MNIST 손글씨 숫자 분류를 위한 딥러닝 모델 구현 및 성능 비교

MLP와 CNN 기반 접근법의 비교 분석

과 목 파이썬프로그래밍

분 반 02분반

전 공 컴퓨터 공학과

학 번 2021214556

이 름 문종건

작성일: 2025-05-25

초록

본 연구에서는 MNIST 손글씨 데이터셋을 활용하여 다층 퍼셉트론(MLP)과 합성곱 신경망(CNN)의 성능을 비교 분석합니다. 각 모델의 구조적 특징과 성능 차이를 이론적으로 고찰하고, 실제 구현을 통해 검증합니다. 또한 사용자 입력 이미지에 대한 전처리 과정을 개선하여 모델의 실용성을 높이는 방안을 제시합니다.

목차

1. 서론

1.1 연구 배경 및 목적

딥러닝 기술의 발전과 함께, 이미지 인식 분야에서 다양한 신경망 아키텍처가 개발되고 있습니다. 특히 손글씨 숫자 인식은 광학 문자 인식(OCR) 시스템의 기초가 되는 중요한 응용 분야입니다. 본 연구에서는 대표적인 딥러닝 아키텍처인 다층 퍼셉트론(MLP)과 합성곱 신경망(CNN)을 활용하여 MNIST 손글씨 숫자 분류 문제에 접근하고, 두 모델의 성능과 특성을 비교 분석합니다.

본 연구의 주요 목적은 다음과 같습니다:

1. MLP와 CNN 모델을 실제로 구현하여 MNIST 데이터셋에 대한 성능 비교
2. 실시간 사용자 입력에 대응하는 인터페이스 개발 및 전처리 과정 최적화
3. 실험 결과를 바탕으로 한 두 모델의 구조적 차이점과 특성 분석

1.2 사용 라이브러리 소개

본 연구에서 사용되는 주요 라이브러리들을 소개하고 각각의 역할을 설명합니다.

주요 라이브러리 역할:
- TensorFlow & Keras: 딥러닝 모델 구현을 위한 프레임워크
- NumPy: 수치 계산 및 배열 처리
- Matplotlib: 데이터 시각화 및 그래프 생성
- Gradio: 사용자 인터페이스 구현을 위한 라이브러리
- PIL (Python Imaging Library): 이미지 전처리 및 변환

CODE

```
# 이 셀은 노트북 실행 시 가장 먼저 한 번만 실행하면 됩니다.
# 필요한 라이브러리 설치 (Gradio)
!pip install gradio -q
# 주요 라이브러리 임포트
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.datasets import mnist # MNIST 데이터셋 로드
from tensorflow.keras.models import Sequential # Sequential 모델 API
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense # Keras 레이어
from tensorflow.keras.utils import to_categorical # 원-핫 인코딩 유틸리티
import numpy as np # 수치 계산
import matplotlib.pyplot as plt # 데이터 시각화
from PIL import Image, ImageOps # 이미지 처리
import gradio as gr # Gradio UI 라이브러리
from io import BytesIO # 바이트 스트림 처리 (이미지 변환 시)
import os # 운영체제 관련 기능 (파일 경로 등)
# Matplotlib 그래프가 셀 내에 바로 표시되도록 하는 설정 (Colab/Jupyter 기본)
%matplotlib inline
print("TensorFlow version:", tf.__version__)
print("Gradio version:", gr.__version__)
print("NumPy version:", np.__version__)
```

Result

```
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 54.2/54.2 MB 14.8 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 323.1/323.1 kB 10.2 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 95.2/95.2 kB 3.6 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 11.5/11.5 MB 45.3 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 72.0/72.0 kB 4.6 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 62.5/62.5 kB 3.8 MB/s eta 0:00:00
TensorFlow version: 2.18.0
Gradio version: 5.31.0
NumPy version: 2.0.2
```

1.3 MNIST 데이터셋 개요

MNIST(Modified National Institute of Standards and Technology) 데이터셋은 손글씨 숫자 인식을 위한 대표적인 벤치마크 데이터셋입니다.

MNIST 데이터셋의 특성

속성	값
이미지 크기	28×28 픽셀
색상	그레이스케일 (0-255)
클래스 수	10개 (0-9 숫자)
훈련 데이터	60,000개
테스트 데이터	10,000개
총 픽셀 수	784개 (28×28)

MNIST 데이터셋은 각 이미지가 28×28 픽셀의 그레이스케일 이미지로 구성되어 있으며, 픽셀 값은 0(검은색)에서 255(흰색) 사이의 값을 가집니다. 이 데이터셋은 간단한 구조와 적절한 크기로 인해 이미지 분류 알고리즘의 성능을 테스트하고 비교하는 데 널리 사용됩니다.

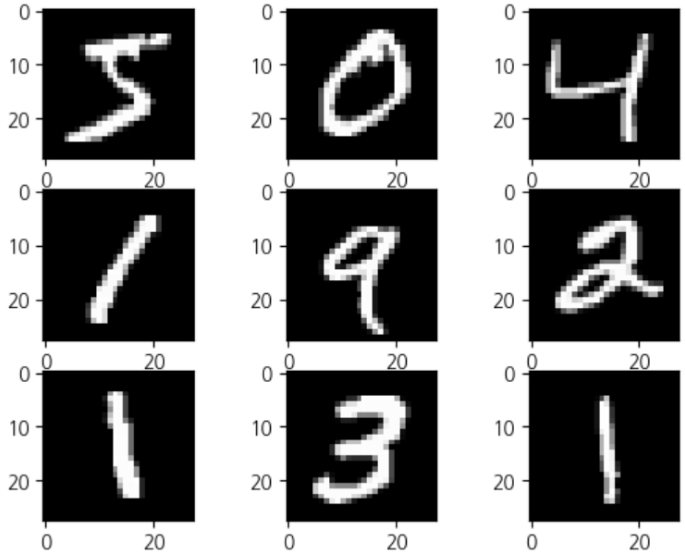
MNIST 데이터셋 시각화

CODE

```
# from keras.datasets import mnist # 상단에서 tensorflow.keras.datasets.mnist로 이미 임포트됨
# from matplotlib import pyplot # 상단에서 matplotlib.pyplot as plt로 이미 임포트됨
# 데이터셋 로드
(trainX, trainy), (testX, testy) = mnist.load_data()
# 로드된 데이터셋 요약
print('학습 데이터: X=%s, y=%s' % (trainX.shape, trainy.shape))
print('테스트 데이터: X=%s, y=%s' % (testX.shape, testy.shape))
# 처음 몇 개의 이미지 시각화
for i in range(9):
    # 서브플롯 정의
    plt.subplot(330 + 1 + i) # 상단 임포트 plt 사용
    # 픽셀 데이터 시각화
    plt.imshow(trainX[i], cmap=plt.get_cmap('gray')) # 상단 임포트 plt 사용
# 그림 표시
plt.show() # 상단 임포트 plt 사용
```

Result

```
학습 데이터: X=(60000, 28, 28), y=(60000,)
테스트 데이터: X=(10000, 28, 28), y=(10000,)
```



1.4 수학적 이론 및 공식

1.4.1 다층 퍼셉트론(MLP)의 수학적 표현

MLP의 각 뉴런은 다음과 같은 계산을 수행합니다:

가중합(Weighted Sum):

$$z = \sum_{i=1}^{n} w_i x_i + b$$

활성화 함수 적용: $$a = \sigma(z)$$

여기서 :
- x_i: 입력값
- w_i: 가중치
- b : 편향($bias$)
- σ : 활성화함수
- z : 가중합
- a : 뉴런의출력값

1.4.2 주요 활성화 함수

1) ReLU (Rectified Linear Unit)

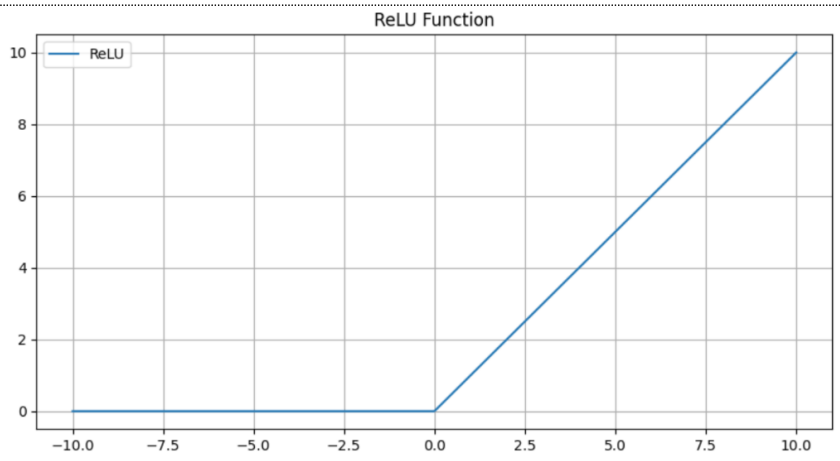
$$f(x) = \max(0, x)$$

ReLU는 음수 입력에 대해 0을 출력하고, 양수 입력에 대해서는 입력값을 그대로 출력하는 단순하면서도 효과적인 활성화 함수입니다.

