* 인공지능 모델 구축에 인공지능 모듈에서 필수 개념 (shape에러 처리는 기본임)

1. 데이터 전처리 파이프라인 (텍스트를 예로하면)

토큰화	텍스트를 숫자로 변환하는 과정
패딩/자르기	모든 입력을 같은 길이로 맞추기
정규화	데이터 분포 조정 (이미지는 0-1,텍스트는 소문자 변환 등)
훈련/검증/테스트 분할	데이터 누수 방지

2. Dataset과 DataLoader 개념

Dataset 클래스	**__getitem__, __len__ 필수 구현**
배치 처리	개별 처리 vs 배치 처리의 효율성 차이
huffle=True	훈련 시 데이터 순서 섞기
batch_size 조절	메모리와 학습 안정성의 트레이드오프

3. 텐서 Shape 이해

차원별 의미	**(배치, 높이, 너비, 채널) 또는 (배치, 시퀀스, 특성)**
Shape 변화 추적	각 레이어 통과 후 텐서 크기 변화
메모리 계산	Shape × 데이터타입 크기로 메모리 사용량 예측

4. 기본 레이어들의 역할

임베딩	범주형 데이터를 연속 벡터로 변환
선형(Dense)	특성변환 및 분류/회귀 출력
활성화함수	비선형성 추가
드롭아웃	과적합 방지

5. 손실함수와 옵티마이저

CrossEntropyLoss	분류
MSELoss	회귀
Adam	더좋은 옵티마이저가 나올수 있으니, 논문등 참고
학습률	가장 중요한 하이퍼파라미터(학습율,배치,모델크기)

6. 그외

메모리관리	OOM 에러대응 (8페이제 OOM 있음), 메모리정리 GPU 이동 및 병렬GPU사용법
과적합 감지와 대응	훈련/검증 손실모니터링, 조기종료, 정규화기법
모델저장과로드	체크포인트, 최고성능모델보존, 재현가능성

* CNN모델에서의 전처리 필수 개념

1. 이미지 정규화(Normalization)

큰 숫자는 그래디언트 폭발, 작은 숫자는 소실 유발

```
# 픽셀 값 범위 조정 (0-255 → 0-1)
transform = transforms.Compose([
    transforms.ToTensor(),  # [0, 255] → [0, 1]
    transforms.Normalize(mean=[0.485, 0.456, 0.406],  # ImageNet 평균
                         std=[0.229, 0.224, 0.225])    # ImageNet 표준편차
])
```

2. 이미지 크기 통일 (Resize/Crop)

CNN 필수 이유: 배치 처리를 위해 모든 이미지가 같은 크기여야 함

```
# 모든 이미지를 같은 크기로
transforms.Resize((224, 224))  # 강제 리사이즈 (비율 깨질 수 있음)
transforms.CenterCrop(224)     # 중앙 잘라내기
transforms.RandomCrop(224)     # 랜덤 위치에서 잘라내기
```

3. 데이터 증강

데이터 부족 해결, 모델 일반화 성능 향상

```
train_transform = transforms.Compose([
    transforms.RandomHorizontalFlip(p=0.5),   # 50% 확률로 좌우 반전
    transforms.RandomRotation(10),            # ±10도 회전
    transforms.ColorJitter(brightness=0.2,    # 밝기 조정
                           contrast=0.2,      # 대비 조정
                           saturation=0.2),   # 채도 조정
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5], std=[0.5])
])
```

4. 채널 순서와 형식 이해)

```
# PIL Image: (Height, Width, Channels) - HWC
# PyTorch: (Channels, Height, Width) - CHW
# 배치: (Batch, Channels, Height, Width) - BCHW

# 변환 과정
PIL_image.shape       # (224, 224, 3)
tensor = ToTensor()(PIL_image)
tensor.shape          # (3, 224, 224)
batch_tensor.shape    # (32, 3, 224, 224)
```

