주제: 손실 함수의 성능 비교 (평균제곱오차, 교차 엔트로피, Huber Loss, KL 발산)

선정 동기: 이 네 가지 손실 함수는 다양한 상황에서 사용될 수 있는 대표적인 손실함수 입니다.이들을 비교함으로써, 어떤 손실 함수가 분류 문제에서 더 좋은 성능을 발휘하는지를 분석할 수 있고, 각 손실 함수의 특성과 장단점을 이해할 수 있어 선정하게 되었습니다.

감자돌이

1. 라이브러리 임포트

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam
import matplotlib.pyplot as plt
```

- **numpy**: 배열 연산을 위한 라이브러리.
- **tensorflow**: 딥러닝 모델 생성 및 학습을 위한 라이브러리.
- **mnist**: 손글씨 숫자 이미지 데이터셋을 가져옵니다.
- **Sequential**: 순차적인 모델을 생성하기 위해 사용됩니다.
- **Dense**: 완전 연결층을 정의합니다.
- **Adam**: 학습을 위한 옵티마이저로, 학습률을 자동 조정합니다.
- **matplotlib.pyplot**: 학습 결과를 시각화하기 위해 사용됩니다.

2. 데이터셋 로드 및 전처리

```
# MNIST 데이터셋 로드 (훈련 데이터와 테스트 데이터 분리)
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# 입력 데이터 전처리: 이미지를 1차원 벡터로 변환하고, 정규화(0~1) 진행
x_train = x_train.reshape(60000, 784).astype('float32') / 255.0
x_test = x_test.reshape(10000, 784).astype('float32') / 255.0

# 출력 데이터(라벨)를 원-핫 인코딩
y_train = tf.keras.utils.to_categorical(y_train, 10)
y_test = tf.keras.utils.to_categorical(y_test, 10)
```

- **mnist.load_data()**: MNIST 데이터셋을 불러옵니다.
  - **x_train**: 훈련용 이미지 데이터 (60,000개, 28x28 크기).
  - **y_train**: 훈련용 레이블 (0~9의 정수 값).
  - **x_test**: 테스트용 이미지 데이터 (10,000개, 28x28 크기).
  - **y_test**: 테스트용 레이블 (0~9의 정수 값).
- **reshape()**: 이미지를 1차원 벡터(784차원)로 변환합니다.
  - **28 x 28 = 784**: 이미지의 각 픽셀 값을 하나의 벡터 요소로 변환.
- **astype('float32') / 255.0**: 정규화 진행.
  - **0~255** 사이의 픽셀 값을 **0~1** 사이 값으로 변환하여 학습 안정성을 높입니다.
- **to_categorical()**: 레이블을 **원-핫 인코딩** 형태로 변환.
  - 예를 들어, 레이블 3은 [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]으로 변환됩니다.

3. 신경망 구조 설정

```
# 각 층의 노드 수 정의
n_input = 784      # 입력층 (28x28 이미지 벡터화)
n_hidden1 = 1024   # 첫 번째 은닉층
n_hidden2 = 512    # 두 번째 은닉층
n_hidden3 = 512    # 세 번째 은닉층
n_hidden4 = 512    # 네 번째 은닉층
n_output = 10      # 출력층 (10개의 클래스)
```

- **n_input**: 입력 노드 수는 784 (28x28 이미지).
- **n_hidden1~n_hidden4**: 은닉층의 노드 수는 각기 다릅니다.
  - **첫 번째 은닉층**: 1024개 노드
  - **두 번째~네 번째 은닉층**: 각각 512개 노드
- **n_output**: 출력 노드 수는 10 (0~9까지의 클래스)

4. 모델 생성 함수 정의

