제공해주신 코드는 전반적으로 매우 깔끔하게 잘 구성되어 있습니다. 이 코드는 **특정 인물들의 얼굴 사진을 분석하여 AI가 이해할 수 있는 데이터(임베딩)로 변환하고, 이를 초고속으로 검색할 수 있는 데이터베이스(FAISS)로 저장하는 파이프라인**입니다.

이해를 돕기 위해 전체적인 파이프라인의 구조를 먼저 살펴보면 좋습니다.

코드가 어떤 흐름으로 작동하는지 구체적인 단계별로 설명해 드리겠습니다.

---

1. 환경 설정 및 AI 모델 준비

- 클라우드 환경(Google Colab)에서 필요한 라이브러리들을 설치하고 구글 드라이브를 연동하여 데이터를 읽고 쓸 준비를 합니다.
- 얼굴의 위치를 찾아내는 역할(Detection)로 **InsightFace**의 `buffalo_l` 모델을 사용합니다.
- 얼굴을 구분하는 핵심 역할(Recognition/Embedding)로 최신 고성능 모델인 **AdaFace**를 Hugging Face에서 다운로드하여 PyTorch 기반으로 GPU에 로드합니다.

2. 데이터 증강 (Data Augmentation)

- `Albumentations` 라이브러리를 사용해 사진의 개수를 인위적으로 늘립니다.
- 원본 사진 한 장에 좌우 반전, 밝기 및 대비 조절, 최대 15도의 회전 효과를 주어 여러 장의 파생 이미지를 만들어냅니다.
- 이는 조명이나 얼굴 각도가 조금씩 다른 상황에서도 AI가 사람을 잘 인식할 수 있도록 돕는 역할을 합니다.

3. 얼굴 탐지 및 정렬 (Detection & Alignment)

- 이미지 내부에서 사람의 얼굴을 찾아내는 과정은 일반적인 객체 탐지(YOLO 등) 모델이 사물에 박스를 치는 원리와 매우 유사합니다.
- InsightFace 모델이 이미지에서 얼굴의 위치(Bounding Box)와 주요 5개 지점(눈, 코, 양 입꼬리)을 찾아냅니다.
- 찾아낸 주요 지점을 기준으로 고개가 삐뚤어져 있다면 정면을 바라보도록 이미지를 반듯하게 회전시키고, 112x112 픽셀의 정사각형 크기로 깔끔하게 잘라냅니다.

4. 특징 추출 (Embedding Extraction)

- 잘라낸 정규화된 얼굴 이미지를 AdaFace 모델에 통과시킵니다.
- 모델은 이 얼굴만의 고유한 특징들을 수백 개의 숫자로 이루어진 하나의 배열(임베딩 벡터)로 압축해 냅니다.
- 이 코드의 핵심은 한 이미지에서 **원본의 벡터, 증강된 이미지들의 벡터, 그리고 이들의 평균 벡터(Average Embedding)**를 모두 추출하여 저장한다는 점입니다. 이 평균 벡터가 인식률을 크게 높여줍니다.

5. 벡터 데이터베이스 저장 (FAISS Indexing)

- 추출된 모든 얼굴 벡터 데이터들의 길이를 1로 맞추는 정규화(L2 Normalization) 작업을 거칩니다.
- 메타의 오픈소스 라이브러리인 **FAISS**를 사용하여 수많은 벡터들 사이의 유사도를 순식간에 계산할 수 있는 인덱스를 생성합니다.
- 완성된 풀스택 서비스 관점에서 보면, 이 과정은 나중에 새로운 얼굴 사진이 들어왔을 때 기존 데이터와 대조해 누구인지 빠르게 찾을 수 있도록 아주 가볍고 빠른 로컬 벡터 DB(Vector DB)를 구축한 것입니다.
- 마지막으로 벡터 정보가 담긴 `.index` 파일과 누구의 얼굴인지 적힌 이름표(Label)인 `.pkl` 파일을 드라이브에 안전하게 저장합니다.

데이터를 추출하고 저장(Train/Indexing)하는 파트가 아주 훌륭하게 작성되었습니다.

이제 저장된 벡터 데이터를 활용해서 카메라나 영상에서 실시간으로 사람의 얼굴을 식별해 내는 추론(Inference) 단계의 코드를 작성해 보거나, 이를 서버에 올려 API 형태로 통신하는 구조를 구상해 볼 수 있을 것 같습니다. 다음 단계로 어떤 부분을 함께 진행해 볼까요?

뭐 어디코드가 뭘 한다는건지 알수 있어야지

아, 제가 너무 뭉뚱그려 설명해 드렸군요! 코드를 직접 다루셔야 하니, **어떤 줄의 코드가 무슨 역할을 하는지** 코드 블록별로 딱 떨어지게 짚어드리겠습니다.

핵심적인 흐름을 따라가며 7가지 단계로 코드를 해부해 보겠습니다.

---

1. 얼굴 탐지기 (Detector) 준비

이 코드는 사진에서 **얼굴이 어디에 있는지 네모 박스를 치는 역할**을 합니다.

Python

```
# --- 2. InsightFace: 얼굴 탐지(Detection) 설정 ---
providers = ['CUDAExecutionProvider', 'CPUExecutionProvider']
detector = insightface.app.FaceAnalysis(name='buffalo_l', providers=providers, allowe
detector.prepare(ctx_id=0, det_size=(640, 640))
```

- `insightface.app.FaceAnalysis(...)` : InsightFace 라이브러리에서 `buffalo_l` 이라는 모델을 불러와 얼굴의 위치(Bounding Box)와 눈, 코, 입의 위치(Landmarks)를 찾을 준비를 합니다.