5.이미지 형식 통일

```
# 다양한 이미지 형식 처리
def preprocess_image(image_path):
    image = Image.open(image_path)

    # RGBA → RGB 변환
    if image.mode == 'RGBA':
        image = image.convert('RGB')

    # 그레이스케일 → RGB 변환
    if image.mode == 'L':
        image = image.convert('RGB')

    return image
```

* CNN모델에서의 전처리 필수 개념

→ 극단적인 이미지 리사이즈시 생기는 문제점 알기
예) 텍스트 인식: 글자가 찌그러져서 읽기 불가, 자동차 분류: 세단이 SUV처럼 보임, 의료 영상: 장기 모양 왜곡으로 오진 가능

- 다운샘플링 시 세부 정보 손실 (한번에 이미지를 줄이지 않고, 10%씩 점진적으로 이미지를 줄이기도 함)
  - 얼굴 인식: 1024×1024 → 32×32로 줄이면 눈, 코, 입 구분 불가
  - 의료 영상: CT 스캔을 과도하게 축소하면 종양 위치 판별 불가
  - 앨리어싱 (Aliasing) 현상으로 인한 고주파가 저주파로 보이거나, 크무늬 패턴이 이상한 줄무늬로 보임

- 비율 왜곡 (Aspect Ratio Distortion)
  - 원본: 세로 직사각형 (600×800)을 타겟: 정사각형 (224×224) 하면 결과: 사람이 뚱뚱해 보임, 건물이 찌그러짐
  - 해결책: 비율 유지

```
transforms.Compos ([ transforms.Resize(256),            # 짧은 쪽을 256으로
                     transforms.CenterCrop(224)         # 중앙에서 정사각형 자르기])
```

→ CNN모델(컴퓨터비전모델) 현업적응기
https://www.ksam.co.kr/p_base.php?action=story_base_view&no=3143&s_category=_3_11_

비교적 높은 정확도의 모델을 구축했으므로 모델을 검사 라인의 카메라에 통합하여 검사 자동화를 진행할 수 있다고 생각되지만, 오분류로 인한 피해 또한 고려하여 산업 현장 적용할지에 대한 검토가 필요하다. 산업 현장에서 CNN을 활용한 이미지 품질 검사 시 고려해야 할 사항은 다음과 같다.

① 데이터 수집 및 라벨링: 다양한 결함 유형을 포함한 충분한 이미지 데이터 확보 및 정확한 라벨링
② 모델의 정확도 및 신뢰성: 오분류를 최소화할 수 있는 고성능 모델 필요
③ 환경: 조명, 각도, 거리 등의 환경적 변화 최소화
④ 실시간 처리 능력: 산업 현장의 빠른 검사 속도에 맞춰 모델의 처리 속도 최적화가 중요
⑤ 모델 유지 및 업데이트: 새로운 결함 유형이나 공정 변화에 맞춰 지속적인 모델의 업데이트 필요

* CNN관련 참고자료

→ CNN이 어떻게 이미지를 보는가?

https://distill.pub/2017/feature-visualization/

→ 데이터부족시 합성데이터 생성방법(구글검색)

Learning to Generate Synthetic Training Data

→ 벤치마크 및 실험 데이터-표준 전처리 방법과 벤치마크 결과

https://www.image-net.org/

→ Papers With Code Leaderboards -최신 모델들의 전처리 설정 비교

https://paperswithcode.com/sota/image-classification-on-imagenet

→ imgaug Library-다양한 증강기법과 시각화도구

https://imgaug.readthedocs.io/

→ 실용 도구 및 라이브러리 문서

→ 엔디비아 GPU 가속 데이터 로딩과 전처리
https://docs.nvidia.com/deeplearning/dali/

→ OpenCV 공식 튜토리(이미지 변환 알고리즘의 수학적 배경
https://docs.opencv.org/4.x/d2/d96/tutorial_py_table_of_contents_imgproc.html