CODE

```
x_relu = np.linspace(-10, 10, 1000)
y_relu = np.maximum(0, x_relu)
plt.figure(figsize=(10, 5))
plt.plot(x_relu, y_relu)
plt.title('ReLU Function')
plt.legend(['ReLU'])
plt.grid(True)
plt.show()
```

Result



2) Softmax 함수

소프트맥스는 로짓(logits)이라고 불리는 숫자들을 합이 1이 되는 확률로 변환하는 훌륭한 활성화 함수입니다. 소프트맥스 함수는 가능한 결과들의 확률 분포를 나타내는 벡터를 출력합니다.

$$P(y=j|z^{(i)}) = \phi(z^{(i)}) = \frac{e^{z^{(i)}}}{\sum_{j=1}^{k} e^{z_j^{(i)}}}$$

1.4.3 CNN의 합성곱 연산

합성곱(Convolution) 연산:

$$(I * K)(i,j) = \sum_m \sum_n I(i+m, j+n) \cdot K(m,n)$$

여기서 :
$-I$: 입력이미지
$-K$: 커널(필터)
$-(i,j)$: 출력특징맵의위치

1.4.4 손실 함수

희소 범주형 교차 엔트로피(Sparse Categorical Cross-Entropy)

$$L = -\sum_{i} y_i \log(\hat{y}_i)$$

여기서 :
$-y_i$: 실제레이블
$-\hat{y}_i$: 예측확률

2. 교재코드 - MLP 기반 모델

2.1 데이터 준비 및 시각화

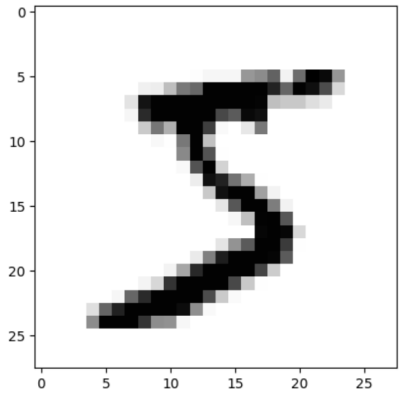
MNIST 데이터셋을 로드하고 기본적인 특성을 파악합니다.

CODE

```
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
print(train_images.shape)
print(train_labels.shape)
print(test_images.shape)
print(test_labels.shape)
plt.imshow(train_images[0], cmap='Greys')
plt.show()
```

Result

```
(60000, 28, 28)
(60000,)
(10000, 28, 28)
(10000,)
```



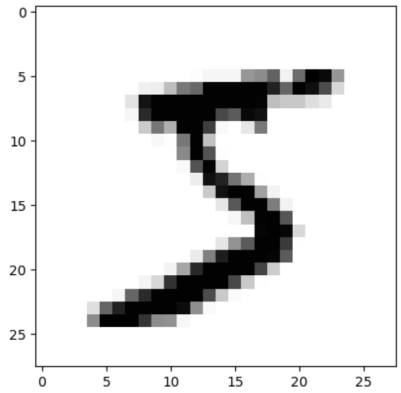
첫 번째 이미지의 픽셀 값을 출력하여 데이터 구조를 이해합니다.

CODE

```
image = train_images[0]
for i in range(28):
    for j in range(28):
        print('{:4d}'.format(image[i][j]), end='')
    print()
plt.imshow(train_images[0], cmap='Greys')
```

Result

```
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   3  18  18  18 126 136 175  26 166 255 247 127   0   0   0   0   0
   0   0   0   0   0   0   0   0  30  36  94 154 170 253 253 253 253 253 225 172 253 242 195  64   0   0   0   0
   0   0   0   0   0   0   0  49 238 253 253 253 253 253 253 253 253 251  93  82  82  56  39   0   0   0   0   0
   0   0   0   0   0   0   0  18 219 253 253 253 253 253 198 182 247 241   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0  80 156 107 253 253 205  11   0  43 154   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0  14   1 154 253  90   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0 139 253 190   2   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0  11 190 253  70   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0  35 241 225 160 108   1   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0  81 240 253 253 119  25   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0  45 186 253 253 150  27   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  16  93 252 253 187   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 249 253 249  64   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  46 130 183 253 253 207   2   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0  39 148 229 253 253 253 250 182   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0  24 114 221 253 253 253 253 201  78   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0  23  66 213 253 253 253 253 198  81   2   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0  18 171 219 253 253 253 253 195  80   9   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0  55 172 226 253 253 253 253 244 133  11   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0 136 253 253 253 212 135 132  16   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
<matplotlib.image.AxesImage at 0x7d87a5246b50>
```



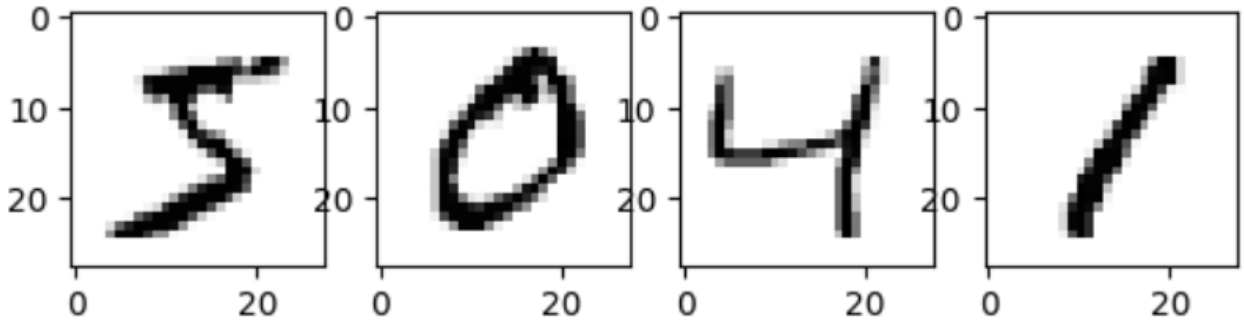
여러 이미지를 동시에 시각화하여 데이터셋의 다양성을 확인합니다.

CODE

```
fig = plt.figure()
ax1 = fig.add_subplot(1,4,1)
ax2 = fig.add_subplot(1,4,2)
ax3 = fig.add_subplot(1,4,3)
ax4 = fig.add_subplot(1,4,4)
ax1.imshow(train_images[0], cmap='Greys')
ax2.imshow(train_images[1], cmap='Greys')
ax3.imshow(train_images[2], cmap='Greys')
ax4.imshow(train_images[3], cmap='Greys')
print('train_labels[:4] =', train_labels[:4])
```

Result

```
train_labels[:4] = [5 0 4 1]
```



2.2 MLP 모델 구현 및 학습

Multi-Layer Perceptron(MLP) 개요

MLP는 가장 기본적인 신경망 구조로, 완전연결층(Dense Layer)만으로 구성됩니다. MNIST와 같은 간단한 이미지 분류 작업의 베이스라인 모델로 적합합니다.

MLP의 특징

구분	내용
구조	입력층 → 은닉층 → 출력층 (완전연결)
입력 처리	2D 이미지를 1D 벡터로 평탄화 (28×28 → 784)
레이블 처리	정수 레이블 직접 사용 [0, 1, 2, ..., 9]
손실 함수	sparse_categorical_crossentropy
장점	구현이 간단하고 빠른 학습
단점	이미지의 공간적 정보 손실

MLP에서 정수 레이블을 사용하는 이유:
1) 효율성: 메모리 사용량이 적고 처리 속도가 빠름
2) 간단함: 별도의 인코딩 과정이 불필요
3) 텐서플로우 최적화: sparse_categorical_crossentropy가 내부적으로 최적 처리

MLP의 한계:
MLP는 이미지를 1차원 벡터로 평탄화하여 처리하기 때문에 픽셀 간의 공간적 관계를 고려하지 못합니다. 이로 인해 동일한 숫자라도 위치나 크기가 변하면 인식 성능이 떨어질 수 있습니다.