2. 얼굴 식별 모델 (AdaFace) 로드

탐지기로 찾은 얼굴이 **누구의 얼굴인지 고유한 특징을 뽑아낼 AI 뇌**를 준비하는 과정입니다.

Python

```
# --- 4. PyTorch 모델 로드 및 GPU 할당 ---
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
adaface_model = net.build_model('ir_101') # AdaFace 모델 껍데기 생성

statedict = torch.load(model_path, map_location=device)["state_dict"] # 학습된 가중치
# ...(중략)...
adaface_model.load_state_dict(model_statedict) # 껍데기에 가중치 입히기
adaface_model.to(device) # GPU 메모리에 모델 올리기
```

- 다운로드한 `adaface_ir101_webface12m.ckpt` 파일(가중치)을 PyTorch 모델에 덮어씌우고 GPU에서 돌아가도록 세팅합니다.

3. (핵심) 얼굴을 숫자로 변환하는 함수

이 스크립트의 심장부입니다. **얼굴 이미지를 AI가 이해할 수 있는 벡터(숫자 배열)로 바꿉니다.**

Python

```
# --- 5. 🚨 중복을 제거한 올바른 임베딩 추출 함수 (PyTorch 버전) ---
def extract_adaface_embedding(img_bgr, face_kps):
    # 1. 눈, 코 위치(face_kps)를 기준으로 고개를 똑바로 맞추고 112x112 크기로 자름
    aligned_face = face_align.norm_crop(img_bgr, landmark=face_kps, image_size=112)
    # 2. 이미지를 모델이 좋아하는 형태(텐서)로 변환
    img_tensor = torch.tensor(img_norm.transpose(2, 0, 1)).float().unsqueeze(0).to(de

    # 3. AdaFace 모델에 이미지를 통과시켜 고유 벡터(embedding) 추출
    with torch.no_grad():
        embedding, _ = adaface_model(img_tensor)

    return embedding.cpu().numpy()[0]
```

4. 데이터 뻥튀기 (Data Augmentation) 설정

원본 사진 한 장만 쓰면 아쉬우니, **각도나 빛을 조금씩 바꿔 여러 장인 것처럼** 만듭니다.

Python

```
# --- 6. 증강 및 전체 프로세싱 ---
augment = A.Compose([
    A.HorizontalFlip(p=0.5), # 50% 확률로 좌우 반전
    A.RandomBrightnessContrast(p=0.3), # 30% 확률로 밝기/대비 조절
    A.Rotate(limit=15, p=0.3), # 30% 확률로 최대 15도 회전
])
```

5. 사진 한 장에서 최적의 데이터 뽑아내기

원본과 뻥튀기한 이미지들을 3번의 함수에 넣어 데이터를 뽑고, **그것들의 평균값**을 구합니다.

Python

```
def get_face_embedding(image_path: str, n_augment: int = 5):
    # ... (중략: 이미지 읽어오기 및 원본 특징 추출) ...

    # n_augment(5번) 만큼 이미지를 변형시켜 각각의 특징을 추출
    for i in range(n_augment):
        augmented = augment(image=img_rgb_for_aug)
        # ... 변형된 이미지에서 얼굴 추출 후 augment_embeddings 리스트에 저장 ...

    results['augmentations'] = np.array(augment_embeddings)

    if all_embeddings:
        # 추출된 모든 특징(원본 + 변형 5개)의 '평균(mean)'을 구해서 인식률 극대화
        results['average'] = np.mean(all_embeddings, axis=0)
```

6. 폴더별로 사람 이름표(Label) 달아주기

드라이브에 있는 **폴더들을 순회하면서 데이터를 모읍니다.**

Python

```
def process_folder(base_path):
    # ... (중략) ...
    for person_dir in base_path.iterdir(): # 사람 폴더(예: '홍길동', '이순신') 순회
        label = person_dir.name # 폴더 이름을 그 사람의 Label(정답지)로 사용

        # ... 폴더 내의 이미지들을 돌면서 get_face_embedding() 실행 ...
        # faiss_data 리스트에 {"label": "홍길동", "embedding": [추출된 숫자들...]} 형
```

7. 고속 검색 DB (FAISS) 생성 및 파일 저장

마지막으로, 추출된 수많은 벡터 데이터들을 **나중에 0.1초 만에 검색할 수 있도록 인덱싱하여 저장**합니다.

Python

```
# --- 7. 실행 및 FAISS 저장 ---
if not train_df.empty:
    # 1. 벡터 데이터들을 FAISS가 좋아하는 형태로 깔끔하게 정렬 및 정규화
    embeddings = np.stack(train_df['embedding'].values).astype('float32')
    embeddings /= np.linalg.norm(embeddings, axis=1, keepdims=True)

    # 2. 코사인 유사도(IP) 기반의 FAISS 인덱스(빈 박스) 생성
    index = faiss.IndexFlatIP(embeddings.shape[1])
    # 3. 빈 박스에 추출한 얼굴 벡터 데이터 쑤셔넣기
    index.add(embeddings)

    labels = train_df['label'].tolist() # 이름표(Label) 따로 분리

    # 4. 벡터 박스와 이름표를 구글 드라이브에 각각 파일로 저장!
    faiss.write_index(index, f"{save_path}/face_faiss_index_v2.index")
    with open(f"{save_path}/face_faiss_labels_v2.pkl", "wb") as f:
        pickle.dump(labels, f)
```

이렇게 나누어 보니 이 코드가 데이터를 읽어와서 어떻게 가공하고 저장하는지 연결이 되시나요?

코드 중에서 특별히 문법이 이해가 안 가거나, 수정해서 쓰고 싶은 특정 블록(예: 이미지 증강 횟수를 바꾸고 싶다거나, 다른 모델을 얹고 싶다거나)이 있다면 말씀해 주세요! 집중적으로 파헤쳐 드리겠습니다.