→ PyTorch 기반 컴퓨터 비전 라이브러리
https://kornia.github.io/

이 미션코드를 통한 중점 학습사항

1. 데이터 전처리 파이프라인

- 토큰화: 텍스트를 숫자로 변환하는 과정
- 패딩/자르기: 모든 입력을 같은 길이로 맞추기
- 정규화: 데이터 분포 조정 (이미지는 0-1, 텍스트는 소문자 변환 등)
- 훈련/검증/테스트 분할: 데이터 누수 방지

2. Dataset과 DataLoader 개념

- Dataset 클래스: __getitem__, __len__ 필수 구현
- 배치 처리: 개별 처리 vs 배치 처리의 효율성 차이
- shuffle=True: 훈련 시 데이터 순서 섞기
- batch_size 조절: 메모리와 학습 안정성의 트레이드오프

- 미션코드에서 추후 RAG를 위해 반드시 알아야할 핵심내용-

- **텍스트를 의미있는 벡터로 변환하는 방법**
  - RAG등 현재는 Sentence-Bert, OpenAi Embeddings, Hugging Face등을 통한 토큰화와 벡터화를 진행함
  - 이 코드의 Word2Vec, FastText, Glove로 텍스트를 벡터로 변환 기법들은
    - 텍스트가 어떻게 숫자가 되는지에 대한 [원리이해]
    - 임베딩이 이상할때 뭐가 문제인지를 알수 있는 [문제해결]
    - 특수한 도메인(분야)에서 직접 임베딩 학습할때 [커스터마이징]

  - **이 코드의** 임베딩 행렬 구축도 RAG에서는 Sentence-Bert(행렬구축다함), 벡터DB에 바로 저장, OpenAI모듈등 활용함
    - [제공코드] word2vec_matrix = np.zeros((len(word2vec_model.wv) + 1, 128))
    - RAG의 벡터 데이터베이스와 동일한 개념일뿐 실제 RAG에서는 자동화가 되어 있는 모듈을 사용함
      문서넣으면 ->자동으로 벡터변환 -> 자동으로 db저장
      검색할때 ->쿼리벡터변환 ->유사도검색 ->결과 리턴함
    - 이 작업을 통해서는
      - 벡터db가 내부적으로 뭘 하는지 이해, 임베딩 모델을 바꾸거나 커스터마이징 할때 필요함

- **OOV 처리 이해 -> 요즘은 서브워드(BPE)토크나이저(모듈)가 있어서, OOV걱정없음**
  - [제공코드] encoded = [self.word2idx.get(word, 0) for word in tokens]

서브워드

```
# 단어를 의미있는 조각으로 나누기
"programming" → ["program", "##ming"] # 2개 토큰
"programmer" → ["program", "##mer"] # 2개 토큰 (program 공유!)
"programmed" → ["program", "##med"] # 새로운 단어도 처리 가능! #

## 표시는 "이전 토큰의 연속"이라는 뜻
```

- 미션코드에서 추후 RAG를 위해 반드시 알아야할 핵심내용-

- **텍스트 전처리 파이프라인 (아주중요, 검색품질에 매우 끈 영향을 미침)**
  - RAG에서도 문서 인덱싱할 때 동일한 전처리 필요
  - RAG에서는 -> 의미보존(문장/문단 경계고려) , 중복제거(같은 내용 여러 번 인덱싱방지),
    언어별처리(한국어 형태소분석:좋은 형태소분석기 찾아야함, 영어처리법)
  - RAG에서는 단어정제 보다는 문서구조화에 더 집중해야함.
  - RAG를 위한 텍스트 전처리에 집중하고자 한다면
    - 문서파싱 중요함(PDF자료 불러오기, 이미지내의텍스트자료꺼내기, HTML문서불러오기, 웹사이트문서가져오기, 오디오자료텍스트로변환하기등)
- 현재 이 코드는 고정길이로 자르게 되어 있고 (max_len) 모델에 넣으려면 고정길이 해야함.
  - 고정길이문제점: 정보손실 / 비효율성 / 메모리 낭비
- 하지만 RAG에서는 문서별 가변길이 가능:
  - 개별 임베딩을, Transformer의 어텐션마스킹(패딩부분무시), 동적배치(비슷한길이끼리 자동 배치묶음)