다음은 기본적인 MLP 모델 구현입니다:

CODE
```
#입력값 전처리, 교재 458p
train_images_mlp = train_images / 255.0
test_images_mlp = test_images / 255.0
#신경망 모델 만들기
mlp_model = keras.Sequential([
  keras.layers.Flatten(input_shape=(28,28)),
  keras.layers.Dense(128, activation='relu'),
  keras.layers.Dense(10, activation='softmax')
])
mlp_model.summary()
#Dense 기반 일반 신경망 (MLP)
#학습시작 (Adam 옵티마이저, 희소범주형 교차 엔트로피 사용)
mlp_model.compile(optimizer='adam',
                  loss='sparse_categorical_crossentropy', #정수레이블용 손실함수
                  metrics=['accuracy'])
history_mlp = mlp_model.fit(train_images_mlp, train_labels, epochs=5, validation_split=0.1) # 검증셋 추가
mlp_model.save('/content/mlp_mnist_model.h5') #사용하기위해 저장
```

Result

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
flatten (Flatten)	(None, 784)	0
dense (Dense)	(None, 128)	100,480
dense_1 (Dense)	(None, 10)	1,290

```
Total params: 101,770 (397.54 KB)
Trainable params: 101,770 (397.54 KB)
Non-trainable params: 0 (0.00 B)
Epoch 1/5
1688/1688 ━━━━━━━━━━━━━━━━━━━━ 14s 7ms/step - accuracy: 0.8688 - loss: 0.4578 - val_accuracy: 0.9643 - val_loss: 0.1287
Epoch 2/5
1688/1688 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9623 - loss: 0.1298 - val_accuracy: 0.9710 - val_loss: 0.0939
Epoch 3/5
1688/1688 ━━━━━━━━━━━━━━━━━━━━ 7s 4ms/step - accuracy: 0.9750 - loss: 0.0849 - val_accuracy: 0.9767 - val_loss: 0.0782
Epoch 4/5
1688/1688 ━━━━━━━━━━━━━━━━━━━━ 10s 4ms/step - accuracy: 0.9831 - loss: 0.0587 - val_accuracy: 0.9805 - val_loss: 0.0735
Epoch 5/5
1688/1688 ━━━━━━━━━━━━━━━━━━━━ 9s 3ms/step - accuracy: 0.9851 - loss: 0.0474 - val_accuracy: 0.9785 - val_loss: 0.0778
```

2.3 성능 평가

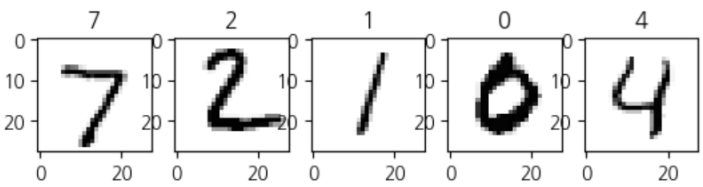
학습된 모델의 성능을 테스트 데이터셋으로 평가합니다.

CODE

```
# MLP 모델 학습 데이터와 테스트 데이터 정확도 및 손실 비교
train_loss_mlp, train_acc_mlp = mlp_model.evaluate(train_images_mlp, train_labels, verbose=0)
test_loss_mlp, test_acc_mlp = mlp_model.evaluate(test_images_mlp, test_labels, verbose=0)
print(f"MLP 모델 학습 데이터 정확도: {train_acc_mlp:.4f}")
print(f"MLP 모델 테스트 데이터 정확도: {test_acc_mlp:.4f}")
print(f"MLP 모델 학습 데이터 손실: {train_loss_mlp:.4f}")
print(f"MLP 모델 테스트 데이터 손실: {test_loss_mlp:.4f}")
fig = plt.figure()
n=5
for i in range(n):
  ax = fig.add_subplot(1,n,i+1)
  ax.imshow(test_images[i], cmap='Greys') # 원본 test_images (정규화 전) 사용
  ax.set_title(test_labels[i])
print(test_images_mlp[0][np.newaxis, :, :].shape)
print(mlp_model.predict(test_images_mlp[0][np.newaxis, :, :]))
for idx in range(5):
  y_pred = mlp_model.predict( test_images_mlp[idx][np.newaxis, :, :], verbose=0)
# 정규화된 이미지로 예측
  print(np.argmax(y_pred))
```

Result

```
MLP 모델 학습 데이터 정확도: 0.9891
MLP 모델 테스트 데이터 정확도: 0.9770
MLP 모델 학습 데이터 손실: 0.0385
MLP 모델 테스트 데이터 손실: 0.0804
(1, 28, 28)
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 62ms/step
[[1.0104480e-07 4.2397161e-08 6.8181027e-05 3.8867813e-04 6.1223552e-12
  3.1821883e-06 9.9302875e-12 9.9951410e-01 9.6892336e-06 1.6085143e-05]]
7
2
1
0
4
```



2.4 과적합 문제

MLP 모델 과적합 문제 설명

MLP 모델은 완전 연결층(Fully Connected Layer) 구조로 인해 학습 데이터에 과도하게 적합될 가능성이 있습니다. 과적합(Overfitting)이란 학습 데이터에 너무 과도하게 맞추어져, 테스트 데이터와 같은 새로운 데이터에서는 성능이 저하되는 현상을 말합니다.
본 연구에서는 에폭 수를 고정하여 학습했기 때문에 과적합 문제가 크게 나타나지 않았습니다. 아래는 학습 데이터와 테스트 데이터의 정확도를 비교하여 과적합 여부를 확인하는 코드입니다.

CODE

```
# 과적합 정도 계산
mlp_overfit_val = train_acc_mlp - test_acc_mlp
print(f"MLP 과적합 정도: {mlp_overfit_val:.4f}")
```

Result

```
MLP 과적합 정도: 0.0121
```

결과 예시

- MLP 모델 학습 데이터 정확도: 0.9891
- MLP 모델 테스트 데이터 정확도: 0.9770
- MLP 과적합 정도: 0.0121

3. MLP 모델 개선

3.1 사용자 인터페이스 구현

Gradio를 활용하여 사용자가 직접 손글씨 숫자를 그리면, 해당 이미지를 전처리하고 모델이 실시간으로 예측 및 결과를 시각화하는 기능을 구현했습니다.

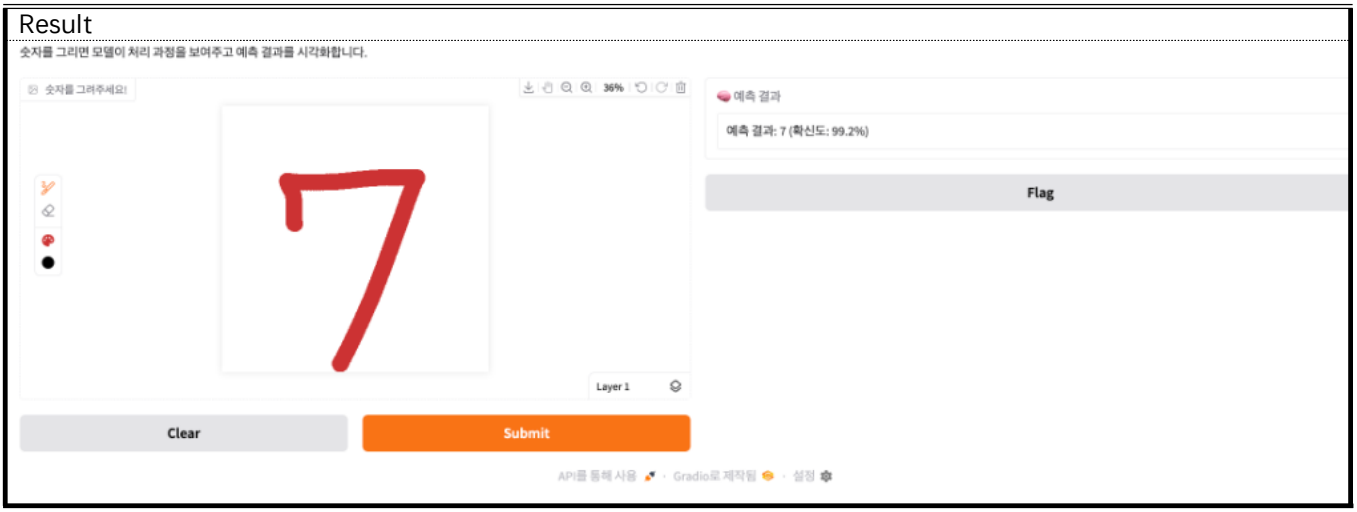
CODE

```
mlp_model = tf.keras.models.load_model("/content/mlp_mnist_model.h5")
# 예측 함수
def analyze_predict_mlp_v1(image):
    if image is None:
        return "그림을 그려주세요"
    try:
        img_data = None
        if isinstance(image, dict):
            possible_keys = ['image', 'composite', 'mask']
            for key in possible_keys:
                if key in image and isinstance(image[key], np.ndarray):
                    img_data = image[key]
                    break
            if img_data is None:
                return "오류: 유효한 이미지 데이터를 찾을 수 없습니다."
        else:
            img_data = image

        #데이터 타입 정규화
        if img_data.dtype != np.uint8:
```

```
            if img_data.max() <= 1.0:
                img_data = (img_data * 255).astype(np.uint8)
            else:
                img_data = img_data.astype(np.uint8)
        # Step 1: 흑백 변환
        img = Image.fromarray(img_data).convert("L")
        # Step 2: 색 반전 및 리사이즈
        img = ImageOps.invert(img).resize((28, 28))
        # Step 3: 정규화 및 입력
        arr = np.array(img) / 255.0
        arr = arr.reshape(1, 28, 28)
        # Step 4: 예측
        prediction = mlp_model.predict(arr, verbose=0)[0]
        predicted = np.argmax(prediction)
        confidence = np.max(prediction) * 100
        return f"예측 결과: {predicted} (확신도: {confidence:.1f}%)"
    except Exception as e:
        return f"오류 발생: {str(e)}"
# Gradio 인터페이스 정의
demo_mlp_v1 = gr.Interface(
    fn=analyze_predict_mlp_v1,
    inputs=gr.Paint(type="numpy", label="숫자를 그려주세요!"),
    outputs=gr.Textbox(label="🔍 예측 결과"),
    title="MNIST 숫자 인식기 (MLP v1 - 기본)",
    description="숫자를 그리면 모델이 처리 과정을 보여주고 예측 결과를 시각화합니다."
)
# Gradio 인터페이스 실행
demo_mlp_v1.launch(share=True)
```

Result



3.2 전처리 과정 개선 및 시각화

사용자 경험을 향상시키기 위해 입력 이미지의 전처리 과정을 개선하고 시각화 기능을 추가합니다.