```
def create_model(loss_function):
    model = Sequential()
    model.add(Dense(units=n_hidden1, activation='tanh', input_shape=(n_input,)))
    model.add(Dense(units=n_hidden2, activation='tanh'))
    model.add(Dense(units=n_hidden3, activation='tanh'))
    model.add(Dense(units=n_hidden4, activation='tanh'))
    model.add(Dense(units=n_output, activation='softmax'))
    model.compile(loss=loss_function, optimizer=Adam(learning_rate=0.0001), metrics=['accuracy'])
    return model
```

- **create_model(loss_function)**: 손실 함수를 인자로 받아 모델을 생성합니다
  - **Sequential()**: 신경망을 순차적으로 쌓는 구조입니다.
  - **Dense()**: 완전 연결층 (fully connected layer)을 정의합니다.
    - **activation='tanh'**: tanh 활성화 함수를 사용하여 비선형성을 추가합니다.
    - **input_shape=(n_input,)**: 첫 번째 층에서는 입력 크기를 지정합니다.
  - **softmax**: 출력층에서 사용하여 다중 클래스 분류 문제를 해결합니다.
  - **compile()**: 손실 함수, 옵티마이저(Adam), 평가 지표(accuracy)를 설정합니다.

5. 모델 생성 및 학습

```
# 평균제곱오차 (MSE) 손실 함수 사용
dmlp_mse = create_model(tf.keras.losses.MeanSquaredError())
hist_mse = dmlp_mse.fit(x_train, y_train, batch_size=128, epochs=30, validation_data=(x_test, y_test), verbose=2)

# 교차 엔트로피 손실 함수 사용
dmlp_ce = create_model(tf.keras.losses.CategoricalCrossentropy())
hist_ce = dmlp_ce.fit(x_train, y_train, batch_size=128, epochs=30, validation_data=(x_test, y_test), verbose=2)

# Huber 손실 함수 사용
dmlp_huber = create_model(tf.keras.losses.Huber())
hist_huber = dmlp_huber.fit(x_train, y_train, batch_size=128, epochs=30, validation_data=(x_test, y_test), verbose=2)

# KL Divergence 손실 함수 사용
dmlp_kl = create_model(tf.keras.losses.KLDivergence())
hist_kl = dmlp_kl.fit(x_train, y_train, batch_size=128, epochs=30, validation_data=(x_test, y_test), verbose=2)
```

1. Mean Squared Error (MSE) - 평균제곱오차
:예측값과 실제값의 차이를 제곱한 후, 이를 평균낸 값을 손실로 사용하는 함수

2. Categorical Crossentropy - 교차 엔트로피
:실제 레이블과 예측 확률 분포 간의 차이를 측정하는 손실 함수

3. Huber Loss - Huber 손실 함수
:MSE와 MAE(Mean Absolute Error)의 장점을 결합한 손실 함수

4. Kullback-Leibler Divergence (KL Divergence) - KL 발산
:두 확률 분포 간의 차이를 측정하는 함수

비교 요약

손실 함수	주요 사용 분야	장점	단점
Mean Squared Error	회귀	간단하고 계산이 빠름	이상치에 민감함
Categorical Crossentropy	분류	확률 기반 분류 문제에 적합	확률값 예측이 필요함
Huber Loss	회귀	이상치에 강건함	δ 값을 설정해야 함
KL Divergence	분포 비교	확률 분포 차이를 명확하게 측정	계산 복잡성 증가

6.모델 평가 및 결과 출력

```
# 모델 평가
res_mse = dmlp_mse.evaluate(x_test, y_test, verbose=0)
res_ce = dmlp_ce.evaluate(x_test, y_test, verbose=0)
res_huber = dmlp_huber.evaluate(x_test, y_test, verbose=0)
res_kl = dmlp_kl.evaluate(x_test, y_test, verbose=0)

