 80 80 78 76 72 46 106 203 236 232 223 206 214 216 200  93  53  43  28 116 200 196  98  61]
 [78 84 82 87 80 80 82 78 78 61 102 195 224 186 184 198 210 214 161  66  68  68  53  66 187 224 151  66]
 [78 82 82 80 80 80 78 76 74 59 85 84 46 106 192 208 212 106  64  78  78  74  66 103 204 192  76]
 [80 80 80 80 80 82 80 80 76 70 56 34 28 50 126 187 220 157  70  84  78  84  80  78  72  89  92  80]
 [78 78 78 78 78 78 78 76 74 76 56 39 28 59 145 195 204 120  70  84  84  87  85  84  84  78  74  76]]
                                    ⋮
                                (이후 생략)
```

그림. . 28 × 28 사이즈의 배열로 변환된 이미지

㉣ 테스트 데이터에 대해서도 위와 같은 과정을 수행한다. 방법은 동일하므로 테스트 데이터에 대해 실행하는 과정은 생략하였다.

③ 신경망 구성하기

기본적으로 CNN을 기반으로 알고리즘을 설계하였으며, 노드의 개수는 총 4개로 구성하였으며, 입력 노드 1개와 히든노드 2개, 그리고 출력 노드 1개로 구성하였다. (각각 L1, L2, L3, model) 이 과정에서 활성화 함수로는 Relu 함수를 이용하였다.

오차 함수로는 교차 엔트로피 함수를 사용하였고, 이는 수식으로 아래와 같다.

$$e = -\sum_{k} t_k \log o_k \quad \cdots(8)$$

이때 o_k는 신경망의 출력값, t_k는 정답 값이다.

학습률의 경우 여러 값을 설정해 본 결과, 0.001이 오차를 최소화 시키는 최적의 값이라는 것을 알 수 있었다. 이들을 코드로 아래와 같이 작성하였다.

코드 2

```
learning_rate = 0.001
#오차 계산 (교차엔트로피 함수가 제공됨)
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(logits=model, labels=Y))

#평균 엔트로피 함수
optimizer = tf.train.AdamOptimizer(learning_rate).minimize(cost)
```

④ 학습 시키기

데이터 개수와 양이 많이 때문에, 미니 배치(batch)를 사용하였으며 총 배치의 개수는 100개로 하였다. 학습 반복 횟수는 50번, 55번... 80번까지 시도해 본 결과 55번이 가장 적당하다는 결론을 내렸다. 55번을 넘어가면 과적합이 발생해서 오히려 정확도가 떨어졌기 때문이다.

아래는 학습을 실행하는 코드이다.

코드 3

```
sess.run(init)  #초기화 실행시키기
batch_size = 100  #미니배치 사용 (100개)
total_batch = int(len(train_input) / batch_size)
all_epoch = 55
for epoch in range(all_epoch):
    total_cost = 0
    for i in range(total_batch):
        start = ((i+1) * batch_size) - batch_size
        end = ((i+1) * batch_size)
        #학습할 데이터를 배치 크기만큼 가져온 뒤,
        batch_xs = train_input[start:end]
        batch_ys = train_label[start:end]
        #입력값 = batch_xs, 출력값 = batch_ys

        _, cost_val = sess.run([optimizer, cost], feed_dict={X: batch_xs, Y: batch_ys})
        total_cost = total_cost + cost_val #오차값 구하기
    print('Epoch:', '%04d' % (epoch + 1), 'Avg. cost = ', '{:3f}'.format(total_cost / total_batch))
```

학습 결과 테스트 데이터로 확인을 하고, 그 정확도를 출력해 준다. 정확도를 출력하는 코드는 '부록' 전체코드에 포함되어 있다.

⑤ 인식한 결과 한글로 출력하기

지문자 데이터들 (Test Data)을 인식하였다면, 그 인식한 결과는 현재 숫자로 되어 있으므로 한글로 변환하는 과정이 필요하였다. result라는 변수에 숫자로 만들어진 인식결과를 저장한 후, 이를 dictionary를 이용해서 한글 자/모음으로 변환하였다.

코드 4

