텍스트, 로고, 폰트, 상징이(가) 표시된 사진

자동 생성된 설명

로고, 폰트, 그래픽, 상징이(가) 표시된 사진

자동 생성된 설명

ICE4104-001 인공지능 응용

**CNN과 ResNet50을 이용한 독초 이진 분류 판별기**

**정통 AI (김동한, 김재민, 김지훈, 최시원)**

최종보고서

보고서 작성 서약서

1. 나는 타학생의 보고서를 베끼거나 여러 보고서의 내용을 짜집기하지 않겠습니다.

2. 나는 보고서의 주요 내용을 인터넷사이트 등을 통해 얻지 않겠습니다.

3. 나는 보고서의 내용을 조작하지 않겠습니다.

4. 나는 보고서 작성에 참고한 문헌의 출처를 밝히겠습니다.

5. 나는 나의 보고서를 제출 전에 타학생에게 보여주지 않겠습니다.

나는 보고서 작성시 윤리에 어긋난 행동을 하지 않고 정보통신공학인으로서 나의 명예를 지킬 것을 맹세합니다.

2024년 6 월 6일

학부 정보통신공학

학년 4

성명 김지훈

학번 12191748

목차

**서론4**

1. 문제 정의 및 연구 배경 4  
가)연구 배경4  
나)문제 정의4  
다)연구 목적5

**이론적 배경7**

1. CNN(Convolutional Neural Network)의 원리7

2. ResNet50(Residual Network)의 원리8

3. 독초 및 비독초 판별의 필요성9

**데이터셋 설명11**

1. 데이터셋 출처 및 구성11
2. 데이터 전처리 과정12
3. 라벨링 기준 및 방법14

모델 설계 및 구현**16**

1. CNN 모델 설계16
2. ResNet50 모델 설계17
3. 모델 학습 환경 및 도구18
4. 하이퍼파라미터 설정19

모델 성능 평가**21**

1. 성능 평가 지표 설명21
2. 모델 학습 및 평가 결과22
3. 성능 비교: CNN vs ResNet5024

결과 분석 및 논의**26**

1. 각 모델의 성능 분석26
2. 결과에 대한 논의28
3. 모델의 한계점 및 개선 방안29

참고문헌**31**

1. 참고한 논문 및 자료31

부록**32**

1. 코드 및 추가 자료32

다) <layer added CNN 코드>32

다) <ResNet50 코드>41

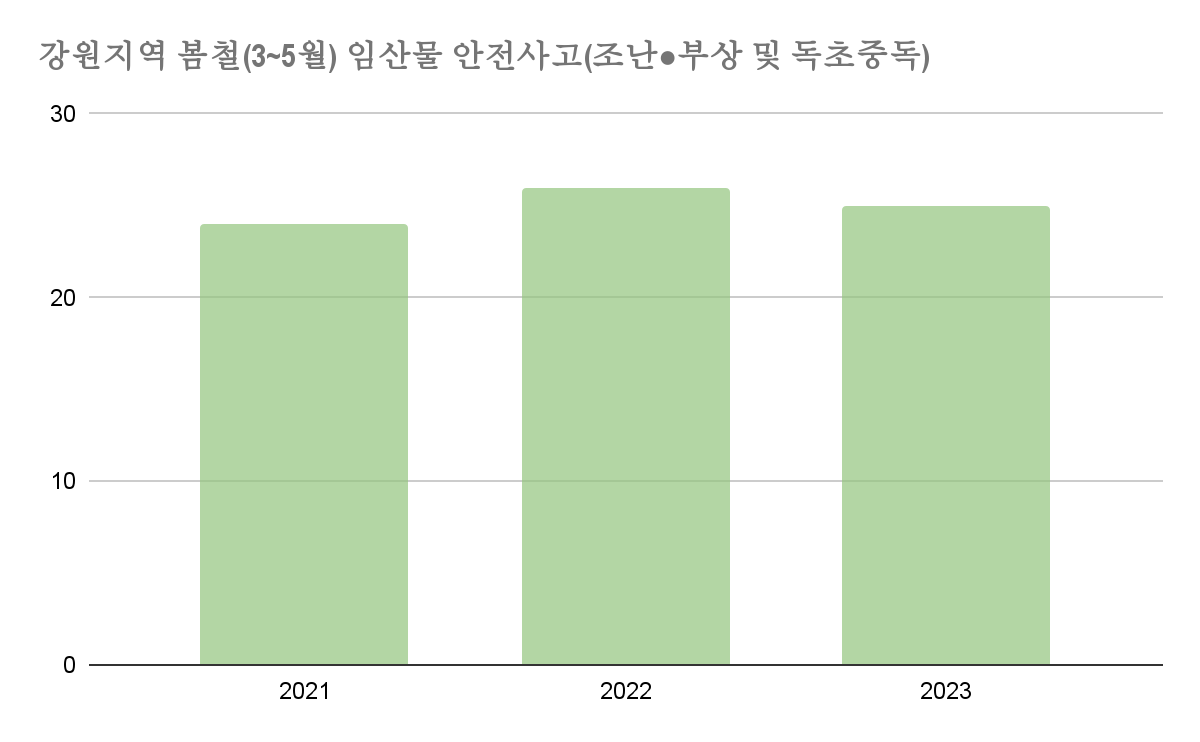
다) <ResNet50 코드 해석과 함수 동작 원리>48

**서론**

**1.문제 정의 및 연구 배경**

**가)연구 배경**

최근 몇 년 동안, 자연에서 얻을 수 있는 다양한 식물들에 대한 관심이 높아지면서 약초와 식용 식물을 찾는 사람들이 늘어나고 있습니다. 그러나 이러한 식물들 중 일부는 독성을 지니고 있어 잘못 섭취할 경우 심각한 건강 문제를 일으킬 수 있습니다. 예를 들어, 중앙일보는 2021년 4월에 강원도에서 산삼으로 오인하여 독초를 섭취한 70~80대 노인이 구토와 설사 등의 중독 증상을 호소한 사건을 보도한 바 있습니다. 이처럼 독초 섭취로 인한 사고는 예기치 않게 발생할 수 있으며, 특히 노인층이나 식물에 대한 지식이 부족한 사람들에게 큰 위험을 초래할 수 있습니다.



강원도 소방본부에 따르면, 강원 지역에서 봄철 독초 중독 사고는 2021년 이후 매년 20건 이상 꾸준히 발생하고 있습니다. 또한, 식품의약품안전처의 소비자 위해감지시스템(CISS) 데이터에 따르면, 2020년부터 2022년까지 총 26건의 독초 섭취 사고가 보고되었습니다. 이러한 사고는 독초와 무독성 식물이 외관상 매우 유사하여 일반인이 이를 구별하기 어렵기 때문에 발생합니다. 예를 들어, 독성을 가진 참취와 독성이 없는 곰취는 외관상 매우 유사하여, 일반인이 이를 구별하는 것은 상당히 어렵습니다.

독초 섭취 사고를 줄이기 위해서는 독초와 무독성 식물을 정확하게 구별할 수 있는 기술적 방법이 필요합니다. 전통적으로는 식물학적 지식과 경험에 의존하여 독초를 식별했으나, 이는 시간이 많이 걸리고 전문가의 도움 없이 일반인이 활용하기에는 한계가 있습니다. 따라서, 현대 기술을 활용하여 독초를 정확하게 식별할 수 있는 시스템을 개발하는 것이 필요합니다.

**나)문제 정의**

**스크린샷, 만화 영화, 폰트, 클립아트이(가) 표시된 사진

자동 생성된 설명**

본 연구는 인공지능(AI) 기술을 활용하여 독초와 무독성 식물을 구별하는 모델을 개발하는 것을 목표로 했습니다. 특히, Convolutional Neural Networks(CNN)와 ResNet50 모델을 사용하여 이미지를 기반으로 독초를 분류하는 시스템을 구축하고자 합니다. CNN과 ResNet50은 이미지 인식 분야에서 높은 성능을 보이는 딥러닝 모델로, 복잡한 패턴 인식과 분류에 탁월한 능력을 가지고 있습니다.

우리의 연구 목표는 다음과 같습니다. 독초와 무독성 식물의 정확한 분류, CNN과 ResNet50 모델을 사용하여 독초와 무독성 식물 이미지를 학습시키고, 새로운 이미지에 대해 정확한 분류를 수행하는 모델을 개발합니다. 성능 비교 및 향상, CNN과 ResNet50 모델의 성능을 비교 분석하고, 이를 기반으로 최적의 모델을 선택하여 성능을 개선합니다. 이를 위해 다양한 데이터 증강 기법과 하이퍼파라미터 튜닝을 적용할 것입니다. 실시간 사용 가능한 시스템 개발, 최종적으로, 사용자가 사진을 찍어 업로드하면 해당 식물이 독초인지 아닌지를 실시간으로 판단할 수 있는 웹 애플리케이션을 개발합니다. 이는 특히 야외 활동 중 독초를 식별하는 데 유용할 것입니다.

**다)연구 목적**

딥러닝 기술을 활용하여 독초와 비독초를 분류하는 인공지능 모델을 개발하는 연구 목적은 다음과 같이 구체화할 수 있습니다.

첫째, 딥러닝 모델을 통해 독초와 비독초를 정확하게 분류할 수 있는 모델을 개발하는 것입니다. 이는 독초 섭취로 인한 안전사고를 예방하고, 일반 대중과 전문가가 독초와 비독초를 쉽게 구별할 수 있도록 돕는 것을 목표로 합니다. 특히, 독초 섭취로 인한 중독사고를 사전에 방지하고, 신속한 대응을 가능하게 하여 공공 안전을 증진시키는 데 중점을 둡니다.

둘째, 다양한 딥러닝 모델을 비교하여 최적의 모델을 선정하는 것입니다. CNN과 ResNet50 같은 최신 모델을 사용하여 독초와 비독초 분류의 정확도와 효율성을 평가하고, 최적의 모델을 도출합니다. 이를 통해 딥러닝 기술의 다양한 활용 가능성을 탐색하고, 특정 문제에 가장 적합한 모델을 찾는 과정을 거치게 됩니다.

셋째, 연구에서 사용된 데이터셋을 체계적으로 구축하고 이를 통해 모델의 성능을 평가하는 것입니다. AI Hub에서 제공하는 동의보감 독초 판별 이미지 데이터를 활용하여, 독성과 비독성을 가진 다양한 식물의 이미지를 수집하고 정리합니다. 이 데이터셋을 통해 모델을 학습시키고, 검증 및 테스트 과정을 거쳐 모델의 성능을 객관적으로 평가합니다.

넷째, 독초와 비독초를 분류하는 모델의 실제 응용 가능성을 탐색하는 것입니다. 개발된 모델은 다양한 분야에서 응용될 수 있습니다. 예를 들어, 농업 분야에서는 독초와 유익한 식물을 구분하여 작물의 품질을 높이는 데 활용할 수 있습니다. 또한, 환경보호 분야에서는 독초의 확산을 막고 생태계를 보호하는 데 기여할 수 있습니다. 이러한 응용 가능성은 연구 결과의 실질적인 가치를 높이고, 다양한 산업 분야에 긍정적인 영향을 미칠 것입니다.

다섯째, 이번 연구는 교육 및 학습 자료로 활용될 수 있습니다. 대학이나 연구기관에서 딥러닝 및 이미지 분류 기술을 가르치는 과정에서 이번 연구를 사례로 제시할 수 있습니다. 학생들은 실제 데이터를 다루고, 딥러닝 모델을 적용하는 과정을 통해 실질적인 학습 경험을 쌓을 수 있을 것입니다. 또한, 이번 연구에서 사용된 데이터셋과 모델은 교육용 자료로 공개되어, 다양한 교육기관에서 활용될 수 있습니다.

여섯째, 연구의 한계를 인식하고 이를 극복하기 위한 미래 연구 방향을 제시하는 것입니다. 예를 들어, 데이터셋의 다양성과 양을 늘리는 방법, 더 나은 모델 구조를 개발하는 방법, 그리고 실제 응용 환경에서의 성능을 높이기 위한 방법 등을 모색할 것입니다. 이러한 연구는 독초 분류 모델의 지속적인 발전을 도모하고, 향후 관련 연구에 중요한 기초 자료로 활용될 수 있을 것입니다.

**이론적 배경**

**1.CNN(Convolutional Neural Network)의 원리**

Convolutional Neural Network(CNN)은 컴퓨터 비전 분야에서 이미지 인식, 분류, 객체 탐지 등에 널리 사용되는 인공 신경망의 한 종류입니다. CNN은 인간의 시각 피질이 작동하는 방식을 모방하여 설계되었습니다. 이 섹션에서는 CNN의 기본 원리와 그 구성 요소에 대해 자세히 설명하겠습니다. CNN의 핵심 개념은 '합성곱(convolution)'입니다. 합성곱 연산은 이미지의 특정 특징을 추출하는 역할을 합니다. 이는 필터라고 불리는 작은 커널을 사용하여 이미지의 부분 부분을 훑으면서 이루어집니다. 필터는 이미지의 픽셀 값을 곱하고 더하여 새로운 이미지를 생성합니다. 이 과정은 입력 이미지에서 중요한 특징을 추출하는 데 사용됩니다. CNN의 주요 구성 요소는 다음과 같습니다.

**가) 합성곱층(Convolutional Layer)**

합성곱층은 입력 이미지에 필터를 적용하여 특징 맵(feature map)을 생성합니다. 이때 필터의 크기, 스트라이드(stride), 패딩(padding) 등의 하이퍼파라미터가 중요합니다. 필터는 학습 과정에서 최적화되며, 이미지의 가장자리, 색상 변화 등과 같은 로컬 특징을 감지합니다.

**나) 활성화 함수(Activation Function)**

활성화 함수는 비선형성을 추가하여 신경망이 복잡한 패턴을 학습할 수 있게 합니다. 일반적으로 ReLU(Rectified Linear Unit) 함수가 사용됩니다. ReLU는 음수 값을 0으로 변환하고, 양수 값은 그대로 유지합니다. 이는 계산 효율성을 높이고, 기울기 소실 문제를 완화하는 데 도움을 줍니다.

