REPORT

보고서 작성 서약서

1. 나는 타학생의 보고서를 복사(Copy)하지 않았습니다.
2. 나는 타학생의 보고서를 인터넷에서 다운로드 하여 대체하지 않았습니다.
3. 나는 타인에게 보고서 제출 전에 보고서를 보여주지 않았습니다.
4. 보고서 제출 기한을 준수하였습니다.

나는 보고서 작성시 위법 행위를 하지 않고,
성.균.인으로서 나의 명예를 지킬 것을 약속합니다.

Class Name : Embedded System Design
Subject : Term Project
Professor : 오윤호
Department : 전자전기공학부
Student ID : 2017310350, 2019313481, 2019310649
Name : 이재윤, 한석현, 임용성
Submission Date : 2021. 11. 30

Real Time Face Recognition using MTCNN and FaceNet with TensorRT

이재윤 2017310350 , 한석현 2019313481 , 임용성 2019310649

Electronic and Electrical Engineering, Sungkyunkwan University, Yuljeon 16419, Korea

1 Introduction

교내에서 도서관에 출입하기 위해서는 QR코드를 찍고 들어가야 합니다. 실제로 도서관이 끝나는 10시 정각에 사람들이 빠져나갈 때 QR코드를 찍는 게이트에서 줄을 지어 기다렸던 적이 있었습니다. 저희 팀은 이것을 해결하기 위해 FaceNet 모델을 이용하여 실시간으로 얼굴 인식을 하여 gate 출입을 통제하면 좋지 않을까? 라는 생각을 하였습니다. 수많은 얼굴 인식 모델 중에서 FaceNet 모델을 선정한 이유는 다음과 같습니다.

1. 학교라는 특성상 매년 신입생이 들어오기 때문에 얼굴을 인식하기 위해서는 DNN 모델을 매년 재학습해야합니다.
2. DNN 모델을 학습하는 과정은 매우 오래걸리고, embedded system에서 학습을 한다는 것은 매년 기기를 교체해줘야한다는 것을 의미하므로 이것은 시간과 돈이 낭비됩니다.
3. 주어진 data가 매우 적다는 것입니다. DNN 모델에서는 정확도를 높이기 위해서 빅데이터가 필수적인데, 매년 신입생의 얼굴 데이터를 많이 만든다는 것은 불가능한 일에 가깝습니다.

위의 3가지 문제점을 해결하기 위해서, metric learning 방식을 사용하므로 few-shot learning이 가능하기 때문에 상대적으로 적은 데이터가 필요하고, 매년 DNN을 training 하지 않아도 된다는 장점이 있는 FaceNet을 선택하게 되었습니다.

2 Theoretical Background

2.1 What is facenet?

facenet은 얼굴 사진을 input 받고 그 얼굴의 특징을 추출하여, 128차원의 임베딩 공간에 얼굴 이미지를 특정하는 DNN입니다. 이 embedding space를 토대로 사람을 인식할 수 있습니다. embedding space는 다음 사진과 같습니다.

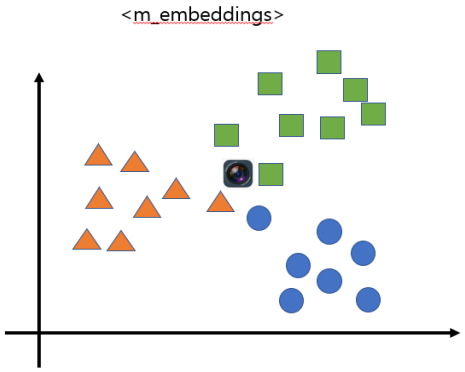


Fig. 1. embedded space.

위의 그림의 도형은 각 사람의 얼굴 이미지입니다. 이것이 위와 같이 128차원의 공간에 존재하고, 같은 사람의 얼굴 이미지는 가깝게, 다른 사람의 얼굴 이미지는 멀게 분포되어있습니다. 예를 들어, 카메라가 얼굴을 input 받아 embedded space에 나타내면 그 점과 가장 가까운 곳에 있는 이미지를 동일한 사람으로 인식하는 방식으로 face recognition을 진행할 수 있습니다. 이 근거는 같은 사람의 face 이미지는 가깝게, 다른 사람의 face 이미지는 멀게 포지션되어있게 facenet이 embedded하기 때문입니다. 이를 구현하기 위해 facenet은 다음의 training과정을 거칩니다.

• **Training process**

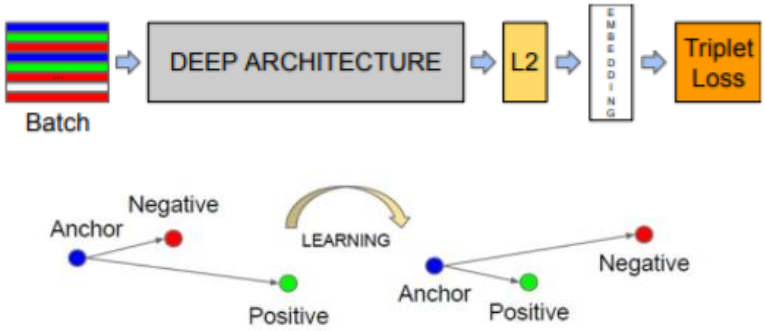


Fig. 2. triplet loss. [1]

Facenet의 training과정은 위와 같습니다. Train image를 batch size로 정의하여 CNN에 input하고 triplet loss라는 손실 함수를 사용하여 CNN을 학습시킵니다. 여기서 Triplet loss란 특정 사람의 이미지를 anchor로 하고, 같은 사람의 이미지지만 가장 벡터 거리가 먼 이미지를 positive로, 다른 사람의 이미지지만 가장 벡터 거리가 가까운 이미지를 negative로 하여, positive는 가깝게, negative는 멀게 하는 것입니다. Facenet 논문에 따르면, triplet loss에 대한 학습 시간은 매우 오래 걸려서 mini batch에 대해 loss update를 수행하여 CNN을 학습시켰다고 기술했습니다. Triplet loss를 loss function 으로 사용하기 때문에, embedded 된 vector는 같은 사람은 가깝게 다른 사람은 멀게 positioning 시킵니다.

• **Inference process**

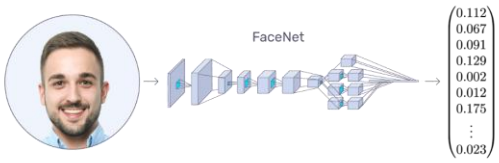


Fig. 3. FaceNet. [2]

사람 이미지가 facenet CNN에 input 되면 그 사람의 이미지의 특징을 추출하여 128차원의 embedding vector를 만듭니다. 이를 토대로 주어진 이미지들에 대한 embedding space를 다음과 같이 형성합니다.

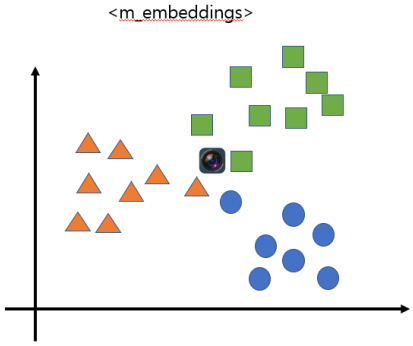


Fig. 4. Embedding Space.