print(f'평균제곱오차의 정확률: {res_mse[1] * 100:.2f}%')
print(f'교차 엔트로피의 정확률: {res_ce[1] * 100:.2f}%')
print(f'Huber Loss의 정확률: {res_huber[1] * 100:.2f}%')
print(f'KL Divergence Loss의 정확률: {res_kl[1] * 100:.2f}%')
```

- **evaluate()**: 테스트 데이터를 사용해 모델의 성능을 평가합니다.
- **res_*[1]**: 정확도(accuracy) 값을 출력합니다.

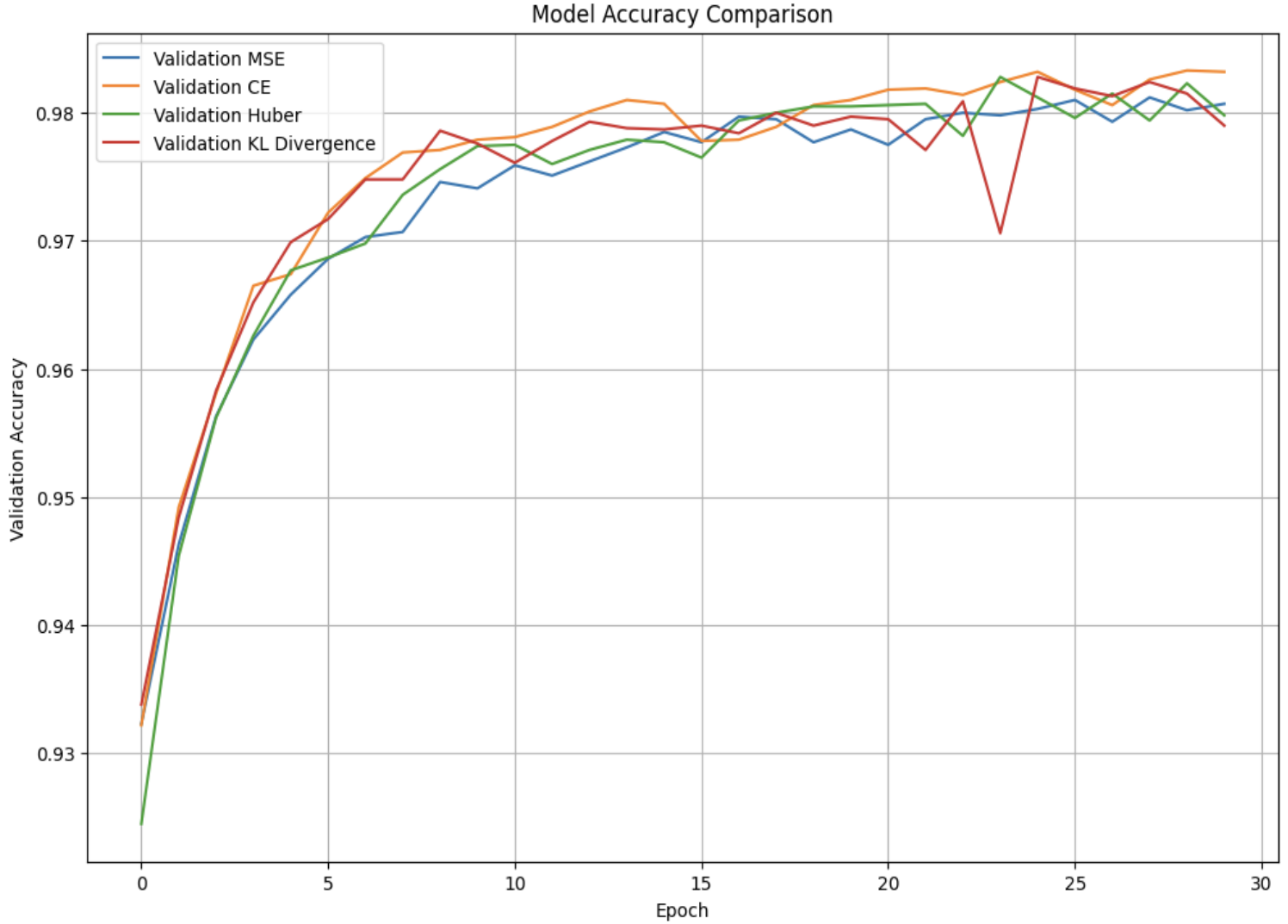
7.박스플롯 시각화

```
# 성능 비교 그래프
plt.figure(figsize=(12, 8))
plt.plot(hist_mse.history['val_accuracy'], label='Validation MSE')
plt.plot(hist_ce.history['val_accuracy'], label='Validation CE')
plt.plot(hist_huber.history['val_accuracy'], label='Validation Huber')
plt.plot(hist_kl.history['val_accuracy'], label='Validation KL Divergence')
plt.title('Model Accuracy Comparison')
plt.xlabel('Epoch')
plt.ylabel('Validation Accuracy')
plt.legend
```

<실행 결과>