```
#결과확인
result = sess.run(model, feed_dict = {X: test_input, Y: test_label, keep_prob : 1})
#얻은 결론값에 해당하는 한글 초성을 출력한다.
char = {0 : 'ㄱ', 1 : 'ㄴ', 2 : 'ㄷ', 3 : 'ㄹ', 4 : 'ㅁ', 5 : 'ㅂ', 6 : 'ㅅ', 7 : 'ㅇ', 8 : 'ㅈ', 9 : 'ㅊ',
10 : 'ㅋ', 11 : 'ㅌ', 12 : 'ㅍ', 13 : 'ㅎ', 14 : 'ㅏ', 15 : 'ㅑ', 16 : 'ㅓ', 17 : 'ㅕ', 18 : 'ㅗ', 19 : 'ㅛ',
20 : 'ㅜ', 21 : 'ㅠ', 22 : 'ㅡ', 23 : 'ㅣ', 24 : 'ㅐ', 25 : 'ㅒ', 26 : 'ㅔ', 27 : 'ㅖ', 28 : 'ㅚ', 29 : 'ㅟ',
30 : 'ㅢ'}
#결과값의 index가 가장 높은 값을 가지는 특성을 이용하였다.
for i in range(len(result)):
    maxi = -100
    ind = 0
    for j in range(0, output_nodes):
        if result[i][j] > maxi:
            maxi = result[i][j]
            ind = j
    print(char[ind])
```

그 원리는 다음과 같다. char이라는 dictionary에 'ㄱ' 부터 'ㅟ' 까지 각각의 문자에 대해서 번호를 부여하였다. 인식 결과, 배열에서 가장 높은 값을 가지는 index가 인식 결과이므로 그 index를 key로 가지는 value를 알아내는 것이다.

⑥ 인식 정확도 높이기

여러 가지 정확도를 더 높이기 위한 방법들을 생각해 보았는데, 크게 2개를 생각하였다. 바로 이미지를 회전시키는 방법과 드롭아웃이다.

그 중 드롭아웃 기법을 사용해 보았는데, 다음 계층으로 넘어갈 때 유지할 데이터 비율을 57% 로 설정하였다. 변수에 할당할 때는, keep_prob = 0.57 로 설정하였다. 그 코드는 아래와 같다.

코드 5

```
#드롭아웃 시 사용할 변수 keep_prob
keep_prob = tf.placeholder(tf.float32)
L3 = tf.nn.dropout(L3, keep_prob)

_, cost_val = sess.run([optimizer, cost], feed_dict={X: batch_xs, Y: batch_ys, keep_prob : 0.57})
result = sess.run(model, feed_dict = {X: test_input, Y: test_label, keep_prob : 1})
```

또한, 연구 진행 후에도 수시로 손 모양의 사진을 추가 함으로써 추가 학습 / 테스트 데이터들를 생성하였고, 그 결과 기존에는 잘 인식이 되지 않던 사진들도 잘 인식되었다.

⑦ TTS를 위한 과정 - **임의 데이터 인식하기**

이 연구에서 최종적으로 목표하는 바는 임의의 지문자 사진을 이용해 하나의 글자를 합성하고, 그 글자들로 이루어진 문장을 음성으로 읽어 주는 것이다. 따라서 먼저 임의의 지문자 데이터들을 인식시키는 알고리즘부터 설계하였다.

ttss라는 폴더에 인식을 원하는 순서대로 아래 그림과 같이 데이터들을 추가하였다.

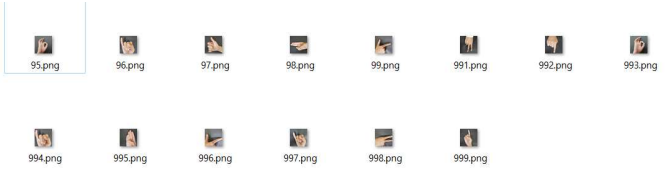

그림. . TTS인식을 위한 이미지 데이터들
해당 이미지들은 '이현수입니다' 라는 문장을 나타낸다.

그리고 이 사진들을 숫자 배열로 가공 하였는데, 이는 위의 학습 데이터들을 변환시키는 ***코드 1*** 과 일치한다.

하지만, 이미지를 변환시킨 후 하나씩 인식시킨 결과 문제가 발생하였는데, 이미지 인식 결과 그 결과들의 index값이 일정하게 밀려서 출력 되었다. 처음에는 인식 오류하고 생각하였으나, 학습 데이터들을 그대로 인식시켜도 다른 문자로 나오는 것을 보아서 인식 오류는 아니라고 생각하였고, index 처리의 오류라고 최종 판단을 내렸다. 이를 위해서 index를 인위적으로 변환하는 작업을 수행하였다.

코드 6