이제 camera를 통해 실시간으로 알려지지 않은 사람의 이미지가 facenet에 input된다면, 마찬가지로 embedding vector를 형성하고, embedding space를 바탕으로 그 사람이 누구인지를 판단하게 됩니다.

2.2 What is MTCNN?

MTCNN은 하나의 이미지에서 다수의 사람의 얼굴을 각각 추출하는 CNN입니다. 다음의 그림은 MTCNN의 결과를 나타낸 것입니다.

Fig. 5. Real time recognition of MTCNN. [3]

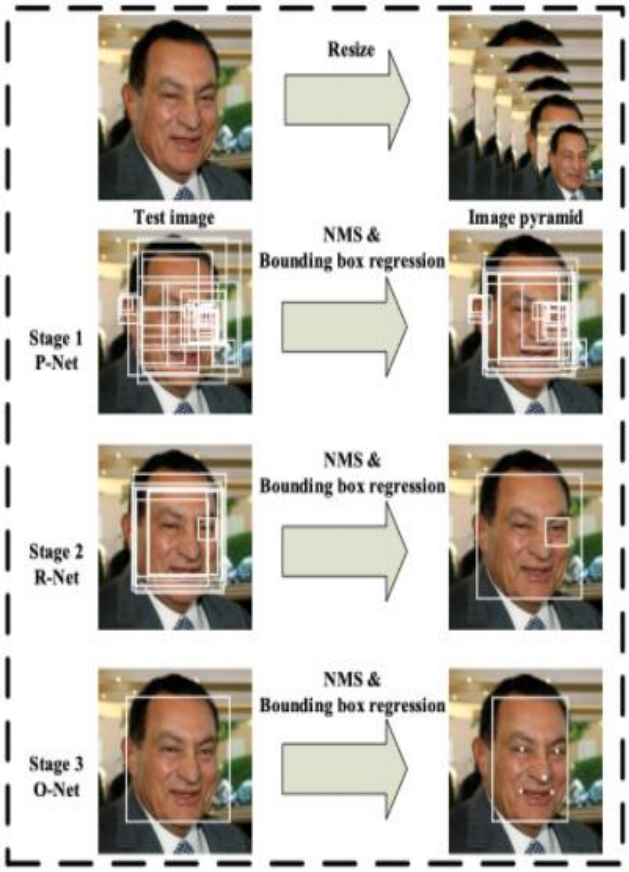


Fig. 6. MTCNN structure. [4]

MTCNN의 구조를 설명하면 다음과 같습니다

좌측의 사진에서 볼 수 있듯이, MTCNN은 P-Net, R-Net, O-Net이 순차적으로 연결된 Cascade 모델입니다.

우선 P-net을 통해 input 이미지를 박스를 이동하면서 scan하는 과정을 거쳐, 얼굴이라고 판단되는 다수의 얼굴들이라고 약하게 추측되는 것들을 추출합니다. 그렇게 되면 얼굴로 추정되는 박스들의 리스트들을 얻을 수 있습니다.

P-net을 토대로 얼굴이라고 추정되는 박스들의 리스트를 받으면, 이제 그 박스가 얼굴인지를 더 정교하게 판단하기 위해 R-Net을 거칩니다. R-Net을 거치면 실제 얼굴인 박스들만 추출되어 NMS와 BBR을 거쳐 O-Net에 전달합니다.

이제, P-Net과 R-Net을 통해 얼굴로 추정하는 박스를 O-Net에 전달합니다. O-Net에서는 얼굴에 해당하는 정보를 더 세밀하게 추상적으로 확인하여 얼굴임을 확인합니다.

2.3 What is k-nn(k-nearest neighborhood) algorithm?

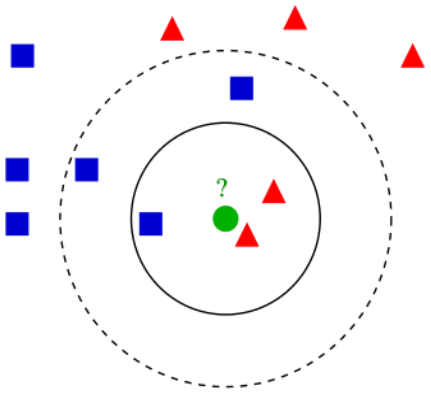


Fig. 7. k-nearest neighborhood algorithm. [5]

k-nn algorithm 은 supervised learning의 한 종류입니다. input data로 부터 가장 가까운 k개의 다른 데이터의 label을 참조하여 input data를 판별합니다. 이때, 거리를 측정할 때에 '유클리디안 거리'를 사용합니다. 만약, k=3이라고 하고, 위의 사진과 같이 파란색 네모 1개, 빨간색 세모 2개가 범위 내에 존재한다면, 빨간색 세모의 빈도가 상대적으로 더 많으므로, 초록색 원은 빨간색 세모와 같은 label로 판별되어집니다. 이것이 k-nn algorithm입니다.

3 Strategy

3.1 past strategies

최초에 저희 팀의 계획은 실시간으로 사람을 인식하기 위해 Facenet의 성능을 향상시키는 것이 목표였습니다. 목표를 위해 다음의 3가지 전략을 세웠습니다.

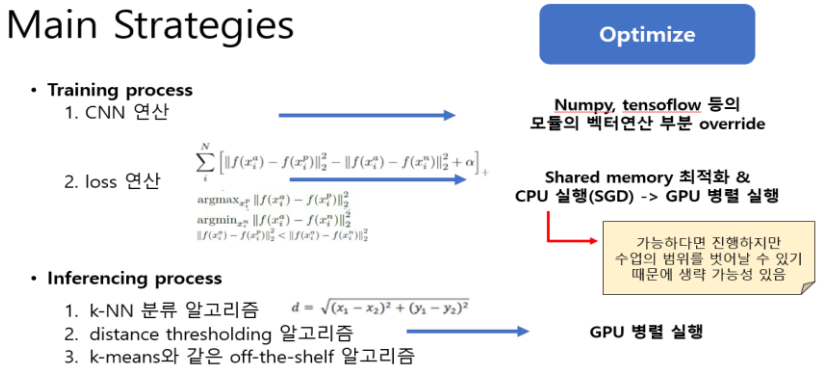


Fig. 8. Past Strategies.

그러나 위의 전략 중, Training 과정의 2번째, loss 연산 부분에서 shared memory를 사용하고, code의 SGD(Stochastic Gradient Descent)를 병렬적으로 처리하는 방식은 교수님께서 너무 어렵기 때문에 진행하지 않아도 된다고 말씀하셨기 때문에, 진행하지 않도록 하였습니다.

3.1.1 vector distance part

우선, Inference 과정에서 사람을 인식할 때, K-NN 알고리즘을 사용하게 되는데, 이때, vector distance를 계산하는 과정에서 저희의 code가 only CPU만 사용하도록 프로그래밍 되어있었는데, 이 부분을 GPU를 이용하여 동작하게 코딩해보았습니다.

```
user14@user14:~/lim$ ./a.out
 Test : 1 , Elapsed Time :  0.029000 ms
 Test : 2 , Elapsed Time :  0.006000 ms
 Test : 3 , Elapsed Time :  0.005000 ms
 Test : 4 , Elapsed Time :  0.006000 ms
 Test : 5 , Elapsed Time :  0.005000 ms
 Test : 6 , Elapsed Time :  0.006000 ms
 Test : 7 , Elapsed Time :  0.008000 ms
 Test : 8 , Elapsed Time :  0.007000 ms
 Test : 9 , Elapsed Time :  0.008000 ms
 Test : 10 , Elapsed Time :  0.007000 ms
--------------------------------------------------------------------------------

 Average Elapsed Time : 0.008700 ms
--------------------------------------------------------------------------------
```

Fig. 9. Elapsed time with only CPU.

```
user14@user14:~/lim$ ./gpu

-----------------------------------------------------

 ElapsedTime = 1.573334 ms

-----------------------------------------------------------------
```

Fig. 10. Elapsed time with GPU.