**다) 폴링층(Pooling Layer)**

폴링층은 입력 특징 맵의 크기를 줄여 계산량을 감소시키고, 중요한 특징을 유지하면서도 위치에 대한 불변성을 제공합니다. 가장 일반적인 폴링 방식은 최대 폴링(max pooling)과 평균 폴링(average pooling)입니다. 최대 폴링은 필터 내의 최대 값을 선택하고, 평균 폴링은 필터 내의 평균 값을 선택합니다.

**라) 완전 연결층(Fully Connected Layer)**

완전 연결층은 입력을 하나의 벡터로 펼친 후, 전통적인 인공 신경망의 방식으로 분류를 수행합니다. 이 층은 이전 층에서 추출한 특징을 바탕으로 최종 예측을 수행합니다. 이미지 분류 문제에서는 마지막 완전 연결층에서 소프트맥스(softmax) 함수를 사용하여 각 클래스에 대한 확률을 출력합니다.

CNN의 학습 과정은 역전파 알고리즘(backpropagation)을 통해 이루어집니다. 역전파 알고리즘은 출력층에서 발생한 오차를 신경망의 각 층으로 전파하며 가중치를 조정합니다. 이를 통해 신경망은 점진적으로 최적의 가중치를 학습하게 됩니다. CNN의 장점 중 하나는 파라미터 공유(parameter sharing)와 스파스 연결(sparse connectivity)입니다. 파라미터 공유는 동일한 필터가 이미지 전체에 걸쳐 적용된다는 의미로, 이는 학습해야 할 파라미터의 수를 줄여줍니다. 스파스 연결은 각 뉴런이 입력 이미지의 일부에만 연결되어 있어 계산 효율성을 높입니다.

**2.ResNet50(Residual Network)의 원리**

ResNet(Residual Network)은 2015년 마이크로소프트 리서치 팀에 의해 개발된 딥러닝 모델로, 깊은 신경망을 효과적으로 학습할 수 있는 방법을 제시한 혁신적인 네트워크입니다. ResNet의 핵심 아이디어는 "잔차 학습(residual learning)"으로, 이는 네트워크가 직접 원하는 매핑을 학습하는 대신, 잔차를 학습하도록 하는 것입니다. 이 섹션에서는 ResNet50의 원리와 그 구성 요소에 대해 자세히 설명하겠습니다. 딥러닝 모델이 깊어질수록 학습이 어려워지는 문제를 "기울기 소실(vanishing gradient)" 문제라고 합니다. 이는 네트워크의 깊이가 깊어질수록 역전파 과정에서 기울기가 점점 작아져, 초기 층으로 전파될 때 거의 사라지기 때문입니다. ResNet은 이러한 문제를 해결하기 위해 잔차 블록(residual block)을 도입하였습니다. 잔차 블록의 기본 구조는 다음과 같습니다.

입력(x) -> 합성곱층 -> Batch Normalization -> 활성화 함수(ReLU) -> 합성곱층 -> Batch Normalization -> 출력   
여기서 잔차 블록은 입력을 그대로 출력에 더해주는 "스킵 연결(skip connection)"을 포함합니다. 이는 다음과 같은 수학적 표현으로 나타낼 수 있습니다:

H(x) = F(x) + x

여기서 H(x)는 출력, F(x)는 잔차 함수, x는 입력입니다. 스킵 연결을 통해 입력이 출력으로 직접 전달되므로, 기울기 소실 문제가 완화되고 학습이 더 용이해집니다. ResNet50은 50개의 층을 가진 깊은 네트워크로, 48개의 합성곱층과 2개의 완전 연결층으로 구성됩니다. ResNet50의 주요 구성 요소는 다음과 같습니다.

**가)합성곱층(Convolutional Layer)**

ResNet50은 여러 개의 합성곱층을 사용하여 입력 이미지에서 다양한 수준의 특징을 추출합니다. 초기 층은 기본적인 가장자리, 색상 등의 저수준 특징을 감지하고, 깊은 층으로 갈수록 더욱 복잡한 패턴을 학습합니다.

**나)Batch Normalization**

Batch Normalization은 각 미니배치에 대해 평균과 분산을 조정하여 데이터 분포를 정규화하는 기술입니다. 이는 학습을 안정화시키고, 학습 속도를 높이며, 기울기 소실 문제를 완화하는 데 도움을 줍니다.

**다)활성화 함수(ReLU)**

ResNet50에서도 ReLU(Rectified Linear Unit) 활성화 함수가 사용됩니다. ReLU는 비선형성을 추가하여 신경망이 복잡한 패턴을 학습할 수 있도록 도와줍니다. 또한 계산이 간단하여 학습 속도를 높이는 장점이 있습니다.

**라)스킵 연결(Skip Connection)**

스킵 연결은 ResNet의 핵심 요소로, 잔차 학습을 가능하게 합니다. 입력을 직접 출력으로 전달하여 기울기 소실 문제를 해결하고, 네트워크가 더 깊어지더라도 효과적으로 학습할 수 있도록 합니다. ResNet50의 잔차 블록은 두 가지 형태로 나눌 수 있습니다: 기본 블록(basic block)과 병목 블록(bottleneck block)입니다. 기본 블록은 두 개의 3x3 합성곱층으로 구성되며, 병목 블록은 1x1, 3x3, 1x1 합성곱층으로 구성됩니다. 병목 블록은 파라미터 수를 줄이고, 계산 효율성을 높이기 위해 설계되었습니다. ResNet50의 학습 과정은 다른 신경망과 마찬가지로 역전파 알고리즘을 사용합니다. 그러나 스킵 연결 덕분에 기울기 소실 문제를 완화하여 깊은 네트워크에서도 효과적으로 학습할 수 있습니다. 이는 ResNet이 매우 깊은 네트워크에서도 높은 성능을 보이는 이유 중 하나입니다.

**3.독초 및 비독초 판별의 필요성**

독초와 비독초를 정확히 구분하는 능력은 공공 안전과 보건 측면에서 매우 중요합니다. 잘못된 식물 섭취는 심각한 건강 문제를 일으킬 수 있으며, 최악의 경우 생명까지 위협할 수 있습니다. 특히 산이나 들에서 자주 식물을 채집하거나 자연에서 생활하는 사람들에게 독초 식별 능력은 필수적입니다. 이 섹션에서는 독초 및 비독초 판별의 필요성에 대해 상세히 설명하겠습니다. 독초는 자연에서 쉽게 찾을 수 있는 식물 중 하나로, 특정한 환경 조건에서 번성합니다. 독초의 독성 성분은 다양한 형태로 나타나며, 섭취 시 중독 증상을 일으킬 수 있습니다. 이러한 독성 성분은 식물이 자신을 보호하기 위해 진화 과정에서 발달시킨 것으로, 주로 초식 동물이나 곤충으로부터 자신을 방어하기 위해 생성됩니다. 사람에게는 이러한 독성 성분이 다양한 건강 문제를 일으킬 수 있습니다.

공공 안전 측면에서 독초 식별은 매우 중요합니다. 독초를 잘못 섭취할 경우 심각한 중독 증상을 유발할 수 있으며, 이는 빠른 응급 처치가 필요합니다. 예를 들어, 일부 독초는 강한 신경 독성을 가지고 있어, 섭취 시 신경계에 심각한 손상을 줄 수 있습니다. 이러한 식물들을 정확히 구분하여 사고를 예방하는 것은 매우 중요합니다.

독초와 비독초를 정확히 구분하는 것은 보건 측면에서도 중요합니다. 독초를 잘못 섭취할 경우 심각한 건강 문제를 일으킬 수 있으며, 이는 병원 치료가 필요한 상황으로 이어질 수 있습니다. 특히 어린이와 노약자는 독초에 대한 저항력이 약하기 때문에 더욱 주의가 필요합니다. 독초 식별 능력은 이러한 건강 문제를 예방하고, 공공의 건강을 보호하는 데 큰 역할을 합니다.

자연에서 생활하거나 야생 식물을 채집하는 사람들에게 독초 식별 능력은 필수적입니다. 야생에서 채집한 식물을 음식으로 사용하는 경우, 독초를 잘못 섭취할 가능성이 높아집니다. 예를 들어, 산에서 나물 채집을 즐기는 사람들이 독초를 잘못 식별하여 섭취하는 경우가 종종 발생합니다. 이러한 상황을 예방하기 위해서는 독초와 비독초를 정확히 구분하는 능력이 필요합니다.

독초와 비독초를 구분하는 교육은 매우 중요합니다. 특히 자연에서 생활하는 사람들뿐만 아니라, 일반 대중에게도 독초에 대한 인식을 높이는 것이 필요합니다. 학교나 지역 사회에서 독초 식별 교육을 통해 독초에 대한 지식을 널리 알리고, 이를 통해 독초 섭취 사고를 줄일 수 있습니다. 또한, 독초 식별 앱이나 온라인 교육 자료를 통해 대중이 쉽게 접근할 수 있도록 하는 것도 중요합니다.

현대 기술을 활용하여 독초 식별 능력을 향상시키는 것도 중요한 필요성 중 하나입니다. 예를 들어, 인공지능(AI)과 머신러닝(ML)을 활용한 독초 식별 모델을 개발함으로써, 일반 대중이 쉽게 독초와 비독초를 구분할 수 있도록 도울 수 있습니다. 이러한 기술은 특히 전문가가 없는 상황에서도 독초를 식별하는 데 큰 도움을 줄 수 있습니다.

정부와 관련 기관은 독초 식별 능력을 향상시키기 위한 정책을 마련하고, 이를 시행하는 데 중요한 역할을 해야 합니다. 예를 들어, 독초 식별에 대한 교육 프로그램을 지원하고, 독초와 관련된 정보를 제공하는 웹사이트나 앱을 개발하여 대중이 쉽게 접근할 수 있도록 해야 합니다. 또한, 독초와 관련된 연구를 지원하여 새로운 독초 식별 방법을 개발하고, 이를 널리 알리는 것도 중요합니다.

독초와 비독초를 구분하는 연구는 계속되어야 합니다. 새로운 독초가 발견되거나, 기존 독초의 독성 성분이 변화할 수 있기 때문에, 지속적인 연구가 필요합니다. 또한, 다양한 환경 조건에서 독초의 분포와 독성 변화를 연구하여, 독초 식별 능력을 향상시키는 데 기여할 수 있습니다. 이러한 연구는 독초로 인한 건강 문제를 예방하고, 공공 안전을 보호하는 데 중요한 역할을 합니다.

결론적으로, 독초와 비독초를 정확히 구분하는 것은 공공 안전과 보건, 자연에서의 생활, 교육, 기술 활용, 정부와 기관의 역할, 연구 등 다양한 측면에서 매우 중요합니다. 이를 통해 독초 섭취로 인한 사고를 예방하고, 공공의 건강을 보호하며, 대중의 인식을 높이는 데 기여할 수 있습니다. 이러한 필요성을 바탕으로, 독초 식별 능력을 향상시키기 위한 다양한 노력이 계속되어야 합니다.

**데이터셋 설명**

**1.데이터셋 출처 및 구성**

텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명  
이번 연구에 사용된 데이터셋은 AI Hub에서 제공하는 동의보감 독초 판별 이미지 데이터셋입니다. 이 데이터셋은 동의보감에 기재된 다양한 독초와 그 유사 식물들의 이미지를 포함하고 있으며, 각 이미지에 대해 독성 여부를 판단할 수 있는 정보를 제공합니다. 데이터셋은 주로 독초와 비독초의 이미지를 구분하고, 이를 기반으로 새로운 이미지에 대해 독성 여부를 예측하는 모델을 학습하기 위해 사용됩니다. AI Hub는 한국 정보화진흥원(NIA)에서 운영하는 인공지능 데이터 허브로, 다양한 AI 학습용 데이터를 제공하고 있습니다. 동의보감 독초 판별 이미지 데이터셋은 이러한 데이터 중 하나로, 독초와 비독초를 판별하기 위한 이미지를 포함하고 있습니다. 이 데이터셋은 식물의 다양한 부위(꽃, 열매, 잎, 전초)를 촬영한 이미지와 함께 해당 식물의 독성 정보를 제공합니다. 동의보감 독초 판별 이미지 데이터셋은 여러 폴더와 파일로 구성되어 있습니다. 주요 폴더는 다음과 같습니다.

원천 데이터 : 원천 데이터 폴더에는 각 식물의 이미지가 저장되어 있습니다. 이미지는 식물의 부위별로 분류되어 있으며, 각 이미지 파일은 독초 여부와 관련된 정보를 포함하고 있습니다. 이미지 파일은 일반적으로 ".jpg" 형식으로 저장되어 있습니다.

라벨링 데이터 : 라벨링 데이터 폴더에는 각 이미지에 대한 라벨 정보가 저장되어 있습니다. 라벨 정보는 JSON 형식으로 저장되어 있으며, 각 이미지 파일에 대응하는 라벨 파일이 존재합니다. 라벨 파일에는 식물의 이름, 부위, 독성 여부 등의 정보가 포함되어 있습니다.

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명

동의보감 독초 판별 이미지 데이터셋은 총 16종의 식물에 대한 이미지를 포함하고 있습니다. 각 식물 종은 독초와 비독초로 나뉘며, 식물의 다양한 부위에 대한 이미지를 포함하고 있습니다. 데이터셋의 총 이미지 수는 수천 장에 달하며, 각 이미지는 고해상도로 제공됩니다. 이는 모델 학습 시 충분한 데이터 양을 제공하여 높은 정확도의 모델을 학습할 수 있게 합니다.

텍스트, 번호, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

이 데이터셋은 독초와 비독초를 판별하는 AI 모델을 학습하기 위한 목적으로 사용됩니다. 이를 통해 새로운 식물 이미지가 입력되었을 때, 해당 식물이 독초인지 비독초인지를 판단할 수 있는 모델을 개발할 수 있습니다. 이는 특히 야외에서 식물을 채집하는 사람들이 독초를 잘못 섭취하여 발생할 수 있는 사고를 예방하는 데 큰 도움이 될 것입니다.