```
han_result = []
#결과출력
ttsresult = sess.run(model, feed_dict = {X: tts_in
put, keep_prob : 1})
for i in range(len(ttsresult)):
    maxi = -1000
    ind = 0
    for j in range(0, len(ttsresult[i])):
        if ttsresult[i][j] > maxi:
            maxi = ttsresult[i][j]
            ind = j
#인위적 변환 예시
    if 25 <= ind and ind <= 30 :
        ind = ind - 22
        print(char[ind])
        han_result.append(char[ind])
        continue
                    ⋮
print(han_result)
```

⑧ TTS를 위한 과정
- 한글 초성, 중성, 종성 합쳐서 한 글자로 만들기

임의의 지문자 사진 데이터들을 인식 하였다면, 그 인식 결과는 초성으로 ('ㄱ', 'ㅏ', 'ㄴ' 처럼) 출력되고 저장된다. 따라서 이들을 하나의 글자로 만들어 주어야 한다.

한글 글자를 합성할 때는, 글자의 시작과 끝을 알아내는 것 역시 필요하다. 예를 들어 인식 결과가 'ㅈㅓㅇㅂㅗ (정보)' 일 때, 'ㅇ' 에서 글자가 바뀐다는 것을 인식해야 한다. 따라서, 글자가 바뀌는 경우를 아래와 같이 2개로 나누어 보았다.

㉠ 모음-자음-자음 순서일 때 :
중성 - 종성(받침) - 초성 순서이다.
㉡ 모음-자음-모음 순서일 때:
중성 - (받침없이) 초성 - 중성 순서이다.

전체 한글 인식 코드는 분량이 매우 길기 때문에 부록에서 첨부하였으며, 아래 ***코드 7*** 에는 글자가 바뀌는 경우에 대한 처리 과정만 첨부하였다.

아래 코드에서 han_result는 인식한 결과를 초성으로 담고 있는 배열, Z는 자음 데이터, Vowel은 모음 데이터를 담고 있는 배열이다. cntind는 반복문에서 index를 나타내는 값이다.

코드 7

```
    # 1글자인 특수한 경우는 따로 처리
    elif len(han_result) == 2:
        a = han_result[0]
        b = han_result[1]
        c = ''
        flag = True #특수한 경우라는 표시
    elif len(han_result) == 3:
        a = han_result[0]
        b = han_result[1]
        c = han_result[2]
        flag = True #특수한 경우라는 표시
# 1. 모음-자음-자음 순서일 때 : 자음에서 바꾸기
    elif cntind >= 1 and han_result[cntind - 1]
in Vowel and han_result[cntind] in Z and
han_result[cntind + 1] in Z:
        a = han_result[cntind - 2]
        b = han_result[cntind - 1]
        c = han_result[cntind]
# 2. 모음-자음-모음순일때 : 모음에서 바꾸기
    elif cntind >= 1 and han_result[cntind - 1]
in Vowel and han_result[cntind] in Z and
han_result[cntind + 1] in Vowel:
        a = han_result[cntind - 2]
        b = han_result[cntind - 1]
        c = ''
```

⑨ TTS를 위한 과정 - **음성파일로 변환시키기**

코드 8

```
# 하나의 단어로 합치기
total_char = '' #문장을 담을 문자열 변수
for i in range(len(char_result)):
    total_char = total_char + char_result[i]

# 최종적으로 TTS변환
tts_file = gTTS(text=total_char, lang='ko')
tts_file.save("FinalTTSFile.mp3")
print('저장 완료!')
```

이로써 최종적으로 지문자 인식부터 TTS 변환까지 코드를 완성하였다. 이 실행 결과는 <5. 연구 결과>에 첨부하였다.

⑩ [향후과제] 영상 인식하기

이 연구를 조금 더 발전 시킨다면, 영상에서 이미지를 추출해 내서, 그 이미지들을 인식시키는 것까지 구현할 수 있을 것이다. (관련 내용은 <6.2 후속연구 제안>에도 나와 있다). 그 뼈대를 잡아 보았다.

먼저 카메라를 켜서, 영상을 촬영하는 코드이다.

코드 9

```
#참고한 코드 https://goo.gl/AYhGJm
import cv2 as cv
cam = cv.VideoCapture(0)

#저장하기
r = cv.VideoWriter_fourcc(*'XVID')
out = cv.VideoWriter('video.avi', r, 30.0, (640, 480))
while(cam.isOpened()):
    ret, frame = cam.read()
    if ret == False:
        continue
    out.write(frame)
    cv.imshow('frame', frame)
    ##종료하려면 ESC키
    if cv.waitKey(1) & 0xFF == 27:
        break
cam.relase()
out.release()
cv.destroyAllWindows()
```

촬영한 영상은 video.avi 라는 파일로 저장이 된다. 이 영상을 읽어와서, 이미지를 추출 (캡처) 해서 저장하는 코드는 아래와 같다.

코드 10

```
#참고한 코드 https://goo.gl/AYhGJm , https://goo.gl/NNBStM
import cv2
from PIL import Image
cam = cv2.VideoCapture('video.avi')
cntvid = 0
while(cam.isOpened()):
    ret, image = cam.read()
    image_rotate = cv2.transpose(image)
    image_rotate = cv2.flip(image_rotate, 1)
    fps = cam.get(cv2.CAP_PROP_FPS)
    if (int(cam.get(1)) % 5 == 0):
        cv2.imwrite("frames/frame%d.png" % cntvid, image_rotate)
        cntvid += 1
    if cntvid == 12 : break
cam.release()
cv2.destroyAllWindows()