위의 그림에서 볼 수 있듯이, only CPU만을 사용하여 vector distance를 계산하면 0.0087ms 가 걸리고, CPU와 GPU를 동시에 사용하면 1.57334ms가 걸리는 것을 확인할 수 있습니다. 즉 시간이 오히려 181배 더 걸리는 것을 확인할 수 있었습니다.

이 부분에 대해 정확히 이해하기 위해서 nvprof를 사용하여 왜 시간이 더 오래걸리는 지를 확인하였는데 다음과 같습니다.

```
==11043== Profiling application: ./gpu
==11043== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   49.42%  15.582us         1  15.582us  15.582us  15.582us  gpu_vector_distance(float*, float*, float*)
                   28.76%  9.0680us         2  4.5340us  3.0750us  5.9930us  [CUDA memcpy HtoD]
                                                                              y DtoH]
      API calls:   99.74%  341.60ms         3  113.87ms  28.073us  341.54ms  cudaMalloc
                    0.06%  209.59us         3  69.862us  17.760us  163.70us  cudaFree
                    0.05%  159.90us         1  159.90us  159.90us  159.90us  cudaLaunchKernel
                    0.04%  131.88us        97  1.3590us     729ns  29.793us  cuDeviceGetAttribute
                    0.00%  9.7910us         1  9.7910us  9.7910us  9.7910us  cuDeviceTotalMem
                    0.00%  6.7710us         3  2.2570us  1.2500us  3.3340us  cuDeviceGetCount
                    0.00%  3.2820us         2  1.6410us  1.1460us  2.1360us  cuDeviceGet
                    0.00%  1.8230us         1  1.8230us  1.8230us  1.8230us  cuDeviceGetName
                    0.00%  1.2500us         1  1.2500us  1.2500us  1.2500us  cuDeviceGetUuid
```

Fig. 11. nvprof result of vector distance gpu code.

위 사진에서 볼 수 있듯이, cudaMalloc. 즉 GPU에 메모리를 할당해주는 과정이 전체 time의 99.74%가 소요되는 것을 확인할 수 있습니다. 즉, gpu를 함께 사용하였을 때, 오히려 더 느리게 되는 것은 Memory에 대한 문제 때문입니다.

결국, 위에서 볼 수 있는 것과 같이 오히려 시간이 더 오래 걸리기 때문에 저희 팀은 vector distance를 계산하는 부분을 gpu를 사용하도록 바꾸지 않기로 하였습니다.

3.1.2 CNN override part

CNN의, matrix multiplication을 부분에서 numpy 혹은 tensorflow 함수를 쓰게 되는데, 이 부분에 대해서 저희 팀은 각 모듈의 library에서 행렬곱하는 부분을 override하여 block과 thread를 바꿈으로써 가장 성능이 좋을 때의 block과 thread를 찾으려고 했습니다. 그러나 사용하는 source code에서 tensorRT를 사용하기 때문에 이 부분의 성능을 향상 시키지 않아도 된다고 판단하였고

따라서 진행하지 않았습니다. TensorRT에 대해 간략히 설명하자면, 아래 Fig. 12에서 볼 수 있듯이, relu, bias, 1x1 conv등을 하나의 CBR로 묶고 각 node에서 공통된 부분을 한번에 처리하여 Optimize하는 것입니다. 비록, convoultion 연산인 matrix multiplication 연산을 최적화 하는 것은 아니지만, open source code가 TensorRT로 최적화 하기 때문에 override를 해서는 안된다고 판단하였습니다.

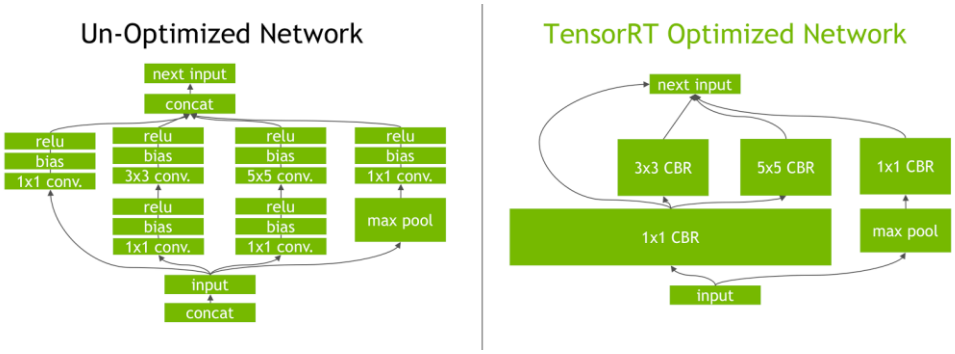


Fig. 12. Optimize strategy by TensorRT. [6]

3.2 Final strategies

결국, 저희 팀의 최적화 전략은 모두 이루어지지 않아도 이미 최적화가 이루어져있다는 것을 알게 되었고, 따라서 다른 전략을 세웠습니다.

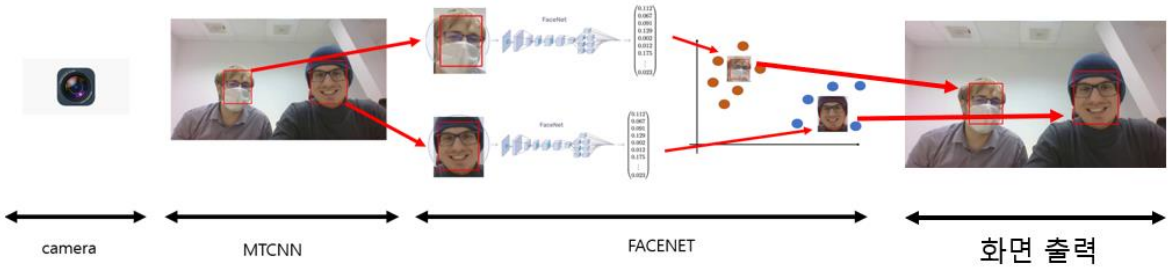


Fig. 13. Face recognition flow of open source code

현재 code는 위와 같습니다. 이번에 사용한 main reference의 소스코드는 face recognition을 수행하는 기능을 담고 있는데, 128차원 임베딩 공간에서 가장 가까운 벡터거리에 있는 이미지만을 가지고 recognition이 수행합니다. 여기에 k-NN 알고리즘을 사용해 인식 정확도 향상을 이루어낼 수 있도록 하였습니다.

k-NN 알고리즘은 어떤 벡터 공간에서 k개의 "k개의 최근접 neighbor 사이에서 과반수 의결에

의해 분류되는" 알고리즘입니다.