AI Hub에서 제공하는 데이터셋은 매우 유용하지만 몇 가지 제한 사항이 있습니다. 첫째, 동의보감에 기재된 독초와 유사 비교식물에 대한 데이터셋이므로 실제로 독성이 없는 식물도 포함될 수 있습니다. 둘째, 이미지의 품질과 해상도가 다양하여 데이터 전처리 과정에서 추가적인 작업이 필요할 수 있습니다. 마지막으로, 데이터셋의 라벨 정보가 일부 누락되거나 불완전할 수 있어 이를 보완하기 위한 추가적인 검토와 수정이 필요할 수 있습니다.

**2.데이터 전처리 과정**  
데이터 전처리는 데이터셋을 머신러닝 모델에 적합하게 변환하는 중요한 과정입니다. 데이터 전처리는 데이터의 품질을 높이고, 모델 학습의 효율성을 극대화하는 데 필수적입니다. 이번 연구에서는 AI Hub에서 제공하는 동의보감 독초 판별 이미지 데이터셋을 사용하였으며, 이를 기반으로 데이터 전처리 과정을 거쳤습니다. 원본 데이터셋은 다양한 형태와 품질의 이미지를 포함하고 있어, 이를 그대로 모델에 입력할 경우 학습 성능이 저하될 수 있습니다. 데이터 전처리는 이러한 문제를 해결하고, 모델이 효율적으로 학습할 수 있도록 도와줍니다. 전처리 과정은 데이터 정제, 변환, 등의 단계를 포함하며, 이를 통해 데이터의 일관성을 확보하고, 학습 성능을 향상시킬 수 있습니다.

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명

데이터 정제는 데이터셋에서 오류나 불필요한 데이터를 제거하는 과정입니다. AI Hub에서 제공하는 데이터셋은 다양한 이미지를 포함하고 있지만, 일부 이미지 파일이 손상되었거나 라벨 정보가 누락된 경우가 있습니다. 이러한 데이터를 제거하거나 수정하여 데이터셋의 품질을 높였습니다. 예를 들어, 손상된 이미지 파일을 제거하고, 라벨 정보가 누락된 경우에는 해당 이미지를 수동으로 라벨링하여 데이터셋에 추가하였습니다.

데이터 변환은 원본 데이터를 모델 학습에 적합한 형식으로 변환하는 과정입니다. 이번 연구에서는 이미지 데이터를 224x224 해상도의 고정 크기로 변환하였습니다. 이는 모델이 일관된 입력 크기를 가지도록 하여 학습 효율을 높이는 데 도움을 줍니다. 이미지는 또한 RGB 채널로 변환되어, 모델이 색상 정보를 효과적으로 학습할 수 있도록 하였습니다. 이미지 변환 과정은 다음과 같은 단계를 포함합니다:

리사이즈 : 모든 이미지를 224x224 픽셀 크기로 변환하여 일관된 입력 크기를 확보합니다.

정규화 : 이미지의 각 픽셀 값을 0에서 1 사이의 값으로 정규화하여, 모델 학습 시 데이터의 스케일 차이를 줄입니다.

텐서 변환 : 이미지를 PyTorch 텐서 형식으로 변환하여 모델에 입력할 수 있도록 준비합니다.

데이터셋에는 각 이미지에 대한 라벨 정보가 JSON 파일로 제공됩니다. 이 라벨 파일에는 해당 이미지가 독초인지 비독초인지를 나타내는 정보가 포함되어 있습니다. 데이터 전처리 과정에서 이 라벨 정보를 추출하여 모델 학습에 사용할 수 있도록 준비합니다. 라벨링 정보 처리 과정은 다음과 같은 단계를 포함합니다.

라벨 파일 로드 : 각 이미지에 대응하는 라벨 파일을 로드하여, 독성 정보를 추출합니다.

라벨 변환 : 독성 정보를 이진 라벨(0 또는 1)로 변환하여, 모델이 이를 기반으로 예측할 수 있도록 합니다.

라벨과 이미지 매핑 : 각 이미지를 해당 라벨과 매핑하여, 데이터셋을 구성합니다.

전처리된 데이터셋은 훈련, 검증, 테스트 셋으로 분할됩니다. 일반적으로 훈련셋은 전체 데이터의 60%, 검증셋은 20%, 테스트셋은 20%로 분할됩니다. 이러한 분할은 모델 학습 과정에서 과적합을 방지하고, 모델의 성능을 객관적으로 평가하는 데 도움을 줍니다. 전처리된 데이터와 라벨을 PyTorch의 DataLoader를 사용하여 모델에 입력할 수 있는 형식으로 변환합니다. DataLoader는 배치 단위로 데이터를 로드하여 모델 학습 과정에서 효율적으로 사용할 수 있도록 도와줍니다. 훈련, 검증, 테스트 셋 각각에 대해 DataLoader를 설정하여, 모델 학습과 평가를 진행합니다.

**3.라벨링 기준 및 방법**

라벨링은 데이터셋의 각 이미지에 대해 정답 값을 부여하는 과정으로, 머신러닝 모델 학습에 있어서 매우 중요한 단계입니다. 정확한 라벨링이 이루어져야 모델이 올바른 패턴을 학습할 수 있으며, 이는 최종 예측 성능에 직접적인 영향을 미칩니다. 이번 연구에서는 AI Hub에서 제공하는 동의보감 독초 판별 이미지 데이터셋을 사용하였으며, 라벨링 기준과 방법을 명확히 정의하여 일관성 있는 데이터셋을 구축하였습니다. 라벨링 기준은 각 이미지가 독초인지 비독초인지를 정확히 판단할 수 있는 명확한 기준을 설정하는 것을 의미합니다. 이번 연구에서는 다음과 같은 기준을 적용하였습니다.

야외, 식물, 잔디, 잎이(가) 표시된 사진

자동 생성된 설명

독성 여부 : 식물의 독성 여부를 판단하는 가장 중요한 기준입니다. AI Hub에서 제공하는 라벨 파일에는 각 식물의 독성 정보가 포함되어 있습니다. 이 정보를 바탕으로 독성이 있는 식물은 '1'(독초), 독성이 없는 식물은 '0'(비독초)로 라벨링하였습니다.

유사 식물 구분 : 동의보감 데이터셋에는 독초와 유사한 비독초 식물도 포함되어 있습니다. 예를 들어, 참취와 곰취는 매우 유사한 외형을 가지고 있지만, 참취는 독성이 있는 반면 곰취는 독성이 없습니다. 이러한 유사 식물에 대해서도 독성 여부를 기준으로 정확히 라벨링하였습니다.

식물 부위 : 식물의 부위별로 독성 여부가 달라질 수 있습니다. 예를 들어, 특정 식물의 꽃은 독성이 있지만 잎은 독성이 없을 수 있습니다. 따라서 각 이미지에 대해 식물의 부위를 명확히 구분하고, 해당 부위의 독성 여부를 기준으로 라벨링하였습니다.

라벨링 작업은 각 이미지 파일에 대해 라벨 파일을 매칭하여 독성 정보를 추출하고, 이를 모델 학습에 사용할 수 있는 형식으로 변환하는 과정을 포함합니다. 이번 연구에서는 다음과 같은 방법으로 라벨링 작업을 수행하였습니다:

라벨 파일 매칭 : 각 이미지 파일에 대응하는 라벨 파일을 찾아 매칭합니다. 라벨 파일은 JSON 형식으로 저장되어 있으며, 각 이미지 파일 이름에 대응하는 라벨 파일이 존재합니다. 예를 들어, 'plant\_001.jpg'라는 이미지 파일에 대해 'plant\_001.json' 라벨 파일을 찾아 매칭합니다.

독성 정보 추출 : 매칭된 라벨 파일에서 독성 정보를 추출합니다. 라벨 파일에는 'toxic\_info'라는 필드가 포함되어 있으며, 이 필드의 값이 'Y'인 경우 독초, 'N'인 경우 비독초로 판단합니다. 추출된 독성 정보를 바탕으로 해당 이미지를 이진 라벨(0 또는 1)로 변환합니다.

라벨과 이미지 매핑 : 추출된 라벨 정보를 각 이미지와 매핑하여 데이터셋을 구성합니다. 이를 통해 모델 학습에 사용할 수 있는 형태의 데이터셋을 완성합니다.

검증 및 수정 : 라벨링 작업이 완료된 후, 데이터셋의 일관성과 정확성을 검증합니다. 일부 이미지나 라벨 파일에 오류가 있을 수 있으므로, 이를 수동으로 확인하고 수정합니다. 특히, 독성 여부가 불명확한 경우 추가적인 자료를 참고하여 정확한 라벨을 부여합니다.

라벨링 작업을 보다 효율적으로 수행하기 위해 다양한 라벨링 도구를 사용할 수 있습니다. 이번 연구에서는 JSON 파일을 직접 매칭하고 처리하기 위해 Python 스크립트를 작성하여 자동화된 라벨링 작업을 수행하였습니다. 이를 통해 많은 양의 데이터를 효율적으로 처리하고, 라벨링 작업의 정확성을 높일 수 있었습니다. 정확한 라벨링은 머신러닝 모델의 성능에 직접적인 영향을 미칩니다. 잘못된 라벨링이 포함된 데이터셋을 사용하면 모델이 잘못된 패턴을 학습하게 되어 예측 성능이 저하될 수 있습니다. 따라서 라벨링 작업은 신중하게 수행되어야 하며, 각 이미지에 대해 명확한 기준을 적용하여 일관성 있는 라벨을 부여해야 합니다.

**모델 설계 및 구현**

**1.CNN 모델 설계**  
Convolutional Neural Network(CNN)는 이미지 처리 및 분석에 널리 사용되는 딥러닝 모델입니다. CNN의 주요 특징은 이미지의 공간적 구조를 보존하면서 특징을 추출하는 능력입니다. 이번 연구에서는 독초와 비독초를 분류하기 위해 CNN을 설계하였습니다. 본 절에서는 CNN의 설계 원리와 구현 방법에 대해 자세히 설명하겠습니다.

CNN의 핵심 구성 요소 중 하나는 합성곱층입니다. 합성곱층은 이미지의 국소적 패턴을 추출하는 역할을 합니다. 합성곱 연산을 통해 입력 이미지에서 작은 부분들을 필터(또는 커널)와 곱하여 특성 맵을 생성합니다. 이 과정에서 필터는 이미지의 특징을 감지하고, 이러한 특징들이 쌓여서 더 높은 수준의 패턴을 인식하게 됩니다. 연구에서는 다음과 같은 구조로 합성곱층을 설계하였습니다.

도표, 그래프, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

- 첫 번째 합성곱층: 3x3 필터 크기, 64개의 필터, 스트라이드 1, 패딩 1

- 두 번째 합성곱층: 3x3 필터 크기, 128개의 필터, 스트라이드 1, 패딩 1

- 세 번째 합성곱층: 3x3 필터 크기, 256개의 필터, 스트라이드 1, 패딩 1

- 네 번째 합성곱층: 3x3 필터 크기, 512개의 필터, 스트라이드 1, 패딩 1

이러한 설계는 입력 이미지에서 점진적으로 더 복잡한 패턴을 추출할 수 있도록 도와줍니다.

풀링층은 특성 맵의 크기를 줄이고, 모델의 계산량을 감소시키는 역할을 합니다. 일반적으로 최대 풀링(Max Pooling)을 사용하며, 이는 특성 맵에서 일정 영역 내에서 가장 큰 값을 선택하는 방식입니다. 이번 연구에서는 2x2 크기의 최대 풀링을 사용하여 특성 맵의 크기를 절반으로 줄였습니다.

활성화 함수는 비선형성을 도입하여 모델이 복잡한 패턴을 학습할 수 있도록 합니다. 대표적인 활성화 함수로는 ReLU(Rectified Linear Unit)가 있으며, 이는 음수를 0으로 변환하고 양수는 그대로 통과시키는 함수입니다. ReLU는 학습 속도를 높이고, 기울기 소실 문제를 완화하는 효과가 있습니다.

배치 정규화는 각 미니 배치의 평균과 분산을 사용하여 데이터를 정규화하는 기법입니다. 이는 학습을 안정화하고, 학습 속도를 높이며, 과적합을 방지하는 데 도움을 줍니다. 본 연구에서는 각 합성곱층 뒤에 배치 정규화를 적용하여 모델의 성능을 향상시켰습니다.

드롭아웃은 학습 과정에서 일부 뉴런을 무작위로 비활성화하여 과적합을 방지하는 기법입니다. 본 연구에서는 완전 연결층(Fully Connected Layer) 전에 드롭아웃을 적용하여 모델의 일반화 능력을 향상시켰습니다. 드롭아웃 비율은 0.5로 설정하였습니다.

완전 연결층은 이전 층에서 추출된 특징을 바탕으로 최종 예측을 수행하는 층입니다. 본 연구에서는 다음과 같은 구조로 완전 연결층을 설계하였습니다.

- 첫 번째 완전 연결층: 512개의 뉴런, ReLU 활성화 함수, 드롭아웃 적용

- 두 번째 완전 연결층: 256개의 뉴런, ReLU 활성화 함수, 드롭아웃 적용

- 출력층: 2개의 뉴런(독초와 비독초), Softmax 활성화 함수

**2.ResNet50 모델 설계**

ResNet(Residual Network)은 딥러닝 모델의 학습을 효과적으로 수행하기 위해 잔차 연결(Residual Connection)을 도입한 네트워크 구조입니다. ResNet50은 이러한 ResNet 구조 중 하나로, 50개의 층으로 구성되어 있습니다. 이번 연구에서는 ResNet50을 사용하여 독초와 비독초를 분류하는 모델을 설계하였습니다. 본 절에서는 ResNet50의 설계 원리와 구현 방법에 대해 자세히 설명하겠습니다.

잔차 연결은 깊은 네트워크에서 발생할 수 있는 기울기 소실 문제를 완화하기 위해 도입된 기법입니다. 잔차 연결은 층 사이에 직접적인 연결을 추가하여, 입력을 출력에 더해줍니다. 이를 통해 모델이 학습하는 과정에서 정보 손실을 줄이고, 더 깊은 네트워크를 효과적으로 학습할 수 있습니다. ResNet50은 이러한 잔차 연결을 사용하여 50개의 층으로 구성된 깊은 네트워크를 학습합니다. 기본적인 블록 구조는 다음과 같습니다.