#이미지 변환 (resize, 28 x 28으로)
for i in range(cntvid):
    path = 'frames/frame' + str(i) + '.png'
    save_path = 're_frames/re_' + str(i) + '.png'
    im_r = Image.open(path)
    size = (28, 28)
    im_r.thumbnail(size)
    im_r.save(save_path)
```

이 코드를 실행하면, 미리 저장해 둔 영상에서 5프레임 마다 자동적으로 이미지를 추출해서 Number 순서로 저장해 준다.

5. 연구 결과

5.1. 실행 결과

가. 기본 실행

위 <4. 연구 방법>에 나와 있는 코드들을 통해 최종적으로 프로그램을 실행시켰다. Train Data 들을 ***코드 3***을 이용해 학습 시키고 미리 생성해 둔 Test Data로 진행한 '점검 과정'의 정확도를 출력한 결과는 아래와 같다. 학습률 0.001, 반복횟수 55번, 드롭아웃 사용한 상태에서 진행하였다.

실행 결과 1

```
학습 시작!
Epoch: 0001 Avg. cost =  3.450872
Epoch: 0002 Avg. cost =  3.436275
Epoch: 0003 Avg. cost =  3.434416
Epoch: 0004 Avg. cost =  3.398020
                    ⋮
Epoch: 0022 Avg. cost =  1.399898
Epoch: 0023 Avg. cost =  1.136453
Epoch: 0024 Avg. cost =  0.867816
Epoch: 0025 Avg. cost =  0.732504
Epoch: 0026 Avg. cost =  0.586714
```

```
                    ⋮
Epoch: 0039 Avg. cost =  0.023444
Epoch: 0040 Avg. cost =  0.020262
Epoch: 0041 Avg. cost =  0.017201
Epoch: 0042 Avg. cost =  0.014406
                    ⋮
Epoch: 0052 Avg. cost =  0.011159
Epoch: 0053 Avg. cost =  0.009803
Epoch: 0054 Avg. cost =  0.012172
Epoch: 0055 Avg. cost =  0.008253
학습 완료!
```

정확도: 1.0 *#정확도 1은 100%를 의미한다.*

초기에는 오차가 크고 잘 줄어들지 않지만, 학습 진행 횟수 (Epoch)가 증가할수록 (특히 epoch = 10 정도부터), 그 오차 (cost)가 점점 줄어드는 것을 알 수 있다. 이후 테스트 데이터들을 인식시켰다.

100%의 정확도는 비정상정인 값이라고 느껴질 수 있다. 하지만, 찍은 여러 개의 사진들 중 임의로 5개 ~ 10개를 골라 테스트 데이터로, 나머지는 학습 데이터도 분류해 놓았기 때문에 이렇게 나왔을 수 있다고 해석하였다. 즉, 전체적인 모양은 구분 및 인식이 된다는 것은 알 수 있었다. 향후에 데이터를 더 추가하였는데, 이에 대해서는 <4.2 연구 절차 및 개발> 의 <⑥ 인식 정확도 높이기> 에 설명하였다.

테스트 데이터들을 인식한 결과를 ***코드 4***를 한글로 변환하였다. (실제로는 세로로 출력되지만 가로로 변환하였음.)

실행 결과 2

```
ㄱ ㄱ ㄱ ㄱ ㄱ ㄴ ㄴ ㄴ ㄴ ㄴ ㄷ … ㅟ ㅟ ㅟ ㅟ
ㄲ ㄲ ㄲ ㄲ ㄲ
```

나. TTS 데이터 인식 실행

먼저, 위 <4.2 연구절차 및 개발>에서 소개한 ‘이현수입니다’ 데이터들을 인식시켜 보았다. 인식, 변환 코드 및 ***코드 6*** 까지 최종적으로 실행하면, 인식한 초성이 담겨져 있는 배열이 출력된다.

실행 결과 3

```
ㅇ
ㅣ
ㅎ
ㅕ
ㄴ
ㅅ
ㅜ
ㅇ
ㅣ
ㅂ
ㄴ
ㅣ
ㄷ
ㅏ
['ㅇ', 'ㅣ', 'ㅎ', 'ㅕ', 'ㄴ', 'ㅅ', 'ㅜ', 'ㅇ', 'ㅣ', 'ㅂ', 'ㄴ
', 'ㅣ', 'ㄷ', 'ㅏ']
```

이후, ***코드 7*** 과 ***코드 8*** 까지 실행시키면 아래와 같이 결과가 출력되고, 해당 문자을 읽어주는 음성 파일이 ‘FinalTTSFile.mp3’ 로 저장된다.

실행 결과 4

```
이현수입니다
저장 완료!
```

#저장된 파일

이 외에도 ‘안녕하세요이현수입니다’, ‘안녕하세요’ 등의 문장들 역시 인식시켜 보았고, 성공하였다.