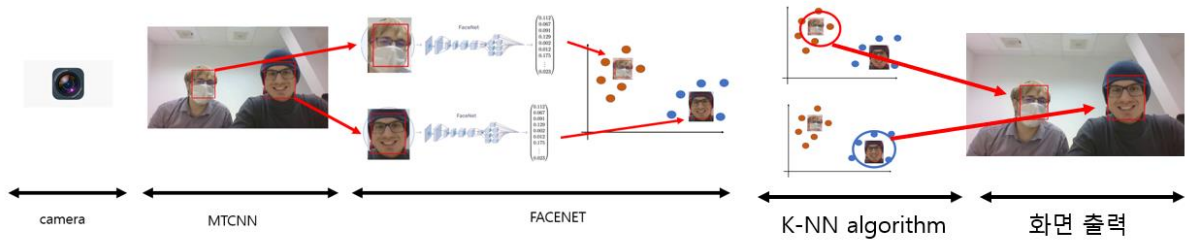


Fig. 14. Face recognition flow of our code

위의 그림에서 볼 수 있듯이, K-NN alogrithm 부분이 추가된 것을 확인할 수 있습니다.

저희 팀은 K-NN alogorithm을 추가하였고, K의 값이 변함에 따라 real time 속도(Fps)와 정확도(%)를 측정하였고, fps와 정확도를 바탕으로 최적의 performance를 내는 k를 찾는 것을 목표로 설정하였습니다.

4 Code Analysis

4.1 Understanding the original source code

저희가 사용한 open source code는 main 함수가 담긴 'main.cpp' 파일 아래 여러 '.cpp' 파일로 구성되어 있습니다. 이 중, 저희가 성능 개선(speedup, accuracy)을 할 여지가 있는 source code를 중심으로 분석해보았습니다.

이중 'main.cpp'에서는 'faceNet.cpp', 'videoStreamer.cpp', 'mtcnn.cpp', baseEngine.cpp' 등' 프로그램 실행에 필요한 library를 불러오는 부분이 포함됩니다. 실시간 카메라 사용을 위한 'baseEngine.cpp', 'videoStreamer.cpp' 등의 library는 성능 개선과 직접적인 관련이 없어 'faceNet.cpp'와 'main.cpp'를 중심으로 분석했습니다.

프로그램이 실행되면, 저장된 CNN 모델('.uff' 파일)과 실시간 카메라 사용을 위해 'baseEngine.cpp', 'videoStreamer.cpp' 에 저장된 코드를 불러옵니다. 이때 카메라에서 들어오는 입력은 'frame' 변수에 행렬 형태로 저장되고, 이 'frame' 변수가 'faceNet.forward(frame, outputBbox);' 구문, 'faceNet.featureMatching(frame);' 구문에 사용됩니다.

또한 카메라에서 오는 영상 정보를 추론할 때, device의 성능에 따라 fps(frame per second)가 정해집니다. device의 성능이 좋을수록 fps는 올라가게 됩니다. 이 open source code에서는 이

fps를 측정할 수 있는 방법을 제공하는데, 'nbFrames' 변수와 시간을 측정할 수 있는 'chrono' 내장 library를 통해 측정할 수 있습니다. 'nbFrames' 변수는 while 반복문 안에서 1씩 증가하기 때문에 추론 횟수와 일치합니다. 이 추론 횟수를 시간(second)으로 나누면 fps가 됩니다.

```
int main()
{
    Logger gLogger = Logger();
    // Register default TRT plugins (e.g. LRelu_TRT)
    if (!initLibNvInferPlugins(&gLogger, "")) { return 1; }

    // USER DEFINED VALUES
    const string uffFile="../facenetModels/facenet.uff";
    const string engineFile="../facenetModels/facenet.engine";
    DataType dtype = DataType::kHALF;
    //DataType dtype = DataType::kFLOAT;
    bool serializeEngine = true;
    int batchSize = 1;
    int nbFrames = 0;
    int videoFrameWidth = 640;
    int videoFrameHeight = 480;
    int maxFacesPerScene = 5;
    float knownPersonThreshold = 1.;
    bool isCSICam = true;
```

Fig. 15. code of 'main.cpp'

앞서 언급한 'faceNet.forward(frame, outputBbox);' 구문은 CNN을 통과시켜 추론을 수행하는 함수, 'faceNet.featureMatching(frame);' 구문은 추론을 거친 embedding vector간의 거리를 측정해 class name을 화면에 띄워주는 함수를 실행하는 부분입니다. 'main.cpp'에서 두 구문이 사용된 부분은 아래와 같습니다.

```
    while (true) {
        videoStreamer.getFrame(frame);
        if (frame.empty()) {
            std::cout << "Empty frame! Exiting...\n Try restarting nvargus-daemon by "
                         "doing: sudo systemctl restart nvargus-daemon" << std::endl;
            break;
        }
        auto startMTCNN = chrono::steady_clock::now();
        outputBbox = mtCNN.findFace(frame);
        auto endMTCNN = chrono::steady_clock::now();
        auto startForward = chrono::steady_clock::now();
        faceNet.forward(frame, outputBbox);
        auto endForward = chrono::steady_clock::now();
        auto startFeatM = chrono::steady_clock::now();
        faceNet.featureMatching(frame);
        auto endFeatM = chrono::steady_clock::now();
        faceNet.resetVariables();
```

Fig. 16. code of 'main.cpp'

다음으로, 'faceNet.cpp'에 포함된, 저희의 성능 개선에 영향을 미치는 함수들에 대한 설명입니다. 아래 함수들은 'main' 함수에서 등장하는 순서로 나열했습니다.

- **'getCroppedFacesAndAlign'** 함수는 카메라로부터 들어온 입력에서 frame(사진) 중 얼굴에 해당하는 위치를 추출해 160x160 크기로 resize하는 역할을 합니다.

- **'preprocessFaces'** 함수는 resize된 얼굴 사진을 CNN에 입력할 수 있도록 이미지를 flatten된 vector array에 저장하고, 정규화를 수행합니다.

- **'doInference'** 함수는 실제 faceNet의 CNN 연산과 embedding 공간에 출력되는 128차원 벡터에 관여하는 함수입니다. cudaStream을 통해 GPU memory allocation과 kernel 함수 부분을 비동기적으로 실행합니다. 'forward', 'forwardAddFace' 함수에 의해 호출됩니다.

- 이후의 function에는 **'KnownID'**라는 구조체가 등장합니다. 이 구조체 변수에는 'className', 'classNumber', 'embeddedFace'를 member로 가지고 있습니다. 'className'은 'imgs' 디렉토리에 있는 이미지 파일 이름이 그대로 입력됩니다. 'classNumber'는 이 항목이 전체 'KnownID' 구조체 변수 중 몇 번째 index인지를 기억하게 되는 역할을 하는 정수형 변수입니다. 'embeddedFace'는 CNN을 통과한 embedding 공간에 존재하는 128차원 벡터를 저장하는 역할을 합니다. 'KnownID'는 'faceNet.cpp'에서 'm_knownFaces' 변수, 'forwardAddFace' 함수의 'person' 변수에 적용되며 하나의 이미지 파일에 대한 얼굴의 특징벡터를 저장하는 역할을 합니다.

- **'forwardAddFace'** 함수는 앞선 'preprocessFaces', 'doInference' 함수를 호출하는 함수이며 프로그램 실행 시 'person' 구조체에 얼굴과 embedding 벡터 정보를 임시 저장한 후 'm_knownFaces'에 추가하는 역할을 합니다.