전처리 개선사항

1. 바운딩 박스 검출: 사용자가 그린 숫자 영역만 자동 감지
2. 20% 패딩 추가: MNIST와 유사한 여백 비율 적용
3. 중앙 정렬: 추출된 숫자를 캔버스 중앙에 배치

이를 통해 사용자 입력을 MNIST 학습 데이터와 더 유사한 형태로 변환하여 인식 정확도를 향상시켰습니다.

CODE

```
# Gradio 헬퍼 함수 정의
def fig_to_pil_image(fig):
    buf = BytesIO()
    fig.savefig(buf, format='png')
    buf.seek(0)
    img = Image.open(buf)
    plt.close(fig)
    return img

#전처리 과정
def plot_processing_steps(original, inverted, resized):
    fig, axs = plt.subplots(1, 3, figsize=(12, 4))
    axs[0].imshow(original, cmap='gray')
    axs[0].set_title("① 원본")
    axs[0].axis("off")
    axs[1].imshow(inverted, cmap='gray')
    axs[1].set_title("② 색 반전")
    axs[1].axis("off")
    axs[2].imshow(resized, cmap='gray')
    axs[2].set_title("③ 28x28 리사이즈")
    axs[2].axis("off")
    fig.tight_layout()
    return fig_to_pil_image(fig)

# 예측 확률 그래프
def plot_prediction_bar(predictions):
    fig, ax = plt.subplots()
    bars = ax.bar(range(10), predictions, color='skyblue')
    ax.set_xticks(range(10))
    ax.set_xlabel("숫자")
    ax.set_ylabel("확률")
    ax.set_title("예측 확률 분포 (%)")
    max_idx = np.argmax(predictions)
    bars[max_idx].set_color('orange')
    for i, v in enumerate(predictions):
        ax.text(i, v + 1, f"{v:.1f}%", ha='center', fontsize=9)
    fig.tight_layout()
    return fig_to_pil_image(fig)
#메인 예측 함수:
def analyze_predict_mlp_v2(image):
    if image is None:
        return "그림을 그려주세요", None, None
    try:
        img_data = None
        # 입력이 딕셔너리 형태일 경우 (gr.Paint의 최신 출력 형태)
        if isinstance(image, dict):
            possible_keys = ['image', 'composite', 'mask']
            for key in possible_keys:
                if key in image and isinstance(image[key], np.ndarray):
                    img_data = image[key]
                    break
            if img_data is None:
                return f"오류: 입력 딕셔너리에서 유효한 이미지 데이터(NumPy 배열)를 찾을 수 없습니다.
딕셔너리 키: {list(image.keys())}", None, None

        # 데이터 타입 정규화
        if img_data.dtype != np.uint8:
```

```
            if img_data.max() <= 1.0:  # 0-1 범위
                img_data = (img_data * 255).astype(np.uint8)
            else:
                img_data = img_data.astype(np.uint8)
        # Step 1: 흑백 변환
        img = Image.fromarray(img_data).convert("L")
        # Step 2: 색 반전 (어두운 배경에 흰 글씨)
        inverted = ImageOps.invert(img)
        inverted_arr = np.array(inverted)  # NumPy 배열로 변환하여 픽셀 처리
        if np.all(inverted_arr == 0):
            return "그림을 그려주세요 (숫자가 그려지지 않았습니다)", None, None
        # 1. 숫자(흰색 픽셀)의 바운딩 박스 찾기
        coords = np.argwhere(inverted_arr > 0)
        row_min, col_min = coords.min(axis=0)
        row_max, col_max = coords.max(axis=0)
        # 2. 바운딩 박스에 맞춰 자르기 (Crop)
        cropped_arr = inverted_arr[row_min:row_max+1, col_min:col_max+1]
        # 3. 패딩 추가하여 중앙 정렬
        height, width = cropped_arr.shape
        side_length = max(height, width)
        padding = int(side_length * 0.2)  # MNIST와 유사한 20% 여백 추가
        padded_length = side_length + 2 * padding
        # 새로운 정사각형 배열 생성 (검은색 배경)
        padded_arr = np.zeros((padded_length, padded_length), dtype=np.uint8)
        # 잘라낸 숫자를 중앙에 배치
        start_row = (padded_length - height) // 2
        start_col = (padded_length - width) // 2
        padded_arr[start_row:start_row + height, start_col:start_col + width] = cropped_arr
        # 4. 28x28로 리사이즈
        resized_for_plot = Image.fromarray(padded_arr)
        resized = resized_for_plot.resize((28, 28))
        # Step 5: 정규화
        arr = np.array(resized) / 255.0
        arr = arr.reshape(1, 28, 28)
        # Step 6: 예측
        prediction = mlp_model.predict(arr, verbose=0)[0]
        predicted = np.argmax(prediction)
        confidence = np.max(prediction) * 100
        # Step 7: 시각화 이미지들 생성
        step_img = plot_processing_steps(img, inverted, resized)
        prob_chart = plot_prediction_bar(prediction * 100)
        # 예측 결과, 전처리 이미지, 확률 그래프 반환
        return f"예측 결과: {predicted} (확신도: {confidence:.1f}%)", step_img, prob_chart
    except Exception as e:
        return f"오류: 예측 처리 중 오류 발생 - {str(e)}", None, None
# □ Gradio 인터페이스 정의
demo_mlp_v2 = gr.Interface(
    fn=analyze_predict_mlp_v2,
    inputs=gr.Paint(type="numpy", label="숫자를 그려주세요!"),
    outputs=[
        gr.Textbox(label="□ 예측 결과"),
        gr.Image(type="pil", label="□ 전처리 과정"),
        gr.Image(type="pil", label="□ 예측 확률 그래프")
    ],
    title="MNIST 숫자 인식기 (MLP v2 - 시각화)",
    description="숫자를 그리면 모델이 처리 과정을 보여주고 예측 결과를 시각화합니다."
)
# Gradio 인터페이스 실행 (share=True로 외부 공유 가능)
demo_mlp_v2.launch(share=True)
```

Result

4. CNN 모델 도입 및 개선

4.1 CNN 모델 구현

Convolutional Neural Network(CNN) 개요

CNN은 이미지 처리에 특화된 신경망으로, 합성곱층과 풀링층을 통해 이미지의 공간적 특성을 보존하며 학습합니다. MLP와 달리 이미지의 위치 정보와 패턴을 효과적으로 인식할 수 있습니다.

CNN과 MLP의 주요 차이점

데이터 전처리 비교

구분	MLP	CNN
입력 형태	1D 벡터 (784개 픽셀)	2D 이미지 + 채널 (28×28×1)
레이블 형태	정수 [0, 1, 2, ..., 9]	원-핫 인코딩
손실 함수	sparse_categorical_crossentropy	categorical_crossentropy

처리 방식의 차이:
MLP: 이미지를 1차원 벡터로 변환하여 처리하므로 위치나 크기 변화에 민감하고, 공간적 관계를 인식하지 못합니다.
CNN: 2차원 이미지 형태를 유지하면서 합성곱 연산을 통해 선, 모서리, 곡선 등의 특성 패턴을 추출하여 학습합니다.