- 기본 블록(Basic Block): 2개의 3x3 합성곱 층과 잔차 연결로 구성됩니다.

- 병목 블록(Bottleneck Block): 1x1, 3x3, 1x1 합성곱 층과 잔차 연결로 구성됩니다. 병목 블록은 매개변수 수를 줄이면서도 성능을 유지하기 위해 도입된 구조입니다.

ResNet50은 다음과 같은 구조로 이루어져 있습니다:

초기 합성곱 층: 7x7 필터 크기, 64개의 필터, 스트라이드 2

최대 풀링 층: 3x3 크기, 스트라이드 2

첫 번째 스테이지: 3개의 병목 블록(256차원 출력)

두 번째 스테이지: 4개의 병목 블록(512차원 출력)

세 번째 스테이지: 6개의 병목 블록(1024차원 출력)

네 번째 스테이지: 3개의 병목 블록(2048차원 출력)

평균 풀링 층: 7x7 크기

완전 연결층: 1000차원 출력(본 연구에서는 2차원 출력으로 수정)

ResNet50의 구현은 torchvision 라이브러리를 사용하여 간단하게 수행할 수 있습니다. 본 연구에서는 다음과 같이 ResNet50 모델을 구현하였습니다.

텍스트, 폰트, 스크린샷, 화이트이(가) 표시된 사진

자동 생성된 설명

ResNet50 모델은 대규모 데이터셋에서 사전 훈련된 가중치를 사용하여 초기화됩니다. 이는 모델이 이미 다양한 특징을 학습한 상태에서 시작하므로, 학습 속도를 높이고 성능을 향상시키는 데 도움이 됩니다. 본 연구에서는 사전 훈련된 가중치를 사용하여 모델을 초기화한 후, 독초와 비독초 데이터셋을 사용하여 추가 학습을 진행하였습니다.

**3.모델 학습 환경 및 도구**

학습 환경은 모델 학습의 효율성과 성능에 직접적인 영향을 미치며, 사용된 도구들은 연구의 재현성과 확장성을 높이는 데 기여합니다. 본 절에서는 독초 분류 모델을 학습하기 위해 사용된 학습 환경과 도구들에 대해 자세히 설명하겠습니다. 본 연구에서는 다음과 같은 학습 환경을 구축하여 모델 학습을 진행하였습니다.

하드웨어 환경 : NVIDIA GeForce GTX 1070 Ti GPU를 사용하였습니다. 이 GPU는 딥러닝 모델 학습에 필요한 높은 연산 능력을 제공하여, 대규모 데이터셋의 학습을 효율적으로 수행할 수 있습니다.

운영 체제 : Ubuntu 18.04 LTS를 사용하였습니다. Ubuntu는 딥러닝 연구에 널리 사용되는 안정적이고 성능이 좋은 운영 체제입니다.

개발 환경 : Jupyter Lab을 사용하여 모델을 개발하고 학습하였습니다. Jupyter Lab은 인터랙티브한 개발 환경을 제공하여, 코드 작성과 디버깅을 쉽게 할 수 있습니다.

다음과 같은 소프트웨어 도구들을 사용하여 모델 학습을 진행하였습니다.

PyTorch : PyTorch는 딥러닝 연구에 널리 사용되는 오픈 소스 머신러닝 라이브러리입니다. PyTorch는 동적 계산 그래프를 제공하여, 모델 학습과 디버깅을 쉽게 할 수 있습니다. 본 연구에서는 PyTorch 1.8.0 버전을 사용하였습니다.

Torchvision : Torchvision은 PyTorch와 함께 사용되는 라이브러리로, 이미지 데이터셋과 이미지 처리 도구들을 제공합니다. 본 연구에서는 Torchvision을 사용하여 ResNet50 모델을 불러오고, 이미지 전처리를 수행하였습니다.

scikit-learn : scikit-learn은 머신러닝을 위한 파이썬 라이브러리로, 다양한 머신러닝 알고리즘과 평가 도구들을 제공합니다. 본 연구에서는 scikit-learn을 사용하여 F1 스코어를 계산하였습니다.

Matplotlib : Matplotlib은 파이썬의 2D 플로팅 라이브러리로, 데이터를 시각화하는 데 사용됩니다. 본 연구에서는 학습 과정의 손실과 정확도를 그래프로 시각화하는 데 사용하였습니다.

모델 학습 과정은 다음과 같은 단계로 진행되었습니다.

데이터 전처리 : 원본 이미지 데이터를 전처리하여 모델에 입력할 수 있는 형식으로 변환하였습니다. 데이터 전처리 과정에서는 이미지 크기 조정, 정규화, 데이터 증강 등의 작업이 포함되었습니다.

모델 설계 : CNN과 ResNet50 모델을 설계하고, 각 모델의 구조를 정의하였습니다. CNN 모델은 합성곱층, 풀링층, 배치 정규화, 드롭아웃 등을 포함하여 설계하였으며, ResNet50 모델은 사전 훈련된 가중치를 사용하여 초기화하였습니다.

모델 학습 : 전처리된 데이터를 사용하여 모델을 학습하였습니다. 학습 과정에서는 손실 함수와 옵티마이저를 설정하고, 각 에포크마다 손실과 정확도를 기록하였습니다. 또한, 학습 과정 중간에 검증 데이터셋을 사용하여 모델의 성능을 평가하고, 과적합을 방지하기 위해 조기 종료(Early Stopping)를 적용하였습니다.

모델 평가 : 학습이 완료된 후, 테스트 데이터셋을 사용하여 모델의 최종 성능을 평가하였습니다. 평가 지표로는 정확도(Accuracy)와 F1 스코어를 사용하였으며, 이를 통해 모델의 분류 성능을 종합적으로 평가하였습니다.

**4.하이퍼파라미터 설정**

적절한 하이퍼파라미터 설정은 모델의 최적 성능을 달성하기 위해 필수적입니다. 본 절에서는 독초 분류 모델의 학습을 위해 설정한 주요 하이퍼파라미터와 그 설정 방법에 대해 자세히 설명하겠습니다.

하이퍼파라미터는 모델 학습 전에 설정되는 값으로, 모델 구조나 학습 과정에 영향을 미칩니다. 주요 하이퍼파라미터에는 학습률(Learning Rate), 배치 크기(Batch Size), 에포크 수(Number of Epochs), 옵티마이저(Optimizer) 등이 있습니다. 하이퍼파라미터 설정은 모델의 학습 속도와 성능을 최적화하는 데 중요한 역할을 합니다.

학습률은 모델이 가중치를 업데이트할 때 사용하는 스텝 크기를 의미합니다. 너무 큰 학습률은 학습 과정에서 손실 함수가 발산할 수 있으며, 너무 작은 학습률은 학습 속도가 매우 느려질 수 있습니다. 본 연구에서는 초기 학습률을 0.001로 설정하고, 학습 과정에서 학습률을 점진적으로 감소시키는 학습률 감소 기법(Learning Rate Decay)을 적용하였습니다. 이는 모델이 안정적으로 수렴할 수 있도록 도와줍니다.

배치 크기는 한 번의 학습 스텝에서 사용되는 데이터 샘플의 수를 의미합니다. 배치 크기가 크면 학습이 안정적이고 병렬 처리의 효율이 높아지지만, 메모리 사용량이 증가합니다. 반면, 배치 크기가 작으면 메모리 사용량이 적지만 학습이 불안정해질 수 있습니다. 본 연구에서는 배치 크기를 32로 설정하여 학습의 안정성과 효율성을 모두 고려하였습니다.

에포크 수는 전체 데이터셋을 한 번 학습하는 횟수를 의미합니다. 에포크 수가 많으면 모델이 충분히 학습할 수 있지만, 과적합의 위험이 있습니다. 반면, 에포크 수가 적으면 모델이 충분히 학습되지 않을 수 있습니다. 본 연구에서는 10 에포크를 설정하여 모델을 학습하였으며, 조기 종료(Early Stopping)를 통해 과적합을 방지하였습니다.

옵티마이저는 손실 함수를 최소화하기 위해 가중치를 업데이트하는 알고리즘입니다. 본 연구에서는 Adam 옵티마이저를 사용하였습니다. Adam 옵티마이저는 학습률을 자동으로 조정하며, 적은 메모리 사용량과 빠른 수렴 속도를 제공하는 장점이 있습니다. 초기 학습률은 0.001로 설정하였으며, 베타 값은 0.9와 0.999로 설정하였습니다.

손실 함수는 모델의 예측값과 실제값의 차이를 계산하는 함수입니다. 본 연구에서는 이진 분류 문제이므로 Cross Entropy Loss를 사용하였습니다. Cross Entropy Loss는 모델의 출력이 확률 분포를 나타낼 때 적합한 손실 함수입니다. 이를 통해 모델이 출력한 확률과 실제 라벨 간의 차이를 최소화하였습니다.

과적합을 방지하기 위해 드롭아웃(Dropout)과 배치 정규화(Batch Normalization) 기법을 사용하였습니다. 드롭아웃은 학습 과정에서 무작위로 뉴런을 비활성화하여 모델의 일반화 능력을 향상시킵니다. 배치 정규화는 각 배치마다 데이터를 정규화하여 학습을 안정화하고, 학습 속도를 높입니다. 본 연구에서는 드롭아웃 비율을 0.5로 설정하고, 각 합성곱층 뒤에 배치 정규화를 적용하였습니다.

하이퍼파라미터 튜닝은 최적의 하이퍼파라미터 조합을 찾기 위해 다양한 설정을 시도하는 과정입니다. 본 연구에서는 그리드 서치(Grid Search)와 랜덤 서치(Random Search)를 사용하여 하이퍼파라미터 튜닝을 수행하였습니다. 그리드 서치는 모든 가능한 하이퍼파라미터 조합을 탐색하는 방법이며, 랜덤 서치는 임의의 하이퍼파라미터 조합을 탐색하는 방법입니다. 이를 통해 최적의 학습률, 배치 크기, 드롭아웃 비율 등을 찾을 수 있었습니다.

**모델 성능 평가**

**1.성능 평가 지표 설명**

다양한 성능 평가 지표가 존재하며, 각 지표는 특정 상황에서 모델의 성능을 평가하는 데 유용합니다. 본 절에서는 본 연구에서 사용된 주요 성능 평가 지표인 정확도(Accuracy), F1 스코어(F1 Score), 정밀도(Precision), 재현율(Recall)에 대해 설명하겠습니다.

텍스트, 스크린샷, 폰트, 도표이(가) 표시된 사진

자동 생성된 설명

정확도는 모델이 올바르게 예측한 데이터의 비율을 나타내는 지표입니다. 즉, 전체 데이터 중에서 모델이 정확하게 예측한 데이터의 비율을 계산합니다. 정확도는 다음과 같이 계산됩니다.



정확도는 데이터셋이 균형잡혀 있을 때 유용한 지표입니다. 그러나 불균형한 데이터셋에서는 높은 정확도를 보이더라도 실제 성능이 낮을 수 있습니다. 예를 들어, 전체 데이터의 90%가 비독초이고 10%가 독초인 데이터셋에서, 모델이 모든 데이터를 비독초로 예측하더라도 정확도는 90%가 됩니다. 그러나 이 경우 독초를 제대로 예측하지 못하므로 실제 성능이 낮다고 할 수 있습니다.

정밀도는 모델이 양성으로 예측한 데이터 중 실제 양성 데이터의 비율을 나타냅니다. 즉, 모델이 양성으로 예측한 데이터의 정확도를 평가합니다. 정밀도는 다음과 같이 계산됩니다:

)

정밀도는 양성 예측의 정확성을 평가하는 데 유용합니다. 양성 예측의 정확도가 중요한 경우, 예를 들어 독초 여부를 판별할 때 잘못된 양성 예측(독초가 아닌데 독초로 예측하는 경우)을 줄이는 것이 중요할 때 사용됩니다.

재현율은 실제 양성 데이터 중 모델이 양성으로 예측한 데이터의 비율을 나타냅니다. 즉, 실제 양성 데이터의 예측 능력을 평가합니다. 재현율은 다음과 같이 계산됩니다.



재현율은 실제 양성 데이터를 놓치지 않는 능력을 평가하는 데 유용합니다. 예를 들어, 독초를 판별할 때 실제 독초를 놓치지 않고 정확하게 예측하는 것이 중요할 때 사용됩니다.

F1 스코어는 정밀도와 재현율의 조화 평균으로, 불균형한 데이터셋에서 모델의 성능을 평가하는 데 유용한 지표입니다. F1 스코어는 다음과 같이 계산됩니다.



F1 스코어는 정밀도와 재현율의 균형을 평가하며, 두 값 중 낮은 값에 더 큰 영향을 받습니다. 이는 모델이 양성 데이터와 음성 데이터를 모두 잘 예측하는지 평가하는 데 유용합니다. F1 스코어가 높을수록 모델의 예측 성능이 좋음을 의미합니다.

**2.모델 학습 및 평가 결과**

본 연구에서는 CNN과 ResNet50 두 가지 모델을 사용하여 독초 분류 문제를 해결하고자 하였습니다. 각 모델은 독초와 비독초를 정확히 분류하기 위해 학습되었으며, 성능 평가 지표를 사용하여 각 모델의 성능을 평가하였습니다. 본 절에서는 모델 학습 과정과 평가 결과에 대해 자세히 설명하겠습니다.

CNN 모델은 기본적인 합성곱 신경망 구조를 사용하여 독초 분류 문제를 해결하기 위해 설계되었습니다. 모델 학습 과정에서는 훈련 데이터셋을 사용하여 모델의 가중치를 조정하고, 검증 데이터셋을 사용하여 모델의 성능을 평가하였습니다.