- **'addNewFace'** 함수는 프로그램 도중에 키보드 'n' 입력이 들어오면, 'img' 디렉토리에 새롭게 추가된 이미지에 대한 embedding 벡터 계산을 추가로 수행할 수 있게 해줍니다. 여기서도 마찬가지로 'forwardAddFace' 함수가 사용됩니다.

- **'forward'** 함수는 'forwardAddFace' 함수와 다르게, 현재 실시간으로 카메라에서 들어오는 frame에 대한 embedding 벡터를 계산해 embedding 공간에 mapping하는 작업을 수행합니다.

- **'featureMatching'** 함수는 'forwardAddFace', 'forward' 함수를 거친 128차원 embedding 공간에 mapping된 벡터들 간의 거리를 계산해서 최종 'winner'의 class name을 화면에 출력하는 작업을 수행합니다. 여기서 'winner'란 가장 거리가 가까운 얼굴 이미지를 기억하는 index 값입니다. 예를 들어, 'm_knownFaces' 구조체에 'className'이 '한석현', 'classNumber'가 23인 index가 있고 이 index가

'winner'가 된다면, 화면에는 '한석현'이라는 'className'이 출력됩니다. 가장 가까운 거리를 구하기 위한 계산도 이 함수에 포함됩니다. Embedding 공간 관련 그림과 위 함수와 관련된 코드 일부는 아래와 같습니다.

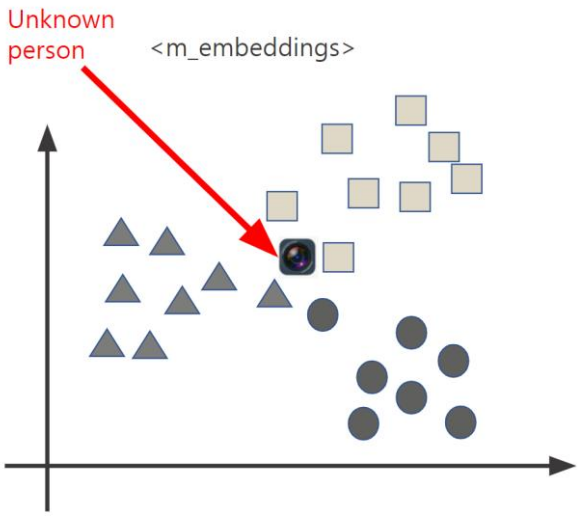


Fig. 17. feature vectors in embedding space(simplified version of 2-D plane)

```
210     for(int i = 0; i < (m_embeddings.size()/128); i++) {
211         double minDistance = 10.* m_knownPersonThresh;
212         float currDistance = 0.;
213         int winner = -1;
214         for (int j = 0; j < m_knownFaces.size(); j++) {
215             std:vector<float> currEmbedding(128);
216             std::copy_n(m_embeddings.begin()+(i*128), 128, currEmbedding.begin());
217             currDistance = vectors_distance(currEmbedding, m_knownFaces[j].embeddedFace);
218             // printf("The distance to %s is %.10f \n", m_knownFaces[j].className.c_str(), currDistance);
219             // if ((currDistance < m_knownPersonThresh) && (currDistance < minDistance)) {
220             if (currDistance < minDistance) {
221                     minDistance = currDistance;
222                     winner = j;
223             }
```

Fig. 18. partial code snippet of function 'featureMatching' of 'faceNet.cpp'

이후 'featureMatching' 함수에서는 'cv::putText~~'와 같은 구문으로 가장 가까운 거리에 있는 index의 'className'을 화면에 출력하게 됩니다.

4.2 Description of code modulation to get better accuracy

저희는 speedup을 진행할 수 없는 전략 속에서, 'nbFrames' 변수를 이용해서 '정확도'를 측정하기 위한 방법을 구상했습니다. 카메라로 실시간 face recognition을 수행하는 것과 open source code에서 recognition을 수행한 전체 frame을 반환하는 변수가 있다는 것에 착안해서, 세 사람 혹은 'new person'으로 추론할 때의 frame count도 같이 계산하는 방식으로 정확도 측정을 수행했습니다.

예를 들어, 총 100개의 frame이 추론되었고 그 중에 '한석현' class가 10개, '임용성' class가 80개, '이재윤' class가 5개, 'new person' class가 5개의 frame이 기록되었다면 임용성 학우에 대한 인식 정확도는 80퍼센트가 되는 방식으로 정확도 측정을 수행했습니다.

정확도 측정을 위해 원본 source code를 수정했으며, 바뀐 부분에 대한 screenshot은 다음과 같습니다.

```
94          // inserted to get return value
95          std::string infperson;
96          infperson = faceNet.featureMatching(k_NN,frame);
97          auto endFeatM = chrono::steady_clock::now();
98          faceNet.resetVariables();
99
100         cv::imshow("VideoSource", frame);
101         nbFrames++;
102         // count each class name
103         if (infperson == "lys") nbFramesLYS++;
104         else if (infperson == "hsh") nbFramesHSH++;
105         else if (infperson == "ljy") nbFramesLJY++;
106         else nbFramesNewPerson++;
```

```
137     std::cout << "Counted " << nbFrames << " frames in " << double(milliseconds)/1000. << " seconds!" <<
138              " This equals " << fps << "fps.\n";
139     std::cout << "Appearance of LYS : " << nbFramesLYS << " frames, percentage : " << double((100*nbFramesLYS)/nbFrames) << "%\n
140     std::cout << "Appearance of HSH : " << nbFramesHSH << " frames, percentage : " << double((100*nbFramesHSH)/nbFrames) << "%\n
141     std::cout << "Appearance of LJY : " << nbFramesLJY << " frames, percentage : " << double((100*nbFramesLJY)/nbFrames) << "%\n
142     std::cout << "Appearance of New Person : " << nbFramesNewPerson << " frames, percentage : " << double((100*nbFramesNewPerson
143     printf("K-NN value k = %d \n", k_NN);
```

Fig. 19. added line in 'main.cpp' for print accuracy

다음은 k-NN 알고리즘을 적용하기 위해 'faceNet.cpp'에서 'featureMatching' 함수를 수정했습니다. k=4인 k-NN 알고리즘을 적용하기 위해 'winner'와 'minDistance'의 index 수를 1개에서 4개로 설정했습니다.

'minDistance' array의 값들을 충분히 큰 값으로 설정한 후, 'currDistance' 값과 매번 비교하면서 가까운 값이 들어올 때마다 'minDistance' list를 업데이트하면 마지막에 가장 작은 4개의 벡터 거리가 저장됩니다.

이때, 'minDistance' array에서 'minDistance' 보다 작은 값으로 업데이트하려면, 값들 중 가장 큰 값을 버리고 새로운 값을 저장해야 가장 작은 k개의 값만 저장할 수 있습니다. 위 내용을 그대로 담은 알고리즘을 아래 예시를 통해 설명하겠습니다.