원-핫 인코딩 사용 이유:
1) 클래스 간 동등성: 모든 숫자를 동등한 범주로 처리
2) 수학적 정확성: 숫자 간 순서나 거리 관계 제거
3) 시각화 편의성: 예측 결과를 직관적으로 비교 가능

CNN 구조의 장점:

레이어	기능
합성곱층	이미지 특성 추출 (에지, 패턴 감지)
풀링층	차원 축소 및 노이즈 감소
완전연결층	최종 분류 결정

먼저 MNIST 데이터셋을 CNN 모델에 맞게 전처리합니다:

CODE

```
# MNIST 데이터셋으로 간단한 CNN 모델 학습 후 저장
# 1. 데이터 준비 - CNN 용 전처리 (MLP와의 차이점)
(x_train_cnn, y_train_cnn), (x_test_cnn, y_test_cnn) = mnist.load_data()
x_train_cnn = x_train_cnn.reshape(-1, 28, 28, 1) / 255.0  # 채널 차원 추가 (28,28,1)
x_test_cnn = x_test_cnn.reshape(-1, 28, 28, 1) / 255.0 # 채널 차원 추가 (28,28,1)
y_train_cnn = to_categorical(y_train_cnn, 10) # 원-핫 인코딩 변환 [0,1,0,0,0,0,0,0,0,0]
y_test_cnn = to_categorical(y_test_cnn, 10) # 원-핫 인코딩 변환 [0,1,0,0,0,0,0,0,0,0]
# 2. 모델 정의
# CNN 모델 변수명 cnn_model 사용
cnn_model = Sequential([ # 원본 파일은 model 변수 사용
    Conv2D(32, (3,3), activation='relu', input_shape=(28,28,1)),
    MaxPooling2D(2,2),
    Flatten(),
    Dense(100, activation='relu'),
    Dense(10, activation='softmax')
])
cnn_model.summary() # 모델요약 정보 제공
# 3. 컴파일 & 학습
cnn_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
history_cnn = cnn_model.fit(x_train_cnn, y_train_cnn, epochs=5, validation_data=(x_test_cnn,
y_test_cnn)) #OVERFITTING테스트
# 4. 저장
# CNN 모델 파일명 cnn_mnist_model.h5로 저장
cnn_model.save('/content/cnn_mnist_model.h5')
```

Result

```
Model: "sequential_1"
```

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d (MaxPooling2D)	(None, 13, 13, 32)	0
flatten_1 (Flatten)	(None, 5408)	0
dense_2 (Dense)	(None, 100)	540,900
dense_3 (Dense)	(None, 10)	1,010

```
Total params: 542,230 (2.07 MB)
Trainable params: 542,230 (2.07 MB)
Non-trainable params: 0 (0.00 B)
Epoch 1/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 36s 19ms/step - accuracy: 0.9115 - loss: 0.3049 - val_accuracy: 0.9788 - val_loss: 0.0655
Epoch 2/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 36s 19ms/step - accuracy: 0.9830 - loss: 0.0559 - val_accuracy: 0.9830 - val_loss: 0.0568
Epoch 3/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 40s 19ms/step - accuracy: 0.9896 - loss: 0.0332 - val_accuracy: 0.9836 - val_loss: 0.0500
Epoch 4/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 36s 19ms/step - accuracy: 0.9926 - loss: 0.0235 - val_accuracy: 0.9850 - val_loss: 0.0497
Epoch 5/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 40s 19ms/step - accuracy: 0.9956 - loss: 0.0145 - val_accuracy: 0.9861 - val_loss: 0.0433
```

4.2 모델 성능 평가 및 시각화

학습된 CNN 모델의 성능을 테스트 데이터셋으로 평가하고, 결과를 시각화합니다.

CODE

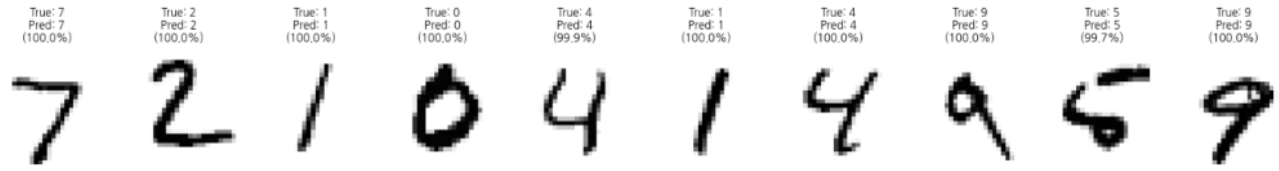
```
# 5. 테스트 코드 추가: 학습된 모델 불러와 예측 및 결과 확인
print("\n## 5. 학습된 모델 불러와 예측 및 결과 확인")
# 저장된 모델 불러오기
cnn_model = tf.keras.models.load_model("/content/cnn_mnist_model.h5")
# CNN 모델 학습 데이터와 테스트 데이터 정확도 및 손실 비교
# 로드된 모델과 4.1절의 _cnn_norm, _cnn_cat 데이터 사용
train_loss_cnn, train_acc_cnn = cnn_model.evaluate(x_train_cnn, y_train_cnn, verbose=0)
test_loss_cnn, test_acc_cnn = cnn_model.evaluate(x_test_cnn, y_test_cnn, verbose=0)
print(f"CNN 모델 학습 데이터 정확도: {train_acc_cnn:.4f}")
print(f"CNN 모델 테스트 데이터 정확도: {test_acc_cnn:.4f}")
print(f"CNN 모델 학습 데이터 손실: {train_loss_cnn:.4f}")
print(f"CNN 모델 테스트 데이터 손실: {test_loss_cnn:.4f}")
# 개별 테스트 이미지에 대한 예측 수행 및 시각화
print("\n개별 테스트 이미지 예측 결과 시각화:")
n_show = 10 # 보여줄 테스트 이미지 개수 (
plt.figure(figsize=(20, 4))
for i in range(n_show):
    img = x_test_cnn[i]
    true_label = np.argmax(y_test_cnn[i])  # 원-핫 인코딩을 정수로 변환
    if i == 0: # 첫 번째 반복에서만 출력
        print("실제 라벨들 (처음 10개): ", np.argmax(y_test_cnn[:n_show], axis=1))
        # 원-핫 인코딩을 정수로 변환
    # 로드된 cnn_model 사용
    predictions = cnn_model.predict(np.expand_dims(img, axis=0), verbose=0)
    # verbose=0으로 예측 과정 출력 숨김
    predicted_digit = np.argmax(predictions[0]) # 예측된 숫자 (가장 높은 확률을 가진 인덱스)
    confidence = np.max(predictions[0]) * 100 # 예측된 숫자의 확신도
    ax = plt.subplot(1, n_show, i + 1)
    plt.imshow(img.reshape(28, 28), cmap='Greys') # 시각화를 위해 28x28 형태로 다시 변환
    plt.title(f"True: {true_label}\nPred: {predicted_digit}\n({confidence:.1f}%)")
    plt.axis('off')
plt.tight_layout()
plt.show()
print("\n처음 10개 테스트 이미지에 대한 예측 결과:")
```

```
predictions_on_test_subset = cnn_model.predict(x_test_cnn[:10], verbose=0)
# 처음 10개 이미지에 대해 예측
predicted_digits_subset = np.argmax(predictions_on_test_subset, axis=1) # 각 이미지별 예측된 숫자
print("예측된 숫자들:", predicted_digits_subset)
print("실제 라벨들 (원-핫): ", y_test_cnn[:10]) # 원본 테스트 라벨과 비교
```

Result

```
## 5. 학습된 모델 불러와 예측 및 결과 확인
CNN 모델 학습 데이터 정확도: 0.9970
CNN 모델 테스트 데이터 정확도: 0.9861
CNN 모델 학습 데이터 손실: 0.0106
CNN 모델 테스트 데이터 손실: 0.0433

개별 테스트 이미지 예측 결과 시각화:
실제 라벨들 (처음 10개):  [7 2 1 0 4 1 4 9 5 9]
```




```
처음 10개 테스트 이미지에 대한 예측 결과:
예측된 숫자들: [7 2 1 0 4 1 4 9 5 9]
실제 라벨들 (원-핫):  [[0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]]
```

4.2.1 과적합 문제

CNN 모델 과적합 문제 설명

CNN 모델은 학습 데이터와 테스트 데이터 간의 성능 차이가 적게 나타나며, 일반적으로 과적합에 강한 특성을 가집니다. 이는 합성곱 연산과 풀링 층을 통해 데이터의 공간적 특성을 보존하며 학습하기 때문입니다.
본 연구에서는 에폭 수를 고정하여 학습했기 때문에 CNN 모델에서도 과적합 문제가 크게 나타나지 않았습니다. 아래는 CNN 모델의 과적합 여부를 확인하는 코드입니다.