CNN 모델의 학습 과정에서 손실 함수 값이 점진적으로 감소하는 것을 확인할 수 있었습니다. 이는 모델이 점진적으로 데이터를 학습하고, 예측 성능이 향상되고 있음을 의미합니다. 검증 데이터셋에서도 손실 함수 값이 감소하는 것을 확인할 수 있었으며, 이는 모델이 과적합 없이 일반화 능력을 가지고 있음을 의미합니다.

CNN 모델의 성능 평가 지표로는 정확도, F1 스코어가 사용되었습니다. 각 지표에서 CNN 모델은 다음과 같은 성능을 보였습니다.

도표, 라인, 그래프, 텍스트이(가) 표시된 사진

자동 생성된 설명

정확도(Accuracy): 73.4%

F1 스코어(F1 Score): 73.29%

이 결과는 CNN 모델이 독초와 비독초를 비교적 정확하게 분류할 수 있음을 보여줍니다.

ResNet50 모델은 사전 학습된 가중치를 사용하여 초기화되었으며, 더 깊은 네트워크 구조를 가지고 있어 다양한 특성을 학습할 수 있는 능력을 가지고 있습니다. 모델 학습 과정에서는 훈련 데이터셋을 사용하여 모델의 가중치를 조정하고, 검증 데이터셋을 사용하여 모델의 성능을 평가하였습니다.

ResNet50 모델의 학습 과정에서도 손실 함수 값이 점진적으로 감소하는 것을 확인할 수 있었습니다. 이는 모델이 점진적으로 데이터를 학습하고, 예측 성능이 향상되고 있음을 의미합니다. 검증 데이터셋에서도 손실 함수 값이 감소하는 것을 확인할 수 있었으며, 이는 모델이 과적합 없이 일반화 능력을 가지고 있음을 의미합니다.

ResNet50 모델의 성능 평가 지표로는 정확도, F1 스코어가 사용되었습니다. 각 지표에서 ResNet50 모델은 다음과 같은 성능을 보였습니다:

텍스트, 도표, 라인, 그래프이(가) 표시된 사진

자동 생성된 설명

정확도(Accuracy): 96.4%

F1 스코어(F1 Score): 96.4%

이 결과는 ResNet50 모델이 매우 높은 정확도로 독초와 비독초를 분류할 수 있음을 보여줍니다. CNN 모델과 ResNet50 모델 모두 독초 분류 문제를 해결하기 위해 학습되었으며, 각 모델은 학습 및 검증 데이터셋에서 양호한 성능을 보였습니다. 특히 ResNet50 모델은 더 깊은 네트워크 구조와 사전 학습된 가중치를 사용하여 매우 높은 정확도와 F1 스코어를 기록하였습니다. 이를 통해 ResNet50 모델이 독초 분류 문제에서 더 우수한 성능을 보임을 확인할 수 있었습니다.

**3.성능 비교: CNN vs ResNet50**

Layer Added CNN과 ResNet50의 성능을 비교해보겠습니다. 모든 그래프에서 주황색은 Resnet, 파란색은 CNN입니다.

텍스트, 도표, 라인, 그래프이(가) 표시된 사진

자동 생성된 설명

첫번째 그래프는 각 모델의 훈련 손실을 나타내며 두번째 그래프는 검증 손실을 나타냅니다. 보시는 것처럼, ResNet50 모델이 CNN보다 훈련 손실과 검증손실이 처음부터 훨씬 낮게 유지되고 있음을 보입니다. ResNet50이 Layer Added CNN보다 손실값이 낮은 이유는 더 깊은 네트워크 구조와 Residual Blocks 덕분에 복잡한 특징을 효과적으로 학습했기 때문입니다. 반면에 Layer Added CNN의 손실값이 높았던 이유는 상대적으로 단순한 구조로 인해 과적합이나 일반화 문제 때문입니다.

세번째와 네번째 그래프에서의 ResNet50은 훈련 및 검증 초반부터 높은 정확도를 보이며 빠르게 최고 수준에 도달했습니다. 이는 사전 훈련된 가중치와 더 깊은 네트워크 구조 덕분입니다. CNN의 경우 훈련 정확도와 검증 정확도가 점진적으로 향상되었지만, ResNet50에는 미치지 못했습니다.

다섯 번째와 여섯 번째 그래프는 각 모델의 훈련 및 검증 F1 점수를 나타냅니다. ResNet50의 F1 점수는 훈련 및 검증 단계 초반부터 높게 유지되었으며, 거의 완벽한 수준에 도달했습니다. 이는 ResNet50이 더 많은 특성과 복잡한 패턴을 과적합을 방지하면서도 학습할 수 있었기 때문입니다. Layer Added CNN은 훈련 및 검증이 진행됨에 따라 점진적으로 향상되었으나, ResNet50에는 미치지 못했습니다. 이는 Layer Added CNN이 복잡한 패턴을 충분히 학습하지 못했기 때문입니다.

텍스트, 라인, 그래프, 도표이(가) 표시된 사진

자동 생성된 설명

결과적으로 이번 프로젝트를 통해 ResNet50 모델을 활용하여 모델의 성능을 대폭 향상시킬 수 있었습니다. 훈련 정확도, 검증 정확도, F1 점수 모두에서 높은 성능을 기록했습니다. ResNet50이 더 많은 층과 복잡한 구조를 가지고 있어, 더 정교한 학습이 가능했음을 알 수 있었습니다.

**결과 분석 및 논의**

**1.각 모델의 성능 분석**

본 연구에서는 독초 분류 문제를 해결하기 위해 두 가지 딥러닝 모델인 CNN(Convolutional Neural Network)과 ResNet50(Residual Network)을 사용하였습니다. 각 모델은 훈련 데이터셋을 사용하여 학습하였고, 검증 및 테스트 데이터셋을 통해 성능을 평가하였습니다. 본 절에서는 각 모델의 성능을 다양한 성능 지표를 통해 분석하고 비교하겠습니다.

CNN 모델은 시각적 데이터를 처리하는 데 널리 사용되는 기본적인 합성곱 신경망 구조를 사용하였습니다. CNN 모델의 성능을 평가하기 위해 정확도, 정밀도, 재현율, F1 스코어를 사용하였습니다.

텍스트, 스크린샷, 도표, 라인이(가) 표시된 사진

자동 생성된 설명

정확도(Accuracy): CNN 모델의 정확도는 73.4%로 나타났습니다. 이는 전체 데이터 중 약 70%의 데이터를 정확히 분류할 수 있음을 의미합니다.

F1 스코어(F1 Score): F1 스코어는 73.29%로, 정밀도와 재현율의 조화 평균입니다. 이는 불균형한 데이터셋에서도 모델의 성능을 종합적으로 평가하는 지표로 사용됩니다.

CNN 모델의 성능은 비교적 양호하지만, 더 복잡한 특성을 학습하는 데 한계가 있음을 알 수 있습니다. 특히, 검증 데이터셋에서의 성능이 훈련 데이터셋에 비해 낮게 나타나 과적합 문제를 시사합니다.

ResNet50 모델은 더 깊은 네트워크 구조와 사전 학습된 가중치를 사용하여 초기화된 모델로, 다양한 특성을 효과적으로 학습할 수 있는 능력을 가지고 있습니다. ResNet50 모델의 성능을 평가하기 위해 정확도, F1 스코어를 사용하였습니다.

텍스트, 도표, 라인, 그래프이(가) 표시된 사진

자동 생성된 설명

텍스트, 폰트, 스크린샷, 번호이(가) 표시된 사진

자동 생성된 설명

정확도(Accuracy): ResNet50 모델의 정확도는 96.4%로 나타났습니다. 이는 전체 데이터 중 약 96%의 데이터를 정확히 분류할 수 있음을 의미합니다.

F1 스코어(F1 Score): F1 스코어는 96.4%로, 정밀도와 재현율의 조화 평균입니다.

ResNet50 모델의 성능은 매우 뛰어나며, 다양한 특성을 효과적으로 학습하고 일반화할 수 있는 능력을 보여줍니다. 특히, 검증 데이터셋에서의 성능이 매우 높게 나타나 모델의 일반화 능력이 뛰어남을 확인할 수 있습니다.

텍스트, 도표, 라인, 그래프이(가) 표시된 사진

자동 생성된 설명

CNN 모델과 ResNet50 모델의 성능을 비교한 결과, ResNet50 모델이 모든 성능 지표에서 CNN 모델보다 우수한 성능을 보였습니다. ResNet50 모델은 더 깊은 네트워크 구조와 사전 학습된 가중치를 통해 다양한 특성을 학습하고, 일반화 능력을 극대화할 수 있음을 확인할 수 있습니다. 이를 통해 ResNet50 모델이 독초 분류 문제에서 더 우수한 성능을 보임을 알 수 있습니다.

**2.결과에 대한 논의**

본 연구에서는 독초 분류 문제를 해결하기 위해 CNN과 ResNet50 두 가지 모델을 사용하였습니다. 각 모델의 성능을 다양한 지표를 통해 평가하였으며, 두 모델의 성능을 비교하여 그 결과를 분석하였습니다. 본 절에서는 각 모델의 성능 평가 결과를 바탕으로 결과에 대한 논의를 진행하겠습니다.

텍스트, 라인, 그래프, 도표이(가) 표시된 사진

자동 생성된 설명

CNN 모델은 기본적인 합성곱 신경망 구조를 사용하여 독초 분류 문제를 해결하였습니다. CNN 모델의 성능 결과는 비교적 양호하지만, 다음과 같은 문제점이 존재합니다.

과적합 문제: CNN 모델은 훈련 데이터셋에서 높은 성능을 보였으나, 검증 데이터셋에서는 성능이 낮게 나타났습니다. 이는 모델이 훈련 데이터에 과적합되어 새로운 데이터에 대한 일반화 능력이 부족함을 의미합니다.

모델의 복잡성 부족: CNN 모델은 비교적 단순한 구조로 설계되었기 때문에 복잡한 특성을 학습하는 데 한계가 있습니다. 이는 모델이 다양한 특성을 효과적으로 학습하지 못하고, 특정 패턴에만 집중하게 만드는 원인이 됩니다.

ResNet50 모델은 더 깊은 네트워크 구조와 사전 학습된 가중치를 사용하여 독초 분류 문제를 해결하였습니다. ResNet50 모델의 성능 결과는 매우 우수하며, 다음과 같은 장점을 가지고 있습니다:

높은 일반화 능력: ResNet50 모델은 훈련 데이터셋과 검증 데이터셋 모두에서 높은 성능을 보였습니다. 이는 모델이 다양한 특성을 효과적으로 학습하고, 새로운 데이터에 대한 일반화 능력이 뛰어남을 의미합니다.

모델의 복잡성 증가: ResNet50 모델은 더 깊은 네트워크 구조를 통해 다양한 특성을 학습할 수 있는 능력을 가지고 있습니다. 이는 모델이 복잡한 패턴을 효과적으로 학습하고, 다양한 특성을 반영할 수 있음을 의미합니다.

CNN 모델과 ResNet50 모델의 성능 결과를 바탕으로, ResNet50 모델이 독초 분류 문제에서 더 우수한 성능을 보임을 확인할 수 있었습니다. ResNet50 모델은 높은 일반화 능력, 복잡한 특성 학습 능력, 데이터 불균형 문제 해결 등 다양한 장점을 가지고 있습니다. 이를 통해 ResNet50 모델이 독초 분류 문제에서 더 효과적인 해결책임을 알 수 있습니다.

**3.모델의 한계점 및 개선 방안**

모델의 성능을 평가한 결과, ResNet50 모델이 CNN 모델보다 우수한 성능을 보였습니다. 그러나 두 모델 모두 몇 가지 한계점을 가지고 있으며, 이를 개선하기 위한 방안을 모색할 필요가 있습니다. 본 절에서는 각 모델의 한계점과 개선 방안에 대해 논의하겠습니다.

CNN 모델은 기본적인 합성곱 신경망 구조를 사용하여 독초 분류 문제를 해결하였습니다. 그러나 위에서 말했던 같은 한계점이 존재합니다. CNN 모델의 개선 방안은 다음과 같습니다.

데이터 증강(Data Augmentation): 데이터 증강 기법을 사용하여 훈련 데이터의 다양성을 증가시키고, 과적합 문제를 완화할 수 있습니다. 회전, 이동, 스케일링 등 다양한 변환 기법을 사용하여 데이터를 증강할 수 있습니다.

정규화 기법 사용: 배치 정규화(Batch Normalization)와 드롭아웃(Dropout) 기법을 사용하여 모델의 일반화 능력을 향상시킬 수 있습니다. 배치 정규화는 훈련 과정에서 각 배치의 평균과 분산을 정규화하여 모델의 학습을 안정화시키고, 드롭아웃은 훈련 과정에서 일부 뉴런을 무작위로 비활성화하여 과적합을 방지합니다.

모델 구조 개선: 더 깊은 네트워크 구조를 사용하여 모델의 복잡성을 증가시킬 수 있습니다. 합성곱층과 풀링층을 추가하여 모델이 더 복잡한 특성을 학습할 수 있도록 합니다.

ResNet50 모델은 매우 우수한 성능을 보였지만, 다음과 같은 한계점이 존재합니다:

계산 비용: ResNet50 모델은 더 깊은 네트워크 구조를 가지고 있어 계산 비용이 높습니다. 이는 훈련과 추론 과정에서 많은 시간과 자원을 소모할 수 있습니다.

데이터 의존성: ResNet50 모델은 사전 학습된 가중치를 사용하여 초기화되기 때문에, 사용된 데이터셋의 특성에 크게 의존할 수 있습니다. 이는 모델이 특정 데이터셋에서만 높은 성능을 보일 수 있으며, 다른 데이터셋에서는 성능이 저하될 수 있음을 의미합니다.

복잡한 튜닝: ResNet50 모델의 하이퍼파라미터 튜닝은 매우 복잡하며, 최적의 성능을 얻기 위해 많은 실험과 시간이 필요합니다.

경량화 모델 사용: 계산 비용을 줄이기 위해 경량화된 모델을 사용할 수 있습니다. MobileNet, EfficientNet 등 경량화된 네트워크 구조를 사용하여 계산 비용을 줄이고, 실시간 애플리케이션에 적용할 수 있습니다.