Times (k = 4)	minDistance[0]	minDistance[1]	minDistance[2]	minDistance[3]	New Input
1	10	10	10	10	5
2	5	10	10	10	3
3	5	3	10	10	11
4	5	3	10	10	7
5	5	3	7	10	2
6	5	3	7	2	4
7	5	3	4	2	1
8	1	3	4	2	-

Table Ⅰ. Example of determining 'minDistance' array (k-NN algorithm applied)

첫 번째 시행에서 'minDistance' array는 모두 충분히 큰 값인 10으로 설정되어있습니다. 여기서 다음 값으로 5가 들어오면, 가장 큰 값인 10이 5로 바뀌게 됩니다. 이어서 'minDistance' array의 최대값보다 더 작은 값이 들어오면, 그 최대값과 교체하는 시행을 계속 하게 되면, 최종적으로 'minDistance'는 'currDistance' 중 가장 작은 값 4개를 가지게 됩니다.

동시에 'winner' array에는 'm_knownFaces' 중 가장 가까운 거리에 있는 4가지의 index가 저장되게 됩니다. 이를 위해서는 가장 큰 값이 'minDistance'의 네 자리 중 어느 곳에 저장되어있는지 알고 있는 것이 효율적입니다. 여기에 'biggest' 변수를 사용합니다. 가장 큰 값을 가지는 index를 저장하고 있다가, 'minDistance'의 최대값보다 작은 값이 들어오면 최대값과 값을 맞바꾸게 됩니다. 이때, biggest의 값을 update하는 과정도 필요합니다. 구현 내용은 아래 코드에 포함되어있습니다.

가장 가까운 k개의 거리를 찾게 된 후에는 k개 중 가장 많이 나온 'className'을 찾아야 합니다. 이를 위해 'winner'에서 가장 많이 출현한 'className'을 저장하는 코드를 추가했습니다. 아래에 해당 코드를 첨부했습니다.

```
255         vector<string> winner_name;
256         int freq, count = 1;
257         //string winner;
258         for (int j = 0; j < K_NN; j++) {
259             winner_name.push_back(m_knownFaces[winner_index[j]].className.substr(0, 3));
260         }
261         for (int j = 0; j < K_NN; j++) {
262             freq = 1;
263             for (int k = 0; k < K_NN; k++) {
264                 if (winner_name[j] == winner_name[k]) freq += 1;
265                 if (freq >= count) {
266                     winner = winner_name[j];
267                     count = freq;
268                 }
269             }
270         }
```

Fig. 20. Modified code for finding mininum distance

또한 'winner'의 'className'을 return할 수 있도록 featureMatching' 함수의 type을 변경했습니다. 이는 original 코드와의 정확도를 비교하기 위한 인물별 frame 계산에 사용되며, 수정된 'main.cpp' 코드에서 'infperson = faceNet.featureMatching(k_NN,frame);' 구문을 사용할 수 있도록 변경했습니다.

```
208 std::string FaceNetClassifier::featureMatching(cv::Mat &image) {
209     int winner = 0;
```

:

```
236     return m_knownFaces[winner].className.substr(0, 3);
237 }
```

```
91         std::string infperson;
92         infperson = faceNet.featureMatching(frame);
```

5 Result Analysis

우선 사진을 구할 수 있는 조원 3명(hsh, ljy, lys)에 대해 embedding을 진행하였습니다. 한 사람당 19장의 사진 데이터로 embedding을 진행하였으며 각 데이터 사진의 이름은 아래 사진에서 확인할 수 있습니다.

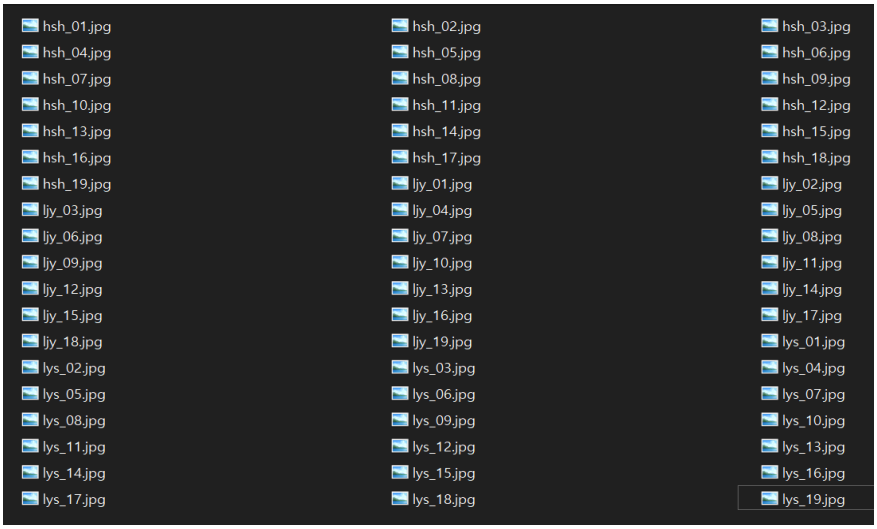


Fig. 21. dataset of three people(lys, hsh, ljy).

5.1 기존 코드와 바꾼 코드의 카메라 인식 결과 비교 & k 최적값 구하기

본 프로젝트의 목표인 정확도의 향상을 확인하기 위해서 기존 코드와 바꾼 코드를 이용하여 얼굴 인식을 진행하고 정확도를 비교했습니다. 또한, k를 4부터 8까지 변화하면서 정확도를 비교하고 k의 최적값을 찾았습니다. 우선, 오차를 최대한 줄이기 위해서 3명에 대해 모두 결과를 비교하였으며 한 사람당 5번의 실행 후, 평균값을 이용하여 정확도를 비교하였습니다. 이때 정확도는 전체 프레임 중에서 해당하는 사람의 프레임의 비율을 뜻합니다. 가만히 카메라를 응시하면 정확도가 비교하기 힘들 정도로 높게 나오기 때문에 보다 정확한 비교를 위해서 추론을 진행하는 시간은 약 15초로 비슷하게 유지하였으며 처음 5초는 카메라를 응시, 다음 5초는 상하좌우로 고개를 돌렸습니다. 그리고 마지막 5초는 다시 카메라를 응시하였으며 중간에 표정변화를 주어 정확도의 차이를 만들었습니다.

또한, 이미 embedding이 진행된 3명의 knownface 3명과 embedding 되지 않은 새로운 new person을 포함하는 총 4명의 사진에 대해서도 기존 코드와 바꾼 코드의 카메라 인식 결과를 비교

하였습니다. 멀티 얼굴 인식 결과의 정확도를 비교하기 위해 4명의 사진을 모두 한 사진에서 같이 인식하였습니다.

이제 위 과정을 통해 나온 결과를 확인하겠습니다. 우선 아래의 첫번째 그림은 face recognition을 진행한 후 출력된 메시지의 한 예시입니다. 실행한 시간과 프레임을 통해 fps를 정상적으로 출력하고 있습니다. 이때, fps란 시간 당 추론한 프레임의 개수로써, face recognition의 속도를 의미한다고 할 수 있습니다. 또한, 각 class에 대한 프레임의 개수와 비율을 출력하여 face recognition 결과의 정확도를 바로 확인할 수 있습니다. 마지막으로는 k 값을 보여주고 있습니다.

그 아래 그림들은 face recognition을 진행한 결과 화면에 나타나는 결과의 한 예시들입니다. 정상적으로 얼굴의 범위를 탐지하고 추론 결과를 박스 왼쪽 하단에 출력하고 있는 것을 통해 face recognition이 잘 실행되는 것을 알 수 있습니다.