CODE

```
# 과적합 정도 계산
cnn_overfit = train_acc_cnn - test_acc_cnn
print(f"CNN 과적합 정도: {cnn_overfit:.4f}")
```

Result

```
CNN 과적합 정도: 0.0109
```

결과 예시

- CNN 모델 학습 데이터 정확도: 0.9970
- CNN 모델 테스트 데이터 정확도: 0.9861
- CNN 과적합 정도: 0.0109

4.3 사용자 인터페이스 구현

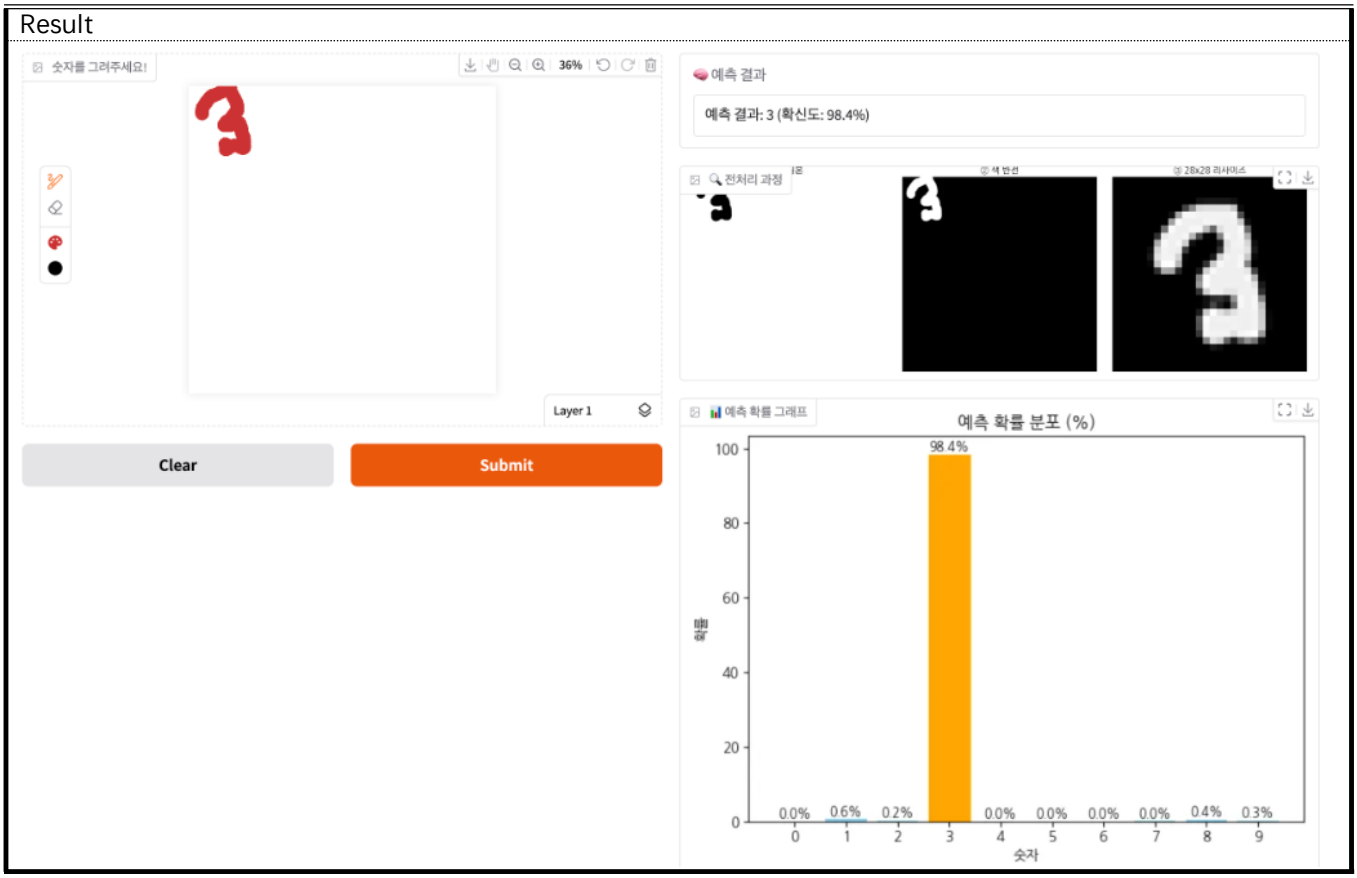
CNN 모델을 기반으로 사용자 인터페이스를 구현합니다. MLPv2 모델과 동일한 전처리 과정(중앙 정렬, 여백 처리, 패딩추가, 28x28 리사이즈)을 적용하되, CNN 모델에 맞는 입력 형태로 변환합니다.

CODE

```
# !pip install gradio
# import gradio as gr
# import numpy as np
# from PIL import Image, ImageOps
# import matplotlib.pyplot as plt
# import tensorflow as tf
#단독실행시 주석해제
#cnn_model = tf.keras.models.load_model("/content/cnn_mnist_model.h5")
# Gradio 헬퍼 함수 정의 (최초 정의)
''' def fig_to_pil_image(fig):
    buf = BytesIO()
    fig.savefig(buf, format='png')
    buf.seek(0)
    img = Image.open(buf)
    plt.close(fig)
    return img '''
#전처리 과정
''' def plot_processing_steps(original, inverted, resized):
    fig, axs = plt.subplots(1, 3, figsize=(12, 4))
    axs[0].imshow(original, cmap='gray')
    axs[0].set_title("① 원본")
    axs[0].axis("off")
    axs[1].imshow(inverted, cmap='gray')
    axs[1].set_title("② 색 반전")
    axs[1].axis("off")
    axs[2].imshow(resized, cmap='gray')
    axs[2].set_title("③ 28x28 리사이즈")
    axs[2].axis("off")
    fig.tight_layout()
    return fig_to_pil_image(fig) '''
# 예측 확률 그래프
''' def plot_prediction_bar(predictions):
    fig, ax = plt.subplots()
    bars = ax.bar(range(10), predictions, color='skyblue')
    ax.set_xticks(range(10))
    ax.set_xlabel("숫자")
    ax.set_ylabel("확률")
    ax.set_title("예측 확률 분포 (%)")
    max_idx = np.argmax(predictions)
    bars[max_idx].set_color('orange')
    for i, v in enumerate(predictions):
        ax.text(i, v + 1, f"{v:.1f}%", ha='center', fontsize=9)
    fig.tight_layout()
    return fig_to_pil_image(fig) '''
#메인 예측 함수
def analyze_predict_cnn(image):
    if image is None:
        return "그림을 그려주세요", None, None
    try:
        img_data = None
        # 입력이 딕셔너리 형태일 경우 (gr.Paint의 최신 출력 형태)
        if isinstance(image, dict):
            possible_keys = ['image', 'composite', 'mask']
            for key in possible_keys:
                if key in image and isinstance(image[key], np.ndarray):
                    img_data = image[key]
                    break
            if img_data is None:
                return f"오류: 입력 딕셔너리에서 유효한 이미지 데이터(NumPy 배열)를 찾을 수 없습니다.
딕셔너리 키: {list(image.keys())}", None, None
```

```
        # 데이터 타입 정규화
        if img_data.dtype != np.uint8:
            if img_data.max() <= 1.0:  # 0-1 범위
                img_data = (img_data * 255).astype(np.uint8)
            else:
                img_data = img_data.astype(np.uint8)
        # Step 1: 흑백 변환
        img = Image.fromarray(img_data).convert("L")
        # Step 2: 색 반전 (어두운 배경에 흰 글씨)
        inverted = ImageOps.invert(img)
        inverted_arr = np.array(inverted)  # NumPy 배열로 변환하여 픽셀 처리
        if np.all(inverted_arr == 0):
            return "그림을 그려주세요 (숫자가 그려지지 않았습니다)", None, None
        # 1. 숫자(흰색 픽셀)의 바운딩 박스 찾기
        coords = np.argwhere(inverted_arr > 0)
        row_min, col_min = coords.min(axis=0)
        row_max, col_max = coords.max(axis=0)
        # 2. 바운딩 박스에 맞춰 자르기 (Crop)
        cropped_arr = inverted_arr[row_min:row_max+1, col_min:col_max+1]
        # 3. 패딩 추가하여 중앙 정렬
        height, width = cropped_arr.shape
        side_length = max(height, width)
        padding = int(side_length * 0.2)  # MNIST와 유사한 20% 여백 추가
        padded_length = side_length + 2 * padding
        # 새로운 정사각형 배열 생성 (검은색 배경)
        padded_arr = np.zeros((padded_length, padded_length), dtype=np.uint8)
        # 잘라낸 숫자를 중앙에 배치
        start_row = (padded_length - height) // 2
        start_col = (padded_length - width) // 2
        padded_arr[start_row:start_row + height, start_col:start_col + width] = cropped_arr
        # 4. 28x28로 리사이즈
        resized_for_plot = Image.fromarray(padded_arr)
        resized = resized_for_plot.resize((28, 28))
        # Step 5: 정규화
        arr = np.array(resized) / 255.0
        arr = arr.reshape(1, 28, 28, 1)
        # Step 6: 예측
        prediction = cnn_model.predict(arr, verbose=0)[0]
        predicted = np.argmax(prediction)
        confidence = np.max(prediction) * 100
        # Step 7: 시각화 이미지들 생성
        step_img = plot_processing_steps(img, inverted, resized)
        prob_chart = plot_prediction_bar(prediction * 100)
        # 예측 결과, 전처리 이미지, 확률 그래프 반환
        return f"예측 결과: {predicted} (확신도: {confidence:.1f}%)", step_img, prob_chart
    except Exception as e:
        return f"오류: 예측 처리 중 오류 발생 - {str(e)}", None, None
# □ Gradio 인터페이스 정의
demo_cnn_final = gr.Interface(
    fn=analyze_predict_cnn,
    inputs=gr.Paint(type="numpy", label="숫자를 그려주세요!"),
    outputs=[
        gr.Textbox(label="□ 예측 결과"),
        gr.Image(type="pil", label="□ 전처리 과정"),
        gr.Image(type="pil", label="□ 예측 확률 그래프")
    ],
    title="MNIST 숫자 인식기 (CNN 최종 - 여백/중앙정렬)",
    description="숫자를 그리면 모델이 처리 과정을 보여주고 예측 결과를 시각화합니다."
)
# Gradio 인터페이스 실행 (share=True로 외부 공유 가능)
demo_cnn_final.launch(share=True)
```

Result



5. 결론 및 성능 비교

5.1 정량적 성능 비교 결과

실험 내용 - 기존 에폭 5의 결과와, 에폭을 10으로 설정하고 추가실험을 진행했습니다.