제일 중요한건 어떻게 문장을 자르냐(청킹)가 제일 큰 관건임 → 의미 보존하며 나누기등 다양한 전략을 사용함

- **지금 코드에서 반드시 고정길이 해야하는 이유**

현재 코드의 패딩/자르기

```
if len(encoded) < self.max_len:
    encoded += [0] * (self.max_len - len(encoded))  # 패딩
else:
    encoded = encoded[:self.max_len]  # 자르기
```

LSTM 배치 처리

```
# LSTM은 배치의 모든 시퀀스가 같은 길이여야 함
lstm_input.shape = (batch_size, seq_len, embedding_dim)
#                                  ↑ 모든 샘플이 같아야 함
```

텐서 연산 제약

```
# PyTorch 텐서는 모든 차원이 일정해야 함
batch = torch.stack([seq1, seq2, seq3])  # 모두 같은 길이
```

- 미션코드에서 추후 RAG를 위해 반드시 알아야할 핵심내용-

- **배치 처리 (모델작성과 튜닝에 많은 영향 미침) -메모리효율성, 연산효율성, 학습안정성등을 고려해야함**
  - 대량의 텍스트를 효율적으로 벡터화하는 방법
  - RAG에서 대규모 문서를 임베딩할 때 필수(벡터DB재구축해야하는경우)

*필수-> 구글 검색: 인공지능모델에서의 OOM
LLM에서의 OOM

(참고코드)

```
# 너무 작은 배치 (batch_size=4)
# 장점: 메모리 적게 사용, 더 자주 업데이트
# 단점: GPU 활용도 낮음, 불안정한 그래디언트

# 너무 큰 배치 (batch_size=512)
# 장점: GPU 활용도 높음, 안정적인 그래디언트
# 단점: 메모리 많이 사용, 수렴 느림

# 적절한 배치 크기 찾기
def find_optimal_batch_size(model, dataset):
    for batch_size in [16, 32, 64, 128, 256]:
        try:
            loader = DataLoader(dataset, batch_size=batch_size)
            batch = next(iter(loader))

            # GPU 메모리 체크
            texts, labels = batch[0].to(device), batch[1].to(device)
            output = model(texts)
            loss = criterion(output, labels)
            loss.backward()

            print(f"Batch size {batch_size}: OK")

        except RuntimeError as e:
            if "out of memory" in str(e):
                print(f"Batch size {batch_size}: OOM")
                break
```

```
# 배치 크기 모니터링
class BatchMonitor:
    def __init__(self):
        self.memory_usage = []
        self.batch_sizes = []

    def log_batch(self, batch):
        batch_size = batch[0].shape[0]
        if torch.cuda.is_available():
            memory = torch.cuda.memory_allocated() / 1024**2
            self.memory_usage.append(memory)
            self.batch_sizes.append(batch_size)
            print(f"Batch size: {batch_size}, GPU memory:
{memory:.1f}MB")

# 사용
monitor = BatchMonitor()
for batch in train_loader:
    monitor.log_batch(batch)
    # 모델 처리...
```

- 미션코드에서 추후 튜닝(모델수정)작업이 주라면 꼭 알야야 하는 부분

-하이퍼 파라미터

```
# 임베딩 관련
vector_size=128, window=5, min_count=1, sg=1  # Word2Vec/FastText
embedding_dim = 200  # GloVe

# 모델 구조
hidden_dim = 128, num_layers=2, dropout=0.5  # LSTM
max_len = 280  # 시퀀스 길이

# 학습 관련
lr=0.005, batch_size=64, epochs=10
```

-임베딩

```
self.embedding = nn.Embedding.from_pretrained(..., freeze=False)
# freeze=False: 사전훈련된 임베딩도 같이 학습
# freeze=True: 임베딩 고정하고 분류기만 학습
```