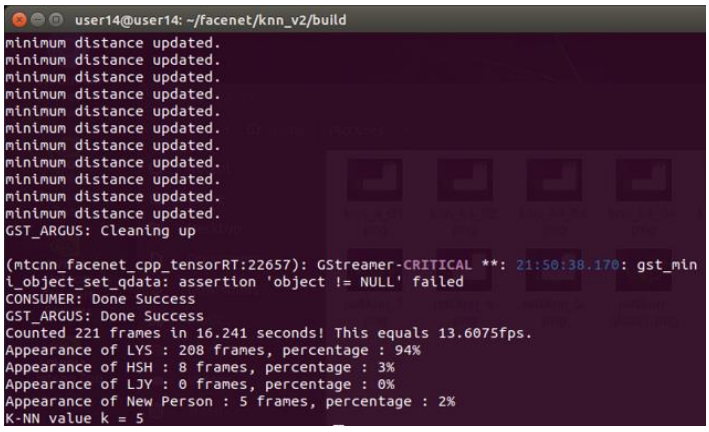
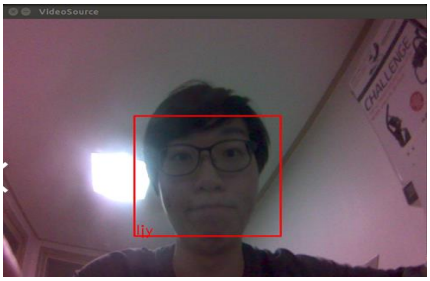
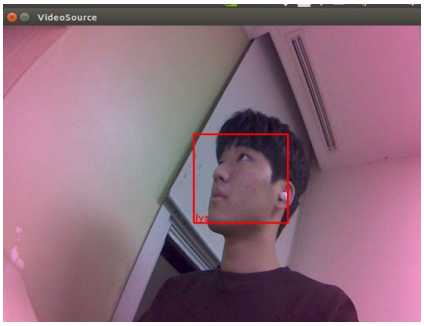


Fig. 22. Output Message.

(a)



(b)



(c)

Fig. 23. Real time face recognition (a) hsh, (b) ljy, (c) lys.

아래 표는 3명에 대해 5번씩 실행한 값을 모든 값을 k의 값에 따라 정리한 결과입니다.

임용성

	fps						정확도						
	notknn	k=4	k=5	k=6	k=7	k=8		notknn	k=4	k=5	k=6	k=7	k=8
1	15.7717	13.9931	13.6075	14.3015	14.7426	14.6935	1	74	85	94	87	96	92
2	13.8889	13.0704	14.9824	15.115	14.7269	13.8163	2	96	74	95	91	83	76
3	15.6477	14.0331	14.8263	13.9564	14.1807	14.0488	3	82	84	81	95	87	97
4	14.188	14.4942	14.7612	17.2414	13.1579	14.3673	4	75	92	97	69	96	94
5	14.5103	16.9536	13.9811	14.4437	14.5028	14.015	5	79	70	90	95	82	96
평균	14.80132	14.50888	14.4317	15.0116	14.26218	14.18818	평균	81.2	81	91.4	87.4	88.8	91

한석현

	fps						정확도						
	notknn	k=4	k=5	k=6	k=7	k=8		notknn	k=4	k=5	k=6	k=7	k=8
1	16.9769	18.7272	13.8224	16.006	17.7341	15.0309	1	84	87	84	93	90	89
2	16.3913	16.6913	16.5176	17.6905	15.1135	14.7783	2	83	95	86	83	90	84
3	16.7805	16.324	14.2798	18.0768	15.7096	15.0847	3	87	91	88	85	89	85
4	16.2055	16.8316	16.5502	17.2985	16.5781	13.7199	4	69	76	85	84	88	94
5	15.2302	14.0441	17.0302	16.5975	15.1245	14.2492	5	79	83	89	92	91	92
평균	16.31688	16.52364	15.64004	17.13386	16.05196	14.5726	평균	80.4	86.4	86.4	87.4	89.6	88.8

이재윤

	fps						정확도						
	notknn	k=4	k=5	k=6	k=7	k=8		notknn	k=4	k=5	k=6	k=7	k=8
1	13.513	17.7349	19.2734	12.9791	14.6541	14.668		85	81	70	93	75	79
2	15.4898	16.5062	18.7574	14.5362	13.8728	13.5583		88	85	89	93	94	84
3	15.6047	15.8441	19.563	10.9315	14.5592	15.2887		97	84	63	92	88	94
4	11.6964	17.5988	17.4205	13.1229	15.0012	15.4297		67	79	76	87	85	82
5	13.2405	15.2213	14.7601	14.6937	13.5575	12.3863		92	92	92	84	81	91
평균	13.90888	16.58106	17.95488	13.25268	14.32896	14.2662	평균	85.8	84.2	78	89.8	84.6	86

각 라인 평균	15.00902667	15.87119333	16.00887333	15.13271333	14.88103333	14.34232667	각 라인 평균	82.46666667	83.86666667	85.26666667	88.2	87.66666667	88.6

Fig. 24. Result Data according to the value of k with 5 times for each.

아래 표와 그래프는 위 모든 데이터의 값의 평균을 구한 결과입니다. 우선, fps의 경우에는 k 값에 따라 큰 차이가 없음을 알 수 있습니다. 모두 15 근처의 값이 나오는 것은 k 값에 상관없이 속도는 비슷함을 의미합니다. 그에 반해, k 값에 따라 정확도는 달라지는 것을 확인할 수 있습니다. KNN 알고리즘을 적용하지 않았을 때보다 KNN을 적용한 후 바뀐 코드에서는 모두 이전에 비해 정확도가 향상한 것을 확인할 수 있습니다.

그 중에서도 k가 6일 때와 8일 때 가장 높은 정확도를 보이는 것을 알 수 있습니다. 대체적으로 k

가 커짐에 따라 정확도가 늘어나는 경향을 띄지만, k가 8일 때는 fps가 가장 낮기 때문에 k의 최적값은 6이라고 할 수 있습니다. fps가 낮다는 것은 프레임을 인식하는 속도가 낮다는 뜻이므로 가장 성능이 좋을 때의 k의 값은 6이라고 결론 지을 수 있습니다.

k	fps	k	정확도
1	15.00902667	1	82.46666667
4	15.87119333	4	83.86666667
5	16.00887333	5	85.26666667
6	15.13271333	6	88.2
7	14.88103333	7	87.66666667
8	14.34232667	8	88.6

Table Ⅱ. Summary of result data

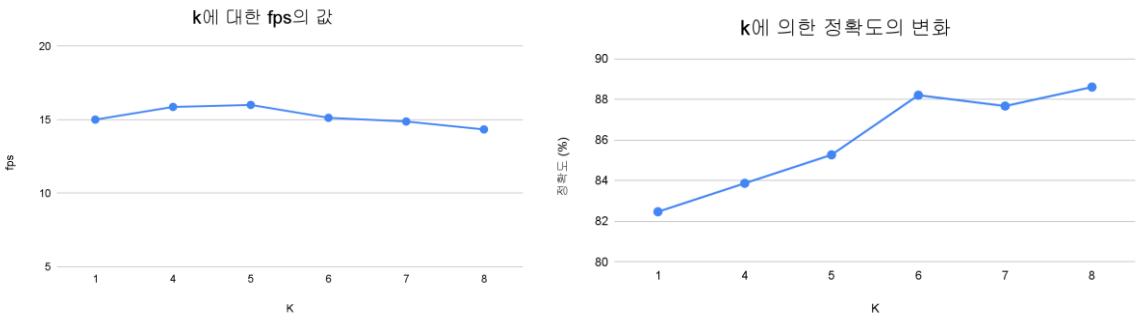


Fig. 25. Result Graph according to the value of k with 5 times for each.