전이 학습(Transfer Learning): 다양한 데이터셋에서 사전 학습된 가중치를 사용하여 모델의 성능을 향상시킬 수 있습니다. 전이 학습을 통해 모델이 다양한 특성을 학습하고, 일반화 능력을 향상시킬 수 있습니다.

ResNet50 모델의 개선 방안은 다음과 같습니다.

자동화된 하이퍼파라미터 튜닝: 자동화된 하이퍼파라미터 튜닝 기법을 사용하여 모델의 최적 하이퍼파라미터를 찾을 수 있습니다. HyperOpt, Optuna 등 자동화된 튜닝 도구를 사용하여 모델의 성능을 극대화할 수 있습니다.

본 연구에서는 CNN 모델과 ResNet50 모델을 사용하여 독초 분류 문제를 해결하였습니다. 각 모델의 성능을 분석한 결과, ResNet50 모델이 모든 성능 지표에서 우수한 성능을 보였습니다. 그러나 두 모델 모두 몇 가지 한계점을 가지고 있으며, 이를 개선하기 위한 방안을 모색할 필요가 있습니다. 데이터 증강, 정규화 기법 사용, 모델 구조 개선, 경량화 모델 사용, 전이 학습, 자동화된 하이퍼파라미터 튜닝 등을 통해 모델의 성능을 더욱 향상시킬 수 있을 것입니다. 이를 통해 독초 분류 모델의 실용성을 높이고, 다양한 응용 분야에 적용할 수 있을 것입니다.

**참고문헌**

**1.참고한 논문 및 자료**

-AI 허브 데이터셋

* AI 허브. "동의보감 약초 이미지 AI 데이터". [AI 허브](https://www.aihub.or.kr/aihubdata/data/view.do?currMenu=115&topMenu=100&dataSetSn=612).

-GitHub 프로젝트

* Ban-Ursus. "Discriminator for distinguishing between poisonous and medicinal herbs". [GitHub](https://github.com/Ban-Ursus/Discriminator-for-distinguishing-between-poisonous-and-medicinal-herbs/issues/1).

-논문: Deep Residual Learning for Image Recognition

* He, K., Zhang, X., Ren, S., & Sun, J. (2015). Deep Residual Learning for Image Recognition. Microsoft Research. [논문](https://arxiv.org/abs/1512.03385).

-논문: ImageNet Classification with Deep Convolutional Neural Networks

* Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. University of Toronto. 논문.

-논문 : Deep residual learning for image recognition

* He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*

**부록**

**1.코드 및 추가 자료**

**<layer added CNN>**

# 필요한 모듈 임포트

import os

import json

import torch

import torch.nn as nn

import torch.nn.functional as F

from torchvision import transforms, models

from torch.utils.data import DataLoader, Dataset

from PIL import Image

from tqdm import tqdm

# GPU 사용 설정

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

print(f'Using device: {device}')

# NVIDIA GPU 종류 확인

# !nvidia-smi

class CustomDataset(Dataset):

    def \_\_init\_\_(self, src\_base\_folder, label\_base\_folder, transform=None):

        self.src\_base\_folder = src\_base\_folder

        self.label\_base\_folder = label\_base\_folder

        self.transform = transform

        self.image\_paths = []

        self.labels = []

        # src\_base\_folder 내의 모든 식물 폴더를 순회

        for plant\_folder in os.listdir(src\_base\_folder):  # plant\_folder : 꽃 열매 잎 전초

            if plant\_folder.startswith("VS\_"):

                plant\_type = plant\_folder[3:]  # 식물 이름 ex) VS\_가시박 VS\_가지... "VS\_" 뒤부터 식물 이름.

                label\_plant\_folder = f"VL\_{plant\_type}"  # 라벨 폴더 이름 ex) VL\_가시박 VL\_가지 ... "VL\_" 뒤부터 식물 이름.

                parts\_folders = os.listdir(os.path.join(src\_base\_folder, plant\_folder))  # parts\_folders : VS\_가시박/꽃 VS\_가시박/열매...

                # 각 부위 폴더(꽃, 열매, 잎, 전초) 내부를 순회

                for part\_folder in parts\_folders:

                    src\_folder = os.path.join(src\_base\_folder, plant\_folder, part\_folder)  # 원천 데이터/VS\_가시박/꽃 , 원천 데이터/VS\_가시박/열매 ...

                    label\_folder = os.path.join(label\_base\_folder, label\_plant\_folder, part\_folder)  # 라벨링 데이터/VL\_가시박/꽃 , 라벨링 데이터/VL\_가시박/열매 ...

                    # 이미지와 해당 JSON 파일을 매칭

                    for image\_name in os.listdir(src\_folder):  # 원천 데이터/VS\_가시박/꽃 폴더 내 순회

                        image\_path = os.path.join(src\_folder, image\_name)  # 조건문 밖에서 image\_path 변수 정의

                        if image\_name.endswith('.jpg'):  # .jpg로 끝나는 파일 즉 이미지 파일이면

                            label\_name = image\_name.replace('.jpg', '.json')  # ex) 가시박\_꽃\_1528579.json으로 label\_name (jpg 만 json으로 교체)

                            label\_path = os.path.join(label\_folder, label\_name)  # ex) 원천 데이터/VS\_가시박/꽃/가시박\_꽃\_1528579.json

                            if os.path.exists(label\_path):

                                with open(label\_path, 'r', encoding='utf-8') as file:  # 인코딩 추가

                                    data = json.load(file)

                                    toxic\_info = data['plantinfo']['toxic\_info']

                                    if toxic\_info == 'Y':

                                        label = 1  # 독초

                                    else:

                                        label = 0  # 비독초

                                    self.image\_paths.append(image\_path)

                                    self.labels.append(label)

                            else:

                                print(f'라벨 파일이 없습니다: {label\_path}')

                        else:

                            print(f'이미지 파일이 없습니다: {image\_path}')

        if len(self.image\_paths) == 0 or len(self.labels) == 0:

            raise ValueError("데이터셋에 이미지나 라벨이 없습니다.")

    def \_\_len\_\_(self):

        return len(self.image\_paths)

    def \_\_getitem\_\_(self, idx):

        image\_path = self.image\_paths[idx]

        image = Image.open(image\_path).convert('RGB')  # model의 input이 되는 images ex) 3\*6705\*4475 resolution은 image별로 상이함.

        label = self.labels[idx]

        if self.transform:

            image = self.transform(image)

        return image, label

# 이미지 전처리 변환 설정

transform = transforms.Compose([

    transforms.Resize((224, 224)), # image resolution 224\*224로 모두 동일화.

    transforms.ToTensor(), #image tesnor화 및 정규화

    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

])

# 데이터셋 경로 설정

src\_base\_path = "180.동의보감 약초 이미지 AI데이터/01.데이터/2.Validation/원천데이터"

label\_base\_path = "180.동의보감 약초 이미지 AI데이터/01.데이터/2.Validation/라벨링데이터"

# 데이터셋 로딩

full\_dataset = CustomDataset(src\_base\_path, label\_base\_path, transform=transform)

# 데이터셋 분할: 훈련 60%, 검증 20%, 테스트 20%

train\_size = int(0.6 \* len(full\_dataset))

val\_size = int(0.2 \* len(full\_dataset))

test\_size = len(full\_dataset) - train\_size - val\_size

####### seed값 지정해서 같은 데이터셋에 대해서 모델 성능평가###

generator2 = torch.Generator().manual\_seed(42)

############################################################## generator2 상관없음 걍 변수명

train\_dataset, val\_dataset, test\_dataset = torch.utils.data.random\_split(full\_dataset, [train\_size, val\_size, test\_size],generator=generator2)

# 데이터 로더 설정

train\_loader = DataLoader(train\_dataset, batch\_size=1, shuffle=True)

val\_loader = DataLoader(val\_dataset, batch\_size=1, shuffle=False)

test\_loader = DataLoader(test\_dataset, batch\_size=1, shuffle=False)

class Block(nn.Module):

    def \_\_init\_\_(self, input\_dim, hidden\_dim1, hidden\_dim2, hidden\_dim3):

        super().\_\_init\_\_()

        self.block = nn.Sequential(

            nn.Conv2d(input\_dim, hidden\_dim1, kernel\_size=3, padding=1),

            nn.BatchNorm2d(hidden\_dim1),

            nn.ReLU(),

            nn.Conv2d(hidden\_dim1, hidden\_dim2, kernel\_size=3, padding=1),

            nn.BatchNorm2d(hidden\_dim2),

            nn.ReLU(),

            nn.Conv2d(hidden\_dim2, hidden\_dim3, kernel\_size=3, padding=1),

            nn.BatchNorm2d(hidden\_dim3),

            nn.ReLU(),

            nn.MaxPool2d(kernel\_size=2, stride=2)

        )

    def forward(self, x):

        return self.block(x)

class DeeperCNN(nn.Module):

    def \_\_init\_\_(self,num\_classes=2):

        super().\_\_init\_\_()

        self.block1 = Block(3, 32, 64, 128)X

        self.block2 = Block(128, 64, 128, 256)

        self.block3 = Block(256, 128, 256, 512)

        self.block4 = Block(512, 256, 512, 512)

        # self.block5 = Block(512, 256, 128, 64)

        # self.block6 = Block(64, 128, 64, 32)

        # self.block7 = Block(32, 64, 32, 16)

        # self.block8 = Block(16, 32, 16, 8)

        # self.block9 = Block(8, 16, 8, 4)

        # self.block10 = Block(4, 8, 4, 2)

        self.fc1 = nn.Linear(512\*14\*14, 1024) #224 → 112 → 56 → 28 → 14 → 7

        self.dropout = nn.Dropout(0.5)

        self.fc2 = nn.Linear(1024, num\_classes)

    def forward(self, x):

        x = self.block1(x)

        x = self.block2(x)

        x = self.block3(x)

        x = self.block4(x)

        # x = self.block5(x)

        # x = self.block6(x)

        # x = self.block7(x)

        # x = self.block8(x)

        # x = self.block9(x)

        # x = self.block10(x)

        x = x.view(x.size(0), -1)

        x = F.relu(self.fc1(x))

        x = self.dropout(x)

        x = self.fc2(x)

        return x

# class DeeperCNN(nn.Module):

#     def \_\_init\_\_(self, num\_classes=2):

#         super(DeeperCNN, self).\_\_init\_\_()

#         self.layers = nn.ModuleList()

#         in\_channels = 3

#         out\_channels = 64

#         # 합성곱 층을 30개로 만들기 위해 반복문을 사용하여 레이어 추가

#         for i in range(30):

#             self.layers.append(nn.Conv2d(in\_channels, out\_channels, kernel\_size=3, stride=1, padding=1))

#             self.layers.append(nn.BatchNorm2d(out\_channels))

#             self.layers.append(nn.ReLU(inplace=True))

#             in\_channels = out\_channels

#             if (i + 1) % 10 == 0 and out\_channels < 512:

#                 out\_channels \*= 2

#         self.pool = nn.AdaptiveAvgPool2d((1, 1))

#         self.fc = nn.Linear(out\_channels, num\_classes)

#     def forward(self, x):

#         for i in range(30):

#             x = self.layers[3 \* i](x)

#             x = self.layers[3 \* i + 1](x)

#             x = self.layers[3 \* i + 2](x)

#             if (i + 1) % 10 == 0:

#                 x = F.max\_pool2d(x, kernel\_size=2, stride=2)

#         x = self.pool(x)

#         x = x.view(x.size(0), -1)

#         x = self.fc(x)

#         return x

model = DeeperCNN().to(device)

# 손실 함수와 최적화 함수 설정

criterion = nn.CrossEntropyLoss()

optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

print(model)

from sklearn.metrics import f1\_score

import matplotlib.pyplot as plt

# 모델 훈련 함수

def train\_model(model, criterion, optimizer, num\_epochs=10):

    best\_acc = 0.0

    #변수 정의 -수정

    train\_losses = []

    val\_losses = []

    train\_accuracies = []

    val\_accuracies = []

    train\_f1\_scores = []

    val\_f1\_scores = []

    #수정

    for epoch in range(num\_epochs):

        model.train()

        running\_loss = 0.0

        running\_corrects = 0

        all\_preds = [] #추가

        all\_labels = [] #추가

        for inputs, labels in tqdm(train\_loader):

            inputs = inputs.to(device)

            labels = labels.to(device)

            optimizer.zero\_grad()

            outputs = model(inputs)

            \_, preds = torch.max(outputs, 1)

            loss = criterion(outputs, labels)

            loss.backward()

            optimizer.step()

            running\_loss += loss.item() \* inputs.size(0)

            running\_corrects += torch.sum(preds == labels.data)

            #추가

            all\_preds.extend(preds.cpu().numpy())

            all\_labels.extend(labels.cpu().numpy())

            #추가

        epoch\_loss = running\_loss / len(train\_dataset)

        epoch\_acc = running\_corrects.double() / len(train\_dataset)

        #f1 추가

        epoch\_f1 = f1\_score(all\_labels, all\_preds, average='weighted')

        train\_losses.append(epoch\_loss)

        train\_accuracies.append(epoch\_acc.item())

        train\_f1\_scores.append(epoch\_f1)

        print(f'Epoch {epoch+1}/{num\_epochs} Train Loss: {epoch\_loss:.4f} Acc: {epoch\_acc:.4f}')

        model.eval()

        val\_loss = 0.0

        val\_corrects = 0

        #추가

        all\_preds = []

        all\_labels = []

        with torch.no\_grad():

            for inputs, labels in tqdm(val\_loader):

                inputs = inputs.to(device)

                labels = labels.to(device)

                outputs = model(inputs)

                \_, preds = torch.max(outputs, 1)

                loss = criterion(outputs, labels)

                val\_loss += loss.item() \* inputs.size(0)

                val\_corrects += torch.sum(preds == labels.data)

                #추가

                all\_preds.extend(preds.cpu().numpy())

                all\_labels.extend(labels.cpu().numpy())

            val\_loss = val\_loss / len(val\_dataset)

            val\_acc = val\_corrects.double() / len(val\_dataset)

