간단한 RNN 구현 예시

학습 목표

- RNN 기본 코드를 구현해 봅니다. 이를 통해 기본 개념을 이해합니다.

목차

01. 간단한 RNN 모델

목차로 이동하기

In [5]:
```
import torch
import torch.nn as nn

# 간단한 RNN 모델 정의
class SimpleRNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(SimpleRNN, self).__init__()
        self.rnn = nn.RNN(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        rnn_out, hidden = self.rnn(x)  # RNN 통과 후 은닉 상태와 출력 반환
        print(type(rnn_out), type(hidden))
        print("은닉상태 : ", rnn_out.shape, "마지막 은닉(출력) 상태 : ", hidden.s
        out = self.fc(rnn_out[:, -1, :])  # 마지막 시간 단계의 출력을 선형 레이어
        return out

# 파라미터 설정
input_size = 10  # 입력 벡터의 크기
hidden_size = 20  # 은닉 상태의 크기 (각 시간 단계에서의 20차원 벡)
output_size = 1  # 최종 출력 크기 (예: 회귀 문제에서는 1차원)

# 모델 초기화
model = SimpleRNN(input_size, hidden_size, output_size)

# 임의의 입력 데이터 생성 (배치 크기: 16, 시퀀스 길이: 3, 입력 차원: 10)
# 시퀀스의 길이는 셀이 3단계가 있다는 의미
input_data = torch.randn(16, 3, input_size)  # (배치 크기, 시퀀스 길이, 입력 차원

# 모델 실행
output = model(input_data)

print("Output shape:", output.shape)  # 출력 모양 출력 (배치 크기, 출력 차원)
```

```
<class 'torch.Tensor'> <class 'torch.Tensor'>
은닉상태 :  torch.Size([16, 3, 20]) 출력 상태 :  torch.Size([1, 16, 20])
Output shape: torch.Size([16, 1])
```

In [17]:
```python
import torch
import torch.nn as nn
import torch.optim as optim

# RNN 모델 정의
class RNNModel(nn.Module):
    def __init__(self):
        super(RNNModel, self).__init__()
        self.rnn = nn.RNN(input_size=10, hidden_size=20, num_layers=2)
        self.fc = nn.Linear(20, 1)  # 출력층 (예: 출력 차원이 1인 경우)

    def forward(self, x):
        out, _ = self.rnn(x)
        out = self.fc(out)  # RNN 출력 후 선형층 추가
        return out

model = RNNModel()

# 최적화 알고리즘 설정 (Adam, RMSprop 등)
optimizer = optim.Adam(model.parameters(), lr=0.001)

# 손실 함수
criterion = nn.MSELoss()

# 임의의 input_data와 target_data 생성 (배치 크기: 5, 시퀀스 길이: 3, 입력 크기:
input_data = torch.randn(3, 5, 10)  # (시퀀스 길이, 배치 크기, 입력 크기)
target_data = torch.randn(3, 5, 1)  # 타겟 값 (시퀀스 길이, 배치 크기, 출력 크기)

# 학습 루프
for epoch in range(100):
    optimizer.zero_grad()  # 기울기 초기화
    output = model(input_data)  # 모델에 데이터 입력
    loss = criterion(output, target_data)  # 손실 계산
    loss.backward()  # 역전파
    optimizer.step()  # 파라미터 업데이트

    if epoch % 10 == 0:  # 10번마다 손실 출력
        print(f'Epoch {epoch}, Loss: {loss.item()}')
```

```
Epoch 0, Loss: 0.5494986176490784
Epoch 10, Loss: 0.3872861862182617
Epoch 20, Loss: 0.26707836985588074
Epoch 30, Loss: 0.1744249314069748
Epoch 40, Loss: 0.11185114085674286
Epoch 50, Loss: 0.07890104502439499
Epoch 60, Loss: 0.0633840411901474
Epoch 70, Loss: 0.05202016606926918
Epoch 80, Loss: 0.0428352989256382
Epoch 90, Loss: 0.035718366503715515
```