다음은 4명의 사진에 대해 face recognition을 진행한 결과를 살펴보겠습니다. 우선 첫번째 사진은 KNN 알고리즘을 적용하지 않은 이전 코드에 대한 face recognition의 결과입니다. 대부분의 사람을 정확히 인식하지만 왼쪽의 한석현(hsh)에 대해서 ljy로 잘못 인식하고 있는 것을 확인할 수 있습니다. 또한 두번째 사진은 KNN 알고리즘을 적용하여 k의 값이 6일 때의 결과입니다. 모든 사람에 대해서 정확하게 결과가 출력되고 있는 것을 확인할 수 있습니다.

Fig. 26. Real time face recognition of multi-faces (a) original code , (b) knn code

6 conclusion

처음에 목표로 했던 faceNet train, inference open source 코드의 optimization을 통해 최적의 block, thread를 찾아 성능 향상을 하려 했지만, open source 를 embedded 하는 과정에서 문제가 있었습니다. Github에 따르면, tensorflow r1.7, Ubuntu 14.04 with Python 2.7 and 3.5 사양으로 tested 되었다 명시되어있었고, 이를 따르기 위해 Ubuntu 14.04 버전인 Jetpack 3.3을 기반으로 Jetson Nano 를 downgrade 하였습니다. 하지만 다음의 2가지 문제가 발생하였습니다.

1. Tensorflow r1.7을 설치하는 과정에서, build를 지원하는 bazel 버전이 jetson nano 의 architecture(arm64/aarch64)와 호환되지 않아 build를 진행할 수 없었습니다.
2. jetpack 3.3에 맞는 tensorflow를 Nvidia 공식 docs를 확인하여 주어진 명령어를 실행하여 'tensorflow < 2' 의 버전을 다운받고 진행하였을 때, github에 명시되어있는 과정을 따라 진행하던 중에, main file에서 모듈을 import하는 부분에서 오류가 발견되었습니다. 같은 디렉토리 안에 있는 다른 파일을 import하는 과정에서 분명 같은 디렉토리 안에 존재함에도 인식하지 못한다거나, 다른 디렉토리안에 존재하지만 import시에는 같은 디렉토리 안에 존재하는 것처럼 코드가 잘못짜여져 있었기 때문에, 이 open source code로는 진행할 수 없다 판단하였습니다.

따라서 최초 strategy를 변경했는데, Real-time multi face recognition을 수행할 수 있는 open source 코드를 이용하였습니다. 사용하는 open source code에서 CNN 연산에 tensorRT가 이미 적용되어있어 CNN의 tensor 연산을 override하는 optimization이 필요가 없었고, 'vector_distance' GPU 코드의 CPU에 대한 speedup의 저하를 확인하였습니다. 따라서 성능(속도)적인 측면에서는 optimization을 할 수 없다 판단하였습니다. 결국, 저희의 Main goal은 k-NN 알고리즘을 적용해서 k에 따른 정확도의 변화를 확인하고 정확도 향상을 이뤄내는 것을 목표로 설정하였습니다.

이후 'fps' 개념으로 추론 정확도를 측정할 수 있는 코드를 추가해 original 코드와 k-NN 알고리즘을 적용한 프로그램 간 정확도 비교를 손쉽게 할 수 있게 만들었습니다.

최적화 결과는 5.5%p의 정확도 향상을 보였는데, 이는 가장 근접 이웃과 비교했을 때(k=1)보다 k-NN 알고리즘을 적용했을 때(k=6) fps 대비 가장 높은 정확도를 보였습니다.

추론 속도를 대표하는 parameter인 fps의 변화와 K에 따른 정확도의 변화를 확인하고 가장 성능과 정확도가 상대적으로 좋은 k=6인 k-NN 알고리즘을 적용한 코드가 각 인물 당 사진의 수가 19개일 때의 저희 3명 얼굴 인식에 대해서 가장 performance가 뛰어난 것을 확인할 수 있었습니다.

이제, 처음에 제시된 문제를 다시 제시하면 다음과 같습니다.

1. 학교라는 특성상 매년 신입생이 들어오기 때문에 얼굴을 인식하기 위해서는 DNN 모델을 매년 재학습해야합니다.
2. DNN 모델을 학습하는 과정은 매우 오래걸리고, embedded system에서 학습을 한다는 것은 매년 기기를 교체해줘야한다는 것을 의미하므로 이것은 시간과 돈이 낭비됩니다.
3. 주어진 data가 매우 적다는 것입니다. DNN 모델에서는 정확도를 높이기 위해서 빅데이터가 필수적인데, 매년 신입생의 얼굴 데이터를 많이 만든다는 것은 불가능한 일에 가깝습니다.

저희는 위의 문제를 다음과 같이 해결했습니다. 1, 2번의 경우에는 FaceNet 모델을 사용함으로써 DNN을 학습하지 않아도 imgs/ 디렉토리에 사진을 추가하기만 하면 Embedding Space에 이미지가 나타나므로 Face Recognition을 진행할 수 있습니다.(하지만, 데이터가 매우 적을경우(ex) 3개) k의 값을 낮추어야 합니다). 3번의 경우에는 인당 19개의 데이터라는 대부분의 DNN 모델에 비해 매우 적은 데이터임에도 불구하고 k=6일 때, 88.2% 라는 높은 정확도를 기록함으로써 해결했음을 확인할 수 있습니다.

Reference

[1] Florian Schroff et al., FaceNet: A Unified Embedding for Face Recognition and Clustering, arXiv:1503.03832v3

[2] "[Face Recognition] 2. FaceNet review," accessed Nov 29, 2021 ,https://hwangtoemat.github.io/paper-review/2020-04-02-FaceNet-%EB%82%B4%EC%9A%A9/

[3] "Real-world adversarial attack on MTCNN face detection system" accessed Nov 29,2021,https://paperswithcode.com/paper/real-world-attack-on-mtcnn-face-detection

[4] "Grind Face Detection [1] MTCNN" accessed Nov 29, 2021, https://yeomko.tistory.com/16

[5] "*k*-nearest neighbors algorithm" accessed Nov 29, 2021, https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm

[6] "NVIDIA TensorRT" accessed Nov 29, 2021, https://developer.nvidia.com/tensorrt#features

[7] Yunho Oh. (2021). ICE3028_41. Embedded System Design. [PowerPoint Slides]. Available:

[8] Yunho Oh. (2021). '*Performance Characterization of High-Level Programming Models for GPU Graph Analytics*'. IEEE

[9] "Face Recognition using Tensorflow", accessed Nov 29, 2021, https://github.com/davidsandberg/facenet

[10] "Face Recognition for NVIDIA Jetson (Nano) using TensorRT", accessed Nov 29, 2021, https://github.com/nwesem/mtcnn_facenet_cpp_tensorRT