-옵티마이저

```
optimizer = torch.optim.Adam(model.parameters(), lr=0.005)
# Adam, SGD, AdamW 등 선택
# 학습률 스케줄링 추가 가능
```

-아키텍처

```
# LSTM 설정
bidirectional=True  # 양방향 여부
num_layers=2        # 레이어 수
dropout=0.5         # 드롭아웃 비율
```

-데이터 전처리

```
# 텍스트 길이 분석 후 결정
max_len = 280  # 95분위수 기준으로 설정
# 너무 짧으면 정보 손실, 너무 길면 연산 비용 증가
```

- 배치크기

```
batch_size=64  # Word2Vec/FastText
batch_size=32  # GloVe (메모리 절약)
```

- 조기 종료 (코드에는없음)
  . Validation loss 모니터링
  .patience 설정

Part. 미션 자료 모델에 대한 간단 설명
- 빨간색이 추가된 설명임

→ 모델작성시 Shape 기반 단계적 최적화

. **메모리 예측**: Shape로 메모리 사용량 미리 계산
. **병목 파악**: 어느 레이어가 가장 느린지 Shape로 판단
. **하이퍼파라미터 제약**: 실현 가능한 설정 범위 결정
. **최적화 방향**: 어떤 차원을 줄일지 우선순위 설정

```
def optimize_model_shapes():
    # 1단계: 메모리 한계 찾기
    max_batch = find_max_batch_size()

    # 2단계: 병목 레이어 파악
    bottleneck = profile_layers()  # LSTM이 병목

    # 3단계: 병목 레이어 최적화
    if bottleneck == 'LSTM':
        # hidden_dim 줄이기 시도
        for hid_dim in [128, 96, 64]:
            acc = test_hidden_dim(hid_dim)
            if acc > threshold:
                best_hidden_dim = hid_dim
                break

    # 4단계: 전체 조합 최적화
    best_config = grid_search(
        batch_sizes=[max_batch//2, max_batch//4],
        hidden_dims=[best_hidden_dim, best_hidden_dim//2],
        seq_lens=[280, 200, 140]
    )

    return best_config
```

각 함수들
8월8일 오전에 별도로 자료제공함.

성능에 영향을 주는 Shape 요소들

1. **max_len**: 길수록 메모리 및 연산량 증가
2. **embedding_dim**: 표현력과 메모리 사용량의 트레이드오프
3. **hidden_dim**: LSTM 용량과 연산 복잡도
4. **batch_size**: 메모리 사용량과 학습 안정성
5. **bidirectional**: 2배 메모리, 더 나은 성능

임베딩 행렬 구성의 상세 분석

```
# Word2Vec 예시
word2vec_model = Word2Vec(sentences=train_sentences, vector_size=128, ...)
vocab_size = len(word2vec_model.wv)  # 예: 50000

# 임베딩 행렬 생성
word2vec_matrix = np.zeros((vocab_size + 1, 128))  # +1은 패딩/OOV용 (인덱스 0)

word2idx_word2vec = {word: idx + 1 for idx, word in enumerate(word2vec_model.wv.index_to_key)}
# 인덱스 0은 예약, 실제 단어는 1부터 시작

for word, idx in word2idx_word2vec.items():
    word2vec_matrix[idx] = word2vec_model.wv[word]  # 벡터 할당

# 최종 행렬 구조:
# word2vec_matrix[0] = [0, 0, 0, ..., 0]      # 패딩용 제로 벡터
# word2vec_matrix[1] = [0.1, -0.3, 0.7, ...]  # 첫 번째 단어 벡터
# word2vec_matrix[2] = [0.5, 0.2, -0.1, ...]  # 두 번째 단어 벡터
# ...
```