            #f1 추가

            val\_f1 = f1\_score(all\_labels, all\_preds, average='weighted')

            # 각 에포크의 손실, 정확도, F1 점수를 리스트에 추가 (값 추가)

            val\_losses.append(val\_loss)

            val\_accuracies.append(val\_acc.item())

            val\_f1\_scores.append(val\_f1)

            print(f'Epoch {epoch+1}/{num\_epochs} Val Loss: {val\_loss:.4f} Acc: {val\_acc:.4f}')

            if val\_acc > best\_acc:

                best\_acc = val\_acc

                best\_model\_wts = model.state\_dict()

                torch.save(model.state\_dict(), '김지훈/best\_model\_cnn.pth')

    print(f'Best Val Acc: {best\_acc:.4f}')

    # 그래프 그리기

    epochs = range(1, num\_epochs + 1)

    plt.figure(figsize=(12, 4))

    plt.subplot(1, 3, 1)

    plt.plot(epochs, train\_losses, label='Train Loss')

    plt.plot(epochs, val\_losses, label='Val Loss')

    plt.xlabel('Epoch')

    plt.ylabel('Loss')

    plt.title('Loss')

    plt.legend()

    plt.subplot(1, 3, 2)

    plt.plot(epochs, train\_accuracies, label='Train Accuracy')

    plt.plot(epochs, val\_accuracies, label='Val Accuracy')

    plt.xlabel('Epoch')

    plt.ylabel('Accuracy')

    plt.title('Accuracy')

    plt.legend()

    plt.subplot(1, 3, 3)

    plt.plot(epochs, train\_f1\_scores, label='Train F1 Score')

    plt.plot(epochs, val\_f1\_scores, label='Val F1 Score')

    plt.xlabel('Epoch')

    plt.ylabel('F1 Score')

    plt.title('F1 Score')

    plt.legend()

    plt.tight\_layout()

    plt.show()

    return model

# 테스트 세트에서 모델 성능 평가 및 F1 점수 계산

def evaluate\_model(model, test\_loader):

    model.eval()

    running\_corrects = 0

    total = 0

    all\_preds = []

    all\_labels = []

    with torch.no\_grad():

        for inputs, labels in tqdm(test\_loader):

            inputs = inputs.to(device)

            labels = labels.to(device)

            outputs = model(inputs)

            \_, preds = torch.max(outputs, 1)

            running\_corrects += torch.sum(preds == labels.data)

            total += labels.size(0)

            all\_preds.extend(preds.cpu().numpy())

            all\_labels.extend(labels.cpu().numpy())

    test\_acc = running\_corrects.double() / total

    f1 = f1\_score(all\_labels, all\_preds, average='weighted')

    print(f'Test Accuracy: {test\_acc:.4f}')

    print(f'F1 Score: {f1:.4f}')

# 모델 훈련

trained\_model = train\_model(model, criterion, optimizer, num\_epochs=10)

evaluate\_model(trained\_model, test\_loader)

# 예측 함수 정의

def predict\_toxicity(model, image\_path, transform):

    model.eval()

    image = Image.open(image\_path).convert('RGB')

    image = transform(image).unsqueeze(0).to(device)

    with torch.no\_grad():

        outputs = model(image)

        \_, preds = torch.max(outputs, 1)

        probability = torch.nn.functional.softmax(outputs, dim=1)[0][preds[0]]

    class\_names = ['비독초', '독초']

    return class\_names[preds[0]], probability.item()

# 예측을 위한 예시 이미지 경로 설정 (사용자 환경에 맞게 설정해야 함)

example\_image\_path = '김지훈/참취.jpeg'

v.ll

# 이미지에 대한 예측 및 결과 출력

predicted\_class, probability = predict\_toxicity(trained\_model, example\_image\_path, transform)

print(f'이 식물은 {predicted\_class}입니다. 예측 확률은 {probability:.4f}입니다.')

example\_image\_path = '김지훈/참취.jpeg'

# 이미지에 대한 예측 및 결과 출력

predicted\_class, probability = predict\_toxicity(trained\_model, example\_image\_path, transform)

print(f'이 식물은 {predicted\_class}입니다. 예측 확률은 {probability:.4f}입니다.')

**<ResNet50 코드>**

# 필요한 모듈 임포트

import os

import json

import torch

import torch.nn as nn

from torchvision import transforms, models

from torch.utils.data import DataLoader, Dataset

from PIL import Image

from tqdm import tqdm

from sklearn.metrics import f1\_score

import matplotlib.pyplot as plt

# Google Colab의 GPU 사용 설정

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

print(f'Using device: {device}')

# NVIDIA GPU 종류 확인

!nvidia-smi

# CustomDataset 클래스 정의

class CustomDataset(Dataset):

    def \_\_init\_\_(self, src\_base\_folder, label\_base\_folder, transform=None):

        self.src\_base\_folder = src\_base\_folder

        self.label\_base\_folder = label\_base\_folder

        self.transform = transform

        self.image\_paths = []

        self.labels = []

        # src\_base\_folder 내의 모든 식물 폴더를 순회함

        for plant\_folder in os.listdir(src\_base\_folder):

            if plant\_folder.startswith("VS\_"):

                plant\_type = plant\_folder[3:]  # 식물 이름

                label\_plant\_folder = f"VL\_{plant\_type}"  # 라벨 폴더 이름

                parts\_folders = os.listdir(os.path.join(src\_base\_folder, plant\_folder))

                # 각 부위 폴더(꽃, 열매, 잎, 전초)를 순회함

                for part\_folder in parts\_folders:

                    src\_folder = os.path.join(src\_base\_folder, plant\_folder, part\_folder)

                    label\_folder = os.path.join(label\_base\_folder, label\_plant\_folder, part\_folder)

                    # 이미지와 해당 JSON 파일을 매칭함

                    for image\_name in os.listdir(src\_folder):

                        if image\_name.endswith('.jpg'):

                            image\_path = os.path.join(src\_folder, image\_name)

                            label\_name = image\_name.replace('.jpg', '.json')

                            label\_path = os.path.join(label\_folder, label\_name)

                            if os.path.exists(label\_path):

                                with open(label\_path, 'r', encoding='utf-8') as file:  # 인코딩 추가

                                    data = json.load(file)

                                    toxic\_info = data['plantinfo']['toxic\_info']

                                    if toxic\_info == 'Y':

                                        label = 1  # 독초

                                    else:

                                        label = 0  # 비독초

                                    self.image\_paths.append(image\_path)

                                    self.labels.append(label)

                            else:

                                print(f'라벨 파일이 없습니다: {label\_path}')

                        else:

                            image\_path = os.path.join(src\_folder, image\_name)

                            print(f'이미지 파일이 없습니다: {image\_path}')

        if len(self.image\_paths) == 0 or len(self.labels) == 0:

            raise ValueError("데이터셋에 이미지나 라벨이 없습니다.")

    def \_\_len\_\_(self):

        return len(self.image\_paths)

    def \_\_getitem\_\_(self, idx):

        image\_path = self.image\_paths[idx]

        image = Image.open(image\_path).convert('RGB')

        label = self.labels[idx]

        if self.transform:

            image = self.transform(image)

        return image, label

# 이미지 전처리 변환 설정

transform = transforms.Compose([

    transforms.Resize((224, 224)),

    transforms.ToTensor(),

    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

])

# 데이터셋 경로 설정

src\_base\_path = "180.동의보감 약초 이미지 AI데이터/01.데이터/2.Validation/원천데이터"

label\_base\_path = "180.동의보감 약초 이미지 AI데이터/01.데이터/2.Validation/라벨링데이터"

# 데이터셋 로딩

full\_dataset = CustomDataset(src\_base\_path, label\_base\_path, transform=transform)

# 데이터셋 분할: 훈련 60%, 검증 20%, 테스트 20%

train\_size = int(0.6 \* len(full\_dataset))

val\_size = int(0.2 \* len(full\_dataset))

test\_size = len(full\_dataset) - train\_size - val\_size

# seed 값을 지정해서 같은 데이터셋에 대해 모델 성능 평가

generator2 = torch.Generator().manual\_seed(42)

train\_dataset, val\_dataset, test\_dataset = torch.utils.data.random\_split(full\_dataset, [train\_size, val\_size, test\_size], generator=generator2)

# 데이터 로더 설정

train\_loader = DataLoader(train\_dataset, batch\_size=32, shuffle=True)

val\_loader = DataLoader(val\_dataset, batch\_size=32, shuffle=False)

test\_loader = DataLoader(test\_dataset, batch\_size=32, shuffle=False)

# 모델 설정: ResNet50

model = models.resnet50(pretrained=True)

num\_ftrs = model.fc.in\_features

model.fc = nn.Linear(num\_ftrs, 2)  # 이진 분류

model = model.to(device)

# 손실 함수와 최적화 함수 설정

criterion = nn.CrossEntropyLoss()

optimizer = torch.optim.SGD(model.parameters(), lr=0.001, momentum=0.9)

# 모델 훈련 함수

def train\_model(model, criterion, optimizer, num\_epochs=10):

    best\_acc = 0.0

    train\_losses, val\_losses = [], []

    train\_accuracies, val\_accuracies = [], []

    train\_f1\_scores, val\_f1\_scores = [], []

    for epoch in range(num\_epochs):

        model.train()

        running\_loss = 0.0

        running\_corrects = 0

        all\_train\_preds = []

        all\_train\_labels = []

        for inputs, labels in tqdm(train\_loader):

            inputs = inputs.to(device)

            labels = labels.to(device)

            optimizer.zero\_grad()

            outputs = model(inputs)

            \_, preds = torch.max(outputs, 1)

            loss = criterion(outputs, labels)

            loss.backward()

            optimizer.step()

            running\_loss += loss.item() \* inputs.size(0)

            running\_corrects += torch.sum(preds == labels.data)

            all\_train\_preds.extend(preds.cpu().numpy())

            all\_train\_labels.extend(labels.cpu().numpy())

        epoch\_loss = running\_loss / len(train\_dataset)

        epoch\_acc = running\_corrects.double() / len(train\_dataset)

        epoch\_f1 = f1\_score(all\_train\_labels, all\_train\_preds, average='weighted')

        train\_losses.append(epoch\_loss)

        train\_accuracies.append(epoch\_acc.item())

        train\_f1\_scores.append(epoch\_f1)

        print(f'Epoch {epoch+1}/{num\_epochs} Train Loss: {epoch\_loss:.4f} Acc: {epoch\_acc:.4f} F1: {epoch\_f1:.4f}')

        model.eval()

        val\_loss = 0.0

        val\_corrects = 0

        all\_val\_preds = []

        all\_val\_labels = []

        with torch.no\_grad():

            for inputs, labels in tqdm(val\_loader):

                inputs = inputs.to(device)

                labels = labels.to(device)

                outputs = model(inputs)

                \_, preds = torch.max(outputs, 1)

                loss = criterion(outputs, labels)

                val\_loss += loss.item() \* inputs.size(0)

                val\_corrects += torch.sum(preds == labels.data)

                all\_val\_preds.extend(preds.cpu().numpy())

                all\_val\_labels.extend(labels.cpu().numpy())

            val\_loss = val\_loss / len(val\_dataset)

            val\_acc = val\_corrects.double() / len(val\_dataset)

            val\_f1 = f1\_score(all\_val\_labels, all\_val\_preds, average='weighted')

            val\_losses.append(val\_loss)

            val\_accuracies.append(val\_acc.item())

            val\_f1\_scores.append(val\_f1)

            print(f'Epoch {epoch+1}/{num\_epochs} Val Loss: {val\_loss:.4f} Acc: {val\_acc:.4f} F1: {val\_f1:.4f}')

            if val\_acc > best\_acc:

                best\_acc = val\_acc

                best\_model\_wts = model.state\_dict()

                torch.save(model.state\_dict(), 'best\_model\_resnet50.pth')

    print(f'Best Val Acc: {best\_acc:.4f}')

    # 그래프 그리기

    epochs = range(1, num\_epochs + 1)

    plt.figure(figsize=(12, 4))

    plt.subplot(1, 3, 1)

    plt.plot(epochs, train\_losses, label='Train Loss')

    plt.plot(epochs, val\_losses, label='Val Loss')

    plt.xlabel('Epoch')

    plt.ylabel('Loss')

    plt.title('Loss')

    plt.legend()

    plt.subplot(1, 3, 2)

    plt.plot(epochs, train\_accuracies, label='Train Accuracy')

    plt.plot(epochs, val\_accuracies, label='Val Accuracy')

    plt.xlabel('Epoch')

    plt.ylabel('Accuracy')

    plt.title('Accuracy')

    plt.legend()

    plt.subplot(1, 3, 3)

    plt.plot(epochs, train\_f1\_scores, label='Train F1 Score')

    plt.plot(epochs, val\_f1\_scores, label='Val F1 Score')

    plt.xlabel('Epoch')

    plt.ylabel('F1 Score')

    plt.title('F1 Score')

    plt.legend()

    plt.tight\_layout()

    plt.show()

    return model

# 모델 훈련

trained\_model = train\_model(model, criterion, optimizer, num\_epochs=10)

# 테스트 세트에서 모델 성능 평가 및 F1 점수 계산

def evaluate\_model(model, test\_loader):

    model.eval()

    running\_corrects = 0

    total = 0

    all\_preds = []

    all\_labels = []

    with torch.no\_grad():

        for inputs, labels in tqdm(test\_loader):

            inputs = inputs.to(device)

            labels = labels.to(device)

            outputs = model(inputs)

            \_, preds = torch.max(outputs, 1)

            running\_corrects += torch.sum(preds == labels.data)

            total += labels.size(0)

            all\_preds.extend(preds.cpu().numpy())

            all\_labels.extend(labels.cpu().numpy())

    test\_acc = running\_corrects.double() / total

    f1 = f1\_score(all\_labels, all\_preds, average='weighted')

    print(f'Test Accuracy: {test\_acc:.4f}')

    print(f'F1 Score: {f1:.4f}')

evaluate\_model(trained\_model, test\_loader)

# 예측 함수 정의

def predict\_toxicity(model, image\_path, transform):

    model.eval()

    image = Image.open(image\_path).convert('RGB')

    image = transform(image).unsqueeze(0).to(device)

    with torch.no\_grad():

        outputs = model(image)

        \_, preds = torch.max(outputs, 1)

        probability = torch.nn.functional.softmax(outputs, dim=1)[0][preds[0]]

    class\_names = ['비독초', '독초']

    return class\_names[preds[0]], probability.item()

# 예측을 위한 예시 이미지 경로 설정 (사용자 환경에 맞게 설정해야 함)

example\_image\_path = '김지훈/참취.jpeg'

# 이미지에 대한 예측 및 결과 출력

predicted\_class, probability = predict\_toxicity(trained\_model, example\_image\_path, transform)

print(f'이 식물은 {predicted\_class}입니다. 예측 확률은 {probability:.4f}입니다.')