```
Epoch 1/30
469/469 - 24s - 50ms/step - accuracy: 0.8942 - loss: 0.0162 - val_accuracy: 0.9323 - val_loss: 0.0106
Epoch 2/30
469/469 - 40s - 85ms/step - accuracy: 0.9372 - loss: 0.0097 - val_accuracy: 0.9463 - val_loss: 0.0083
Epoch 3/30
469/469 - 43s - 91ms/step - accuracy: 0.9523 - loss: 0.0075 - val_accuracy: 0.9563 - val_loss: 0.0067
Epoch 4/30
469/469 - 43s - 93ms/step - accuracy: 0.9617 - loss: 0.0060 - val_accuracy: 0.9623 - val_loss: 0.0059
Epoch 5/30
469/469 - 21s - 46ms/step - accuracy: 0.9688 - loss: 0.0050 - val_accuracy: 0.9658 - val_loss: 0.0052
Epoch 6/30
469/469 - 22s - 47ms/step - accuracy: 0.9738 - loss: 0.0042 - val_accuracy: 0.9686 - val_loss: 0.0049
Epoch 7/30
469/469 - 39s - 84ms/step - accuracy: 0.9781 - loss: 0.0036 - val_accuracy: 0.9703 - val_loss: 0.0045
Epoch 8/30
469/469 - 41s - 88ms/step - accuracy: 0.9808 - loss: 0.0032 - val_accuracy: 0.9707 - val_loss: 0.0044
Epoch 9/30
469/469 - 42s - 90ms/step - accuracy: 0.9839 - loss: 0.0027 - val_accuracy: 0.9746 - val_loss: 0.0040
Epoch 10/30
469/469 - 41s - 87ms/step - accuracy: 0.9858 - loss: 0.0024 - val_accuracy: 0.9741 - val_loss: 0.0039
Epoch 11/30
469/469 - 41s - 88ms/step - accuracy: 0.9881 - loss: 0.0021 - val_accuracy: 0.9759 - val_loss: 0.0037
Epoch 12/30
469/469 - 39s - 84ms/step - accuracy: 0.9898 - loss: 0.0018 - val_accuracy: 0.9751 - val_loss: 0.0037
Epoch 13/30
469/469 - 41s - 87ms/step - accuracy: 0.9912 - loss: 0.0016 - val_accuracy: 0.9762 - val_loss: 0.0035
Epoch 14/30
469/469 - 42s - 90ms/step - accuracy: 0.9920 - loss: 0.0014 - val_accuracy: 0.9773 - val_loss: 0.0034
Epoch 15/30
469/469 - 41s - 87ms/step - accuracy: 0.9934 - loss: 0.0012 - val_accuracy: 0.9785 - val_loss: 0.0032
Epoch 16/30
469/469 - 40s - 86ms/step - accuracy: 0.9940 - loss: 0.0011 - val_accuracy: 0.9777 - val_loss: 0.0034
Epoch 17/30
469/469 - 40s - 85ms/step - accuracy: 0.9950 - loss: 9.5544e-04 - val_accuracy: 0.9797 - val_loss: 0.0031
Epoch 18/30
469/469 - 22s - 47ms/step - accuracy: 0.9955 - loss: 8.5453e-04 - val_accuracy: 0.9795 - val_loss: 0.0030
Epoch 19/30
469/469 - 41s - 88ms/step - accuracy: 0.9957 - loss: 8.2182e-04 - val_accuracy: 0.9777 - val_loss: 0.0032
Epoch 20/30
469/469 - 41s - 87ms/step - accuracy: 0.9968 - loss: 6.6389e-04 - val_accuracy: 0.9787 - val_loss: 0.0031
Epoch 21/30
469/469 - 40s - 86ms/step - accuracy: 0.9962 - loss: 7.1232e-04 - val_accuracy: 0.9775 - val_loss: 0.0034
Epoch 22/30
469/469 - 40s - 86ms/step - accuracy: 0.9970 - loss: 5.9388e-04 - val_accuracy: 0.9795 - val_loss: 0.0031
Epoch 23/30
469/469 - 43s - 93ms/step - accuracy: 0.9975 - loss: 5.0735e-04 - val_accuracy: 0.9800 - val_loss: 0.0031
Epoch 24/30
469/469 - 40s - 85ms/step - accuracy: 0.9979 - loss: 4.3076e-04 - val_accuracy: 0.9798 - val_loss: 0.0031
Epoch 25/30
469/469 - 41s - 87ms/step - accuracy: 0.9976 - loss: 4.8646e-04 - val_accuracy: 0.9803 - val_loss: 0.0030
Epoch 26/30
469/469 - 41s - 87ms/step - accuracy: 0.9980 - loss: 4.0568e-04 - val_accuracy: 0.9810 - val_loss: 0.0028
Epoch 27/30
469/469 - 40s - 84ms/step - accuracy: 0.9982 - loss: 3.7851e-04 - val_accuracy: 0.9793 - val_loss: 0.0030
Epoch 28/30
469/469 - 22s - 46ms/step - accuracy: 0.9974 - loss: 4.8131e-04 - val_accuracy: 0.9812 - val_loss: 0.0030
Epoch 29/30
469/469 - 41s - 87ms/step - accuracy: 0.9979 - loss: 4.1878e-04 - val_accuracy: 0.9802 - val_loss: 0.0030
```