DataLoader에서의 배치 처리

```
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)

for batch_idx, (texts, labels) in enumerate(train_loader):
    # texts.shape = (64, 280)  # 64개 샘플, 각각 280 토큰
    # labels.shape = (64,)     # 64개 레이블

    # GPU로 이동
    texts = texts.to(device)  # (64, 280) on CUDA
    labels = labels.to(device)  # (64,) on CUDA

    # 모델 forward
    outputs = model(texts)  # (64, 20) - 20개 클래스 확률

    # 손실 계산
    loss = criterion(outputs, labels)  # CrossEntropyLoss
    # outputs: (64, 20) - logits
    # labels: (64,) - 정수 인덱스 (0~19)
```

EmbeddingLSTM 모델의 Shape 변화

```
class EmbeddingLSTM(nn.Module):
    def __init__(self, embedding_matrix, hidden_dim, output_dim, num_layers=2, dropout=0.5):
        super(EmbeddingLSTM, self).__init__()

        # 임베딩 행렬 정보
        num_embeddings, embedding_dim = embedding_matrix.shape
        # Word2Vec: (vocab_size + 1, 128)
        # FastText: (vocab_size + 1, 128)
        # GloVe: (400001, 200)

        self.embedding = nn.Embedding.from_pretrained(
            torch.tensor(embedding_matrix, dtype=torch.float).to(device),
            freeze=False
        )

        self.lstm = nn.LSTM(
            input_size=embedding_dim,     # 128 (Word2Vec/FastText) 또는 200 (GloVe)
            hidden_size=hidden_dim,       # 128
            num_layers=num_layers,        # 2
            batch_first=True,
            dropout=dropout,              # 0.5
            bidirectional=True            # 이 부분이 중요!
        )

        #  현재 코드의 문제점
        self.fc = nn.Linear(hidden_dim, output_dim)  #? 여기 shape 꼭 확인해보세요
```

Forward Pass에서의 Shape 변화

```
def forward(self, x):
    # 입력: x.shape = (batch_size, max_len) = (64, 280)

    # 1. 임베딩 레이어
    embedded = self.embedding(x)
    # embedded.shape = (batch_size, max_len, embedding_dim)
    # Word2Vec/FastText: (64, 280, 128)
    # GloVe: (64, 280, 200)

    # 2. LSTM 레이어
    output, (hidden, cell) = self.lstm(embedded)

    # LSTM 출력 분석:
    # output.shape = (batch_size, max_len, hidden_dim * 2)  # bidirectional이므로 *2
    # Word2Vec/FastText: (64, 280, 256)
    # GloVe: (64, 280, 256)

    # hidden.shape = (num_layers * 2, batch_size, hidden_dim)  # bidirectional이므로 *2
    # (4, 64, 128) - 2개 레이어 × 2방향

    # cell.shape = (num_layers * 2, batch_size, hidden_dim)
    # (4, 64, 128)

    # 3. 마지막 hidden state 선택
    last_hidden = hidden[-1]  # 마지막 레이어의 forward 방향만
    # last_hidden.shape = (batch_size, hidden_dim) = (64, 128)

    # 현재 코드의 문제: bidirectional의 backward 방향 무시됨!

    # 4. 완전연결층
    output = self.fc(last_hidden)
    # output.shape = (batch_size, output_dim) = (64, 20)

    return output
```

GloVe의 특별한 경우

```
# GloVe 6B.200d의 경우
GLOVE_FILE = "glove.6B.200d.txt"
embedding_dim = 200
vocab_size = 400000  # 사전 훈련된 단어 수

glove_embeddings = {}  # 딕셔너리로 로드
with open(GLOVE_FILE, 'r', encoding='utf-8') as f:
    for line in f:
        values = line.split()
        word = values[0]
        coeffs = np.asarray(values[1:], dtype='float32')  # 200차원 벡터
        glove_embeddings[word] = coeffs

# 행렬 구성
word2idx_glove = {word: idx + 1 for idx, word in enumerate(glove_embeddings.keys())}
glove_matrix = np.zeros((len(word2idx_glove) + 1, 200))
# shape: (400001, 200)

# 메모리 사용량: 400001 * 200 * 4 bytes = 약 320MB
```