02. RNN을 이용한 긍정, 부정 감정 분석

목차로 이동하기

In [6]:
```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, Dataset

# 데이터셋 클래스 정의
class SimpleTextDataset(Dataset):
    def __init__(self, texts, labels, vocab):
        self.texts = texts
        self.labels = labels
        self.vocab = vocab

    def __len__(self):
        return len(self.texts)

    def __getitem__(self, idx):
        text = self.texts[idx]
        label = self.labels[idx]

        # 텍스트를 인덱스로 변환
        text_tensor = torch.tensor([self.vocab[word] for word in text.split()],
        return text_tensor, torch.tensor(label, dtype=torch.long)
```

In [18]:
```
# 간단한 단어 사전 생성
vocab = {"I": 0, "love": 1, "this": 2, "movie": 3, "hate": 4, "film": 5, "is": 6

# 문장 데이터
texts = [
    "I love this movie",   # 긍정
    "I hate this movie",   # 부정
    "this film is great",  # 긍정
    "this film is terrible" # 부정
]
labels = [1, 0, 1, 0]  # 1: 긍정, 0: 부정

# 데이터셋 초기화
dataset = SimpleTextDataset(texts, labels, vocab)

# 데이터 로더 - 한번에 데이터를 몇개 제공할지 결정.
dataloader = DataLoader(dataset, batch_size=2, collate_fn=lambda batch: (
    nn.utils.rnn.pad_sequence([item[0] for item in batch], batch_first=True),
    torch.tensor([item[1] for item in batch])
))
```

In [14]:
```
# 간단한 텍스트 RNN 모델 정의
class SimpleTextRNN(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_size, output_size):
        super(SimpleTextRNN, self).__init__()
        # 단어 임베딩 레이어
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        # RNN 레이어
        self.rnn = nn.RNN(embedding_dim, hidden_size, batch_first=True)

        # 출력 레이어
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = self.embedding(x)  # 입력을 임베딩으로 변환
        rnn_out, hidden = self.rnn(x)  # RNN 통과
```

```
        print("입력 임베딩 : ", x.shape)
        # rnn_out : (배치크기, 시퀀스 길이(각문장의 단어수), 각 타임스텝의 16차원
        # hidden : (레이어수, 배치크기, 은닉상태의 크기
        print("각 시간 단계의 RNN출력 : ", rnn_out.shape, "\nRNN의 마지막 시간 단

        out = self.fc(rnn_out[:, -1, :])  # 마지막 타임스텝의 출력만 사용
        return out

# 파라미터 설정
vocab_size = len(vocab)  # 어휘 크기
embedding_dim = 10  # 임베딩 차원
hidden_size = 16  # 은닉 상태 크기
output_size = 2  # 출력 크기 (긍정/부정 이진 분류)

print("어휘 크기 : ", vocab_size)
print("임베딩 차원 : ", embedding_dim)
print("은닉 상태 크기 : ", hidden_size)
print("출력 크기 : ", output_size)


# 모델 초기화
model = SimpleTextRNN(vocab_size, embedding_dim, hidden_size, output_size)

# 최적화 알고리즘과 손실 함수 설정
optimizer = optim.Adam(model.parameters(), lr=0.001)
criterion = nn.CrossEntropyLoss()

# 학습 루프
for epoch in range(10):  # 10 에포크 동안 학습
    model.train()
    total_loss = 0
    for text_tensor, labels in dataloader:
        optimizer.zero_grad()
        output = model(text_tensor)  # 모델에 입력
        loss = criterion(output, labels)  # 손실 계산
        loss.backward()  # 역전파
        optimizer.step()  # 가중치 업데이트
        total_loss += loss.item()

    print(f'Epoch {epoch+1}, Loss: {total_loss/len(dataloader)}')
```

```
어휘 크기 :  10
임베딩 차원 :  10
은닉 상태 크기 :  16
출력 크기 :  2
입력 임베딩 :  torch.Size([2, 4, 10])
각 시