```
평균제곱오차의 정확률: 98.07%
교차 엔트로피의 정확률: 98.32%
Huber Loss의 정확률: 97.98%
KL Divergence Loss의 정확률: 97.90%
```



네 가지 손실함수 epoch 30번씩 학습(사진은 mse)

<실행 결과 분석>

주요 결과 요약:

1.MSE (Mean Squared Error):

최종 정확도: **98.07%**

초기 학습 속도는 다소 느리지만, 안정적인 성능을 보여줍니다.

다른 손실 함수들에 비해 약간 낮은 최종 정확도.

2.Cross Entropy (교차 엔트로피):

최종 정확도: **98.32%**

다른 손실 함수들보다 빠르게 수렴하며, 최종 정확도에서도 가장 높은 성능을 보입니다.

특히 초기 에포크에서 높은 성능을 기록하며 안정적으로 유지됩니다.

3.Huber Loss:

최종 정확도: **97.98%**

중간에 다른 손실 함수들보다 정확도가 낮아지는 구간이 있지만, 후반부에서는 안정적인 성능을 보입니다.

극단값에 강건한 특성이 있어 일부 데이터에서 좋은 성능을 보일 가능성.

4.KL Divergence Loss:

최종 정확도: **97.90%**

에포크 10 이후 급격히 상승하지만, 최종 정확도는 다른 손실 함수들보다 약간 낮습니다.

비슷한 분포를 가정한 경우 성능이 높을 수 있으나, 이 경우 다소 낮은 성능을 보임.

분석:

-교차 엔트로피 (CE)가 가장 높은 최종 정확도(98.32%)를 보이며, 다른 손실 함수들에 비해 안정적인 학습 과정을 나타냅니다.

-MSE는 비교적 안정적이지만, 초기 학습 속도가 느리고 최종 성능이 교차 엔트로피보다 다소 낮습니다.

-Huber 손실은 중간에 다소 불안정한 모습을 보였으나, 후반부에서 회복되었습니다. 이는 이상치가 많은 데이터에서 유리할 가능성이 있습니다.

-KL Divergence는 다소 불안정한 학습을 보이며, 특히 초반과 중반 학습에서 변동이 심합니다.

결론:

-안정적이고 높은 성능을 원할 경우 **Cross Entropy**가 가장 적합해 보입니다.

-데이터에 이상치(outlier)가 포함된 경우라면 **Huber Loss**를 고려할 수 있습니다.

-MSE는 비교적 안정적이지만, 정확도에서는 다소 아쉬운 성능을 보였습니다.

-KL Divergence는 특수한 분포 가정을 하는 경우에 성능이 더 높아질 수 있으나, 이번 데이터셋에서는 상대적으로 낮은 정확도를 기록했습니다.

참고 문헌

- ‘파이썬으로 만드는 인공지능‘ 5장
  [프로그램 5-10] –손실 함수의 성능 비교: 평균제곱오차와 교차 엔트로피