5에폭

MLP

Model: "sequential"

Layer (type)	Output Shape	Param #
flatten (Flatten)	(None, 784)	0
dense (Dense)	(None, 128)	100,480
dense_1 (Dense)	(None, 10)	1,290

Total params: 101,770 (397.54 KB)
Trainable params: 101,770 (397.54 KB)
Non-trainable params: 0 (0.00 B)

```
Epoch 1/5
1688/1688 ━━━━━━━━━━━━━━━━━━━━ 14s 7ms/step - accuracy: 0.8688 - loss: 0.4578 - val_accuracy: 0.9643 - val_loss: 0
Epoch 2/5
1688/1688 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9623 - loss: 0.1298 - val_accuracy: 0.9710 - val_loss: 0.
Epoch 3/5
1688/1688 ━━━━━━━━━━━━━━━━━━━━ 7s 4ms/step - accuracy: 0.9750 - loss: 0.0849 - val_accuracy: 0.9767 - val_loss: 0.
Epoch 4/5
1688/1688 ━━━━━━━━━━━━━━━━━━━━ 10s 4ms/step - accuracy: 0.9831 - loss: 0.0587 - val_accuracy: 0.9805 - val_loss: 0
Epoch 5/5
1688/1688 ━━━━━━━━━━━━━━━━━━━━ 9s 3ms/step - accuracy: 0.9851 - loss: 0.0474 - val_accuracy: 0.9785 - val_loss: 0.
```

MLP 모델 학습 데이터 정확도: 0.9891
MLP 모델 테스트 데이터 정확도: 0.9770
MLP 모델 학습 데이터 손실: 0.0385
MLP 모델 테스트 데이터 손실: 0.0804

MLP 과적합 정도: 0.0121

CNN

```
super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d (MaxPooling2D)	(None, 13, 13, 32)	0
flatten_1 (Flatten)	(None, 5408)	0
dense_2 (Dense)	(None, 100)	540,900
dense_3 (Dense)	(None, 10)	1,010

Total params: 542,230 (2.07 MB)
Trainable params: 542,230 (2.07 MB)
Non-trainable params: 0 (0.00 B)

```
Epoch 1/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 36s 19ms/step - accuracy: 0.9115 - loss: 0.3049 - val_accuracy: 0.9788 - val_loss: 0.0655
Epoch 2/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 36s 19ms/step - accuracy: 0.9830 - loss: 0.0559 - val_accuracy: 0.9830 - val_loss: 0.0568
Epoch 3/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 40s 19ms/step - accuracy: 0.9896 - loss: 0.0332 - val_accuracy: 0.9836 - val_loss: 0.0500
Epoch 4/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 36s 19ms/step - accuracy: 0.9926 - loss: 0.0235 - val_accuracy: 0.9850 - val_loss: 0.0497
Epoch 5/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 40s 19ms/step - accuracy: 0.9956 - loss: 0.0145 - val_accuracy: 0.9861 - val_loss: 0.0433
```

5. 학습된 모델 불러와 예측 및 결과 확인
CNN 모델 학습 데이터 정확도: 0.9970
CNN 모델 테스트 데이터 정확도: 0.9861
CNN 모델 학습 데이터 손실: 0.0106
CNN 모델 테스트 데이터 손실: 0.0433

CNN 과적합 정도: 0.0109

10에폭

MLP

Model: "sequential_2"

Layer (type)	Output Shape	Param #
flatten_2 (Flatten)	(None, 784)	0
dense_4 (Dense)	(None, 128)	100,480
dense_5 (Dense)	(None, 10)	1,290

Total params: 101,770 (397.54 KB)
Trainable params: 101,770 (397.54 KB)
Non-trainable params: 0 (0.00 B)

```
Epoch 1/10
1688/1688 ━━━━━━━━━━━━━━━━━━━━ 8s 4ms/step - accuracy: 0.8706 - loss: 0.4541 - val_accuracy: 0.9672 - val_loss: 0.1223
Epoch 2/10
1688/1688 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9629 - loss: 0.1320 - val_accuracy: 0.9742 - val_loss: 0.0883
Epoch 3/10
1688/1688 ━━━━━━━━━━━━━━━━━━━━ 7s 4ms/step - accuracy: 0.9757 - loss: 0.0829 - val_accuracy: 0.9765 - val_loss: 0.0825
Epoch 4/10
1688/1688 ━━━━━━━━━━━━━━━━━━━━ 10s 4ms/step - accuracy: 0.9815 - loss: 0.0605 - val_accuracy: 0.9737 - val_loss: 0.0926
Epoch 5/10
1688/1688 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9854 - loss: 0.0468 - val_accuracy: 0.9783 - val_loss: 0.0689
Epoch 6/10
1688/1688 ━━━━━━━━━━━━━━━━━━━━ 11s 4ms/step - accuracy: 0.9898 - loss: 0.0336 - val_accuracy: 0.9800 - val_loss: 0.0731
Epoch 7/10
1688/1688 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.9915 - loss: 0.0276 - val_accuracy: 0.9820 - val_loss: 0.0765
Epoch 8/10
1688/1688 ━━━━━━━━━━━━━━━━━━━━ 7s 4ms/step - accuracy: 0.9939 - loss: 0.0212 - val_accuracy: 0.9775 - val_loss: 0.0811
Epoch 9/10
1688/1688 ━━━━━━━━━━━━━━━━━━━━ 11s 4ms/step - accuracy: 0.9944 - loss: 0.0190 - val_accuracy: 0.9798 - val_loss: 0.0781
Epoch 10/10
1688/1688 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.9959 - loss: 0.0136 - val_accuracy: 0.9792 - val_loss: 0.0900
```

MLP 모델 학습 데이터 정확도: 0.9955
MLP 모델 테스트 데이터 정확도: 0.9779
MLP 모델 학습 데이터 손실: 0.0192
MLP 모델 테스트 데이터 손실: 0.0853

MLP 과적합 정도: 0.0176

CNN