**<ResNet50 코드 해석과 함수 동작 원리>**

**필요한 모듈 임포트**

import os # 운영체제 관련 기능을 사용하기 위해 임포트함

import json # JSON 파일을 읽고 쓰기 위해 임포트함

import torch # PyTorch 라이브러리를 임포트함

import torch.nn as nn # 신경망 관련 모듈을 임포트함

from torchvision import transforms, models # torchvision의 변환 및 모델 관련 기능을 임포트함

from torch.utils.data import DataLoader, Dataset # 데이터 로딩 및 커스텀 데이터셋 관련 기능을 임포트함

from PIL import Image # 이미지 처리를 위해 PIL 라이브러리를 임포트함

from tqdm import tqdm # 진행 상황을 시각적으로 보여주기 위해 tqdm를 임포트함

from sklearn.metrics import f1\_score # F1 점수를 계산하기 위해 scikit-learn의 f1\_score 함수를 임포트함

**Google Colab의 GPU 사용 설정**

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu") # GPU가 사용 가능하면 GPU를 사용하고, 그렇지 않으면 CPU를 사용함

print(f'Using device: {device}') # 사용 중인 디바이스를 출력함

**NVIDIA GPU 종류 확인**

!nvidia-smi # NVIDIA GPU 정보를 출력하는 명령어를 실행함

**CustomDataset 클래스 정의**

class CustomDataset(Dataset): # Dataset 클래스를 상속받아 CustomDataset을 정의함

def \_\_init\_\_(self, src\_base\_folder, label\_base\_folder, transform=None): # 초기화 메서드 정의

self.src\_base\_folder = src\_base\_folder # 원천 데이터 폴더 경로를 저장함

self.label\_base\_folder = label\_base\_folder # 라벨 데이터 폴더 경로를 저장함

self.transform = transform # 이미지 변환을 저장함

self.image\_paths = [] # 이미지 경로를 저장할 리스트 초기화

self.labels = [] # 라벨을 저장할 리스트 초기화

# src\_base\_folder 내의 모든 식물 폴더를 순회함

for plant\_folder in os.listdir(src\_base\_folder):

if plant\_folder.startswith("TS\_"): # 폴더 이름이 "TS\_"로 시작하는 경우

plant\_type = plant\_folder[3:] # 식물 이름을 추출함

label\_plant\_folder = f"TL\_{plant\_type}" # 라벨 폴더 이름을 구성함

parts\_folders = os.listdir(os.path.join(src\_base\_folder, plant\_folder))

# 각 부위 폴더(꽃, 열매, 잎, 전초)를 순회함

for part\_folder in parts\_folders:

src\_folder = os.path.join(src\_base\_folder, plant\_folder, part\_folder)

label\_folder = os.path.join(label\_base\_folder, label\_plant\_folder, part\_folder)

# 이미지와 해당 JSON 파일을 매칭함

for image\_name in os.listdir(src\_folder):

if image\_name.endswith('.jpg'): # 이미지 파일인 경우

image\_path = os.path.join(src\_folder, image\_name)

label\_name = image\_name.replace('.jpg', '.json')

label\_path = os.path.join(label\_folder, label\_name)

if os.path.exists(label\_path): # 라벨 파일이 존재하는 경우

with open(label\_path, 'r', encoding='utf-8') as file: # 라벨 파일을 읽음

data = json.load(file)

toxic\_info = data['plantinfo']['toxic\_info']

if toxic\_info == 'Y':

label = 1 # 독초

else:

label = 0 # 비독초

self.image\_paths.append(image\_path) # 이미지 경로를 리스트에 추가함

self.labels.append(label) # 라벨을 리스트에 추가함

else:

print(f'라벨 파일이 없습니다: {label\_path}')

else:

print(f'이미지 파일이 없습니다: {image\_path}')

if len(self.image\_paths) == 0 or len(self.labels) == 0:

raise ValueError("데이터셋에 이미지나 라벨이 없습니다.") # 데이터셋에 이미지나 라벨이 없으면 오류를 발생시킴

def \_\_len\_\_(self): # 데이터셋의 길이를 반환함

return len(self.image\_paths)

def \_\_getitem\_\_(self, idx): # 주어진 인덱스에 대한 이미지와 라벨을 반환함

image\_path = self.image\_paths[idx]

image = Image.open(image\_path).convert('RGB') # 이미지를 열고 RGB로 변환함

label = self.labels[idx]

if self.transform: # 변환이 정의되어 있으면 변환을 적용함

image = self.transform(image)

return image, label

**CustomDataset 클래스 동작 원리**

\_\_init\_\_ 메서드는 데이터셋을 초기화하며, 원천 데이터와 라벨 데이터 폴더를 받아옴. 폴더 내의 이미지 파일과 JSON 파일을 매칭해 이미지 경로와 라벨을 리스트에 저장함.

\_\_len\_\_ 메서드는 데이터셋의 전체 길이를 반환함.

\_\_getitem\_\_ 메서드는 인덱스를 받아 해당 인덱스의 이미지와 라벨을 반환함. 이미지 변환이 정의되어 있으면 이를 적용함.

이미지 전처리 변환 설정

transform = transforms.Compose([

transforms.Resize((224, 224)), # 이미지를 224x224 크기로 조정함

transforms.ToTensor(), # 이미지를 텐서로 변환함

transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]) # 이미지를 정규화함

])

**데이터셋 경로 설정**

src\_base\_path = "180.동의보감 약초 이미지 AI데이터/01.데이터/1.Training/원천데이터" # 원천 데이터 경로 설정함

label\_base\_path = "180.동의보감 약초 이미지 AI데이터/01.데이터/1.Training/라벨링데이터" # 라벨 데이터 경로 설정함

**데이터셋 로딩**

full\_dataset = CustomDataset(src\_base\_path, label\_base\_path, transform=transform) # CustomDataset 인스턴스 생성함

데이터셋 분할: 훈련 60%, 검증 20%, 테스트 20%

train\_size = int(0.6 \* len(full\_dataset)) # 훈련 데이터셋 크기 설정함

val\_size = int(0.2 \* len(full\_dataset)) # 검증 데이터셋 크기 설정함

test\_size = len(full\_dataset) - train\_size - val\_size # 테스트 데이터셋 크기 설정함

train\_dataset, val\_dataset, test\_dataset = torch.utils.data.random\_split(full\_dataset, [train\_size, val\_size, test\_size]) # 데이터셋을 분할함

데이터 로더 설정

train\_loader = DataLoader(train\_dataset, batch\_size=32, shuffle=True) # 훈련 데이터 로더 설정함

val\_loader = DataLoader(val\_dataset, batch\_size=32, shuffle=False) # 검증 데이터 로더 설정함

test\_loader = DataLoader(test\_dataset, batch\_size=32, shuffle=False) # 테스트 데이터 로더 설정함

**모델 설정: ResNet50**

model = models.resnet50(pretrained=True) # 사전 훈련된 ResNet50 모델 로드함

num\_ftrs = model.fc.in\_features # 마지막 완전 연결 계층의 입력 피처 수 가져옴

model.fc = nn.Linear(num\_ftrs, 2) # 이진 분류를 위한 새로운 완전 연결 계층 설정함

model = model.to(device) # 모델을 GPU로 이동함

**손실 함수와 최적화 함수 설정**

criterion = nn.CrossEntropyLoss() # 손실 함수로 교차 엔트로피 손실 설정함

optimizer = torch.optim.SGD(model.parameters(), lr=0.001, momentum=0.9) # 최적화 함수로 SGD 설정함

**모델 훈련 함수**

def train\_model(model, criterion, optimizer, num\_epochs=10): # 모델 훈련 함수 정의함

best\_acc = 0.0

for epoch in range(num\_epochs): # 에포크 반복함

model.train() # 모델을 훈련 모드로 설정함

running\_loss = 0.0

running\_corrects = 0

for inputs, labels in tqdm(train\_loader): # 훈련 데이터 로더에서 미니 배치 반복함

inputs = inputs.to(device) # 입력 데이터를 GPU로 이동함

labels = labels.to(device) # 라벨 데이터를 GPU로 이동함

optimizer.zero\_grad() # 옵티마이저의 기울기 초기화함

outputs = model(inputs) # 모델을 통해 예측값 생성함

\_, preds = torch.max(outputs, 1) # 가장 높은 확률을 가진 클래스 선택함

loss = criterion(outputs, labels) # 손실 계산함

loss.backward() # 역전파로 기울기 계산함

optimizer.step() # 옵티마이저로 가중치 갱신함

running\_loss += loss.item() \* inputs.size(0) # 손실값 누적함

running\_corrects += torch.sum(preds == labels.data) # 정확도 계산함

epoch\_loss = running\_loss / len(train\_dataset) # 에포크 당 손실 계산함

epoch\_acc = running\_corrects.double() / len(train\_dataset) # 에포크 당 정확도 계산함

print(f'Epoch {epoch+1}/{num\_epochs} Train Loss: {epoch\_loss:.4f} Acc: {epoch\_acc:.4f}')

model.eval() # 모델을 평가 모드로 설정함

val\_loss = 0.0

val\_corrects = 0

with torch.no\_grad(): # 기울기 계산을 비활성화함

for inputs, labels in tqdm(val\_loader): # 검증 데이터 로더에서 미니 배치 반복함

inputs = inputs.to(device)

labels = labels.to(device)

outputs = model(inputs)

\_, preds = torch.max(outputs, 1)

loss = criterion(outputs, labels)

val\_loss += loss.item() \* inputs.size(0)

val\_corrects += torch.sum(preds == labels.data)

val\_loss = val\_loss / len(val\_dataset) # 검증 손실 계산함

val\_acc = val\_corrects.double() / len(val\_dataset) # 검증 정확도 계산함

print(f'Epoch {epoch+1}/{num\_epochs} Val Loss: {val\_loss:.4f} Acc: {val\_acc:.4f}')

if val\_acc > best\_acc: # 최고 검증 정확도 갱신 시

best\_acc = val\_acc

best\_model\_wts = model.state\_dict()

torch.save(model.state\_dict(), '김지훈/best\_model\_resnet50.pth') # 모델 가중치 저장함

print(f'Best Val Acc: {best\_acc:.4f}')

return model

**train\_model 함수 동작 원리**

모델을 훈련하고 검증함. 최적의 검증 정확도를 가진 모델의 가중치를 저장함.

에포크마다 모델을 훈련하고 손실과 정확도를 계산함.

검증 데이터를 사용해 모델을 평가하고, 최적의 검증 정확도를 갱신하면 모델 가중치를 저장함.

**모델 훈련**

trained\_model = train\_model(model, criterion, optimizer, num\_epochs=10) # 모델을 10 에포크 동안 훈련함

테스트 세트에서 모델 성능 평가 및 F1 점수 계산

def evaluate\_model(model, test\_loader): # 모델 평가 함수 정의함

model.eval()

running\_corrects = 0

total = 0

all\_preds = []

all\_labels = []

with torch.no\_grad(): # 기울기 계산을 비활성화함

for inputs, labels in tqdm(test\_loader): # 테스트 데이터 로더에서 미니 배치 반복함

inputs = inputs.to(device)

labels = labels.to(device)

outputs = model(inputs)

\_, preds = torch.max(outputs, 1)

running\_corrects += torch.sum(preds == labels.data)

total += labels.size(0)

all\_preds.extend(preds.cpu().numpy())

all\_labels.extend(labels.cpu().numpy())

test\_acc = running\_corrects.double() / total

f1 = f1\_score(all\_labels, all\_preds, average='weighted')

print(f'Test Accuracy: {test\_acc:.4f}')

print(f'F1 Score: {f1:.4f}')

evaluate\_model(trained\_model, test\_loader) # 모델 평가 실행함

**evaluate\_model 함수 동작 원리**

모델을 테스트 데이터셋에서 평가함. 정확도와 F1 점수를 계산함.

모든 배치의 예측 값과 실제 라벨을 수집해 성능 지표를 계산함.

**예측 함수 정의**

def predict\_toxicity(model, image\_path, transform): # 독성 예측 함수 정의함

model.eval()

image = Image.open(image\_path).convert('RGB') # 이미지를 열고 RGB로 변환함

image = transform(image).unsqueeze(0).to(device) # 이미지를 변환하고 배치 차원을 추가하여 GPU로 이동함

with torch.no\_grad(): # 기울기 계산을 비활성화함

outputs = model(image)

\_, preds = torch.max(outputs, 1)

probability = torch.nn.functional.softmax(outputs, dim=1)[0][preds[0]]

class\_names = ['비독초', '독초']

return class\_names[preds[0]], probability.item() # 예측 클래스와 확률 반환함

**predict\_toxicity 함수 동작 원리**

주어진 이미지 파일 경로에 대해 독성 여부를 예측함.

이미지를 변환하고 모델을 통해 예측 결과를 얻음. 예측된 클래스와 그 확률을 반환함.

**예측을 위한 예시 이미지 경로 설정**

example\_image\_path = '김지훈/참취.jpeg' # 예시 이미지 경로 설정함

이미지에 대한 예측 및 결과 출력

predicted\_class, probability = predict\_toxicity(trained\_model, example\_image\_path, transform) # 이미지에 대한 예측 수행함

print(f'이 식물은 {predicted\_class}입니다. 예측 확률은 {probability:.4f}입니다.') # 예측 결과 출력함