```
Epoch 1/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 40s 21ms/step - accuracy: 0.9095 - loss: 0.3025 - val_accuracy: 0.9777 - val_loss:
Epoch 2/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 35s 19ms/step - accuracy: 0.9842 - loss: 0.0530 - val_accuracy: 0.9830 - val_loss:
Epoch 3/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 35s 19ms/step - accuracy: 0.9903 - loss: 0.0308 - val_accuracy: 0.9847 - val_loss:
Epoch 4/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 40s 18ms/step - accuracy: 0.9936 - loss: 0.0203 - val_accuracy: 0.9849 - val_loss:
Epoch 5/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 39s 21ms/step - accuracy: 0.9952 - loss: 0.0145 - val_accuracy: 0.9875 - val_loss:
Epoch 6/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 37s 20ms/step - accuracy: 0.9971 - loss: 0.0095 - val_accuracy: 0.9847 - val_loss:
Epoch 7/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 36s 19ms/step - accuracy: 0.9969 - loss: 0.0088 - val_accuracy: 0.9889 - val_loss:
Epoch 8/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 42s 20ms/step - accuracy: 0.9982 - loss: 0.0061 - val_accuracy: 0.9879 - val_loss:
Epoch 9/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 35s 19ms/step - accuracy: 0.9993 - loss: 0.0032 - val_accuracy: 0.9875 - val_loss:
Epoch 10/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 42s 20ms/step - accuracy: 0.9986 - loss: 0.0039 - val_accuracy: 0.9871 - val_loss:
```

CNN 모델 학습 데이터 정확도: 0.9983
CNN 모델 테스트 데이터 정확도: 0.9871
CNN 모델 학습 데이터 손실: 0.0050
CNN 모델 테스트 데이터 손실: 0.0591

CNN 과적합 정도: 0.0112

성능 비교

에폭 수	(5 에폭 학습 기준)			10 에폭 학습 기준		
평가 기준	MLP	CNN	차이	MLP	CNN	차이
학습 정확도	98.91%	99.70%	+0.79%	99.55%	99.83%	+0.28%
테스트 정확도	97.70%	98.61%	+0.91%	97.79%	98.71%	+0.92%
학습 손실	0.0385	0.0106	-0.0279	0.0192	0.0050	-0.0142
테스트 손실	0.0804	0.0433	-0.0371	0.0853	0.0591	-0.0262
과적합 정도	1.31%	1.09%	-0.22%	1.76%	1.12%	-0.64%

MLP / CNN 의 에폭수 변화에 따른 성능 결과 해석

1. 고성능 구간에서의 유의미성:
   97% 이상 정확도에서 CNN의 1% 우위는 모델 효율성을 반영하는 유의미한 개선이다.
2. 모델 구조의 효율성:
   이미지의 특징 학습에 특화된 CNN 아키텍처가 MLP 대비 우수한 성능과 일반화 능력을 나타냈다.
3. 일관성:
   에폭 수 변화에도 불구하고 CNN은 주요 지표에서 MLP보다 일관되게 우수한 결과를 보였다.
4. 데이터셋 특성:
   CNN은 MNIST 데이터셋의 공간적 정보를 MLP보다 효과적으로 활용하여 성능 우위를 확보했다.
5. 실용적 가치:
   CNN의 높은 정확도와 낮은 과적합은 실제 응용에서 시스템 신뢰도 향상에 기여한다.

5.2 실험을 통해 확인된 MLP와 CNN의 차이점 분석

MLP와 CNN의 성능 비교

평가 기준	MLP	CNN
학습 에폭	5, 10	5, 10
파라미터 수	상대적으로 많음	상대적으로 적음
사용자 입력 인식률	양호	우수
전처리 복잡도	단순	중간
변형된 입력에 대한 견고성	제한적	상대적으로 우수

MLP와 CNN의 구조적 차이점

특성	MLP	CNN
연결 방식	완전 연결	지역적 연결
가중치 공유	없음	있음
공간적 정보 보존	없음	있음
매개변수 효율성	낮음	높음
이미지 처리 적합성	제한적	우수
과적합 저항성	낮음	높음

5.3 연구 성과 및 주요 개선 사항

주요 연구 성과

모델 아키텍처 비교: MLP는 단순한 구조로도 구현이 용이하고 학습 데이터에 대해 높은 정확도를 보였으나, 이미지의 공간적 특성을 충분히 활용하지 못해 테스트 데이터에서는 성능 저하 및 상대적으로 높은 과적합 경향을 나타냈습니다. 반면, CNN은 합성곱 층과 풀링 층을 통해 이미지의 공간적 특성을 효과적으로 학습하여, 테스트 데이터에서 더 높은 정확도를 달성했으며 과적합 문제도 MLP에 비해 현저히 적었습니다. 손실 데이터를 포함한 전반적인 학습 안정성 측면에서도 CNN이 우위를 보였습니다.
사용자 인터페이스 개발: Gradio를 활용하여 사용자가 직접 손글씨 숫자를 입력하고 모델의 예측 결과를 실시간으로 확인할 수 있는 직관적인 그리기 인터페이스를 구현했습니다. 이를 통해 모델의 실용성을 높이고 사용자 경험을 개선했습니다.
전처리 과정 개선 및 시각화: 사용자 입력 이미지에 대해 중앙 정렬, 여백 처리, 크기 조정 등의 전처리 과정을 개선하여 MNIST 데이터셋의 형식과 유사하게 만들어 모델 입력 데이터의 품질을 향상시켰습니다. 이러한 전처리 단계와 최종 예측 확률 분포를 시각화하여 제공함으로써, 모델의 판단 과정을 사용자가 직관적으로 이해할 수 있도록 돕고 교육적 효과를 높였습니다.

사용된 기술적 처리 및 결과

사용자 입력 데이터 전처리 : 사용자 입력 데이터를 중앙 정렬하고 적절한 여백을 추가한 후, MNIST 데이터셋과 동일한 28x28 픽셀 크기로 조정하여 모델 입력 데이터의 일관성과 품질을 확보했습니다. 이는 모델이 학습 데이터와 유사한 분포의 입력을 받을 수 있도록 하여 예측 성능 향상에 기여했습니다.

오류 입력 처리 : 빈 이미지나 잘못된 형식의 입력 데이터를 감지하고 사용자에게 적절한 안내 메시지를 반환하는 오류 처리 로직을 추가하여 시스템의 안정성을 높였습니다.
시각화 기능 강화 : 입력 이미지가 원본에서부터 색상 반전, 최종 리사이즈 단계까지 어떻게 변환되는지 전처리 과정을 시각적으로 제시했습니다. 또한, 모델이 각 숫자를 예측한 확률을 막대그래프로 보여줌으로써, 모델의 판단 근거와 확신 수준을 사용자가 명확히 이해할 수 있도록 지원했습니다.
모델 비교 실험 설계 : 동일한 MNIST 데이터셋과 일관된 실험 환경 하에서 MLP와 CNN 모델을 각각 학습시키고 평가했습니다. 이를 통해 두 모델의 구조적 차이가 실제 이미지 분류 성능, 특히 공간적 특징 학습 능력과 과적합 방지 능력에서 어떠한 차이를 나타내는지 체계적으로 분석하고 비교할 수 있었습니다.

5.4 최종 결론 및 향후 방향

MNIST 손글씨 숫자 데이터셋을 활용한 본 연구를 통해, 다층 퍼셉트론(MLP)과 합성곱 신경망(CNN)이라는 두 가지 대표적인 딥러닝 모델의 구조적 특징과 실제 분류 성능을 비교 분석했습니다. 연구 결과, MLP는 간단한 구조로도 학습 데이터에 대해 높은 성능을 보였으나 이미지의 공간적 정보를 충분히 활용하지 못해 새로운 데이터(테스트 데이터)에 대한 일반화 성능이 상대적으로 낮고 과적합 경향이 더 크게 나타났습니다. 반면, CNN은 합성곱 및 풀링 연산을 통해 이미지 내 공간적 계층 구조를 효과적으로 학습함으로써, 더 적은 파라미터로도 MLP보다 우수한 테스트 정확도를 달성했으며 과적합 문제에서도 더 강인한 모습을 보였습니다.

사용자 입력 이미지를 효과적으로 처리하기 위해 중앙 정렬, 여백 추가, 크기 정규화 등의 전처리 기법을 적용하였으며, 이 과정을 시각화하여 제공했습니다. 또한, Gradio 라이브러리를 활용하여 사용자가 직접 손글씨를 입력하고 모델의 예측 결과를 실시간으로 확인할 수 있는 사용자 친화적 인터페이스를 성공적으로 구축하여 모델의 실용성을 증명했습니다.

본 연구에서 개발된 모델들은 크기 조절 및 중앙 정렬을 통해 입력 숫자의 위치 변화에는 비교적 잘 대응하였으나, 입력 이미지의 기울기가 심하거나 회전된 경우, 또는 숫자의 두께가 매우 다양하게 변하는 등 다양한 형태의 왜곡에 대해서는 인식률이 저하될 수 있다는 한계점을 가지고 있습니다. 이러한 다양한 변형에 보다 효과적으로 대응하기 위해서는 데이터 증강(Data Augmentation) 기법을 적극적으로 활용하거나, 더 복잡하고 깊은 신경망 아키텍처(예: ResNet, DenseNet 등)의 도입, 또는 이미지의 특정 부분에 집중하여 특징을 추출하는 어텐션 메커니즘(Attention Mechanism)과 같은 고급 딥러닝 기술을 적용하는 후속 연구가 필요할 수 있습니다.

그럼에도 불구하고, 이번 연구는 기초적인 딥러닝 모델을 직접 구현하고 그 성능을 체계적으로 비교 분석하며, 실제 사용자와 상호작용할 수 있는 애플리케이션으로 발전시키는 전 과정을 다루었다는 점에서 의의가 있습니다. 이는 향후 더 복잡한 실제 문제 해결을 위한 딥러닝 모델 설계 및 개발 프로젝트를 수행하는 데 있어 유용한 경험과 기초 지식을 제공할 것으로 기대됩니다.

6. 참고문헌

1. 천인국, 박동규, 강영민. (2023). 따라하며 배우는 파이썬과 데이터 과학. 생능출판.

2. LeCun, Y., Bottou, L., Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep Learning. MIT Press.

3. TensorFlow. (n.d.). TensorFlow Documentation. Retrieved from https://www.tensorflow.org/

4. Stanford University. (n.d.). CS231n: Convolutional Neural Networks for Visual Recognition. Retrieved from http://cs231n.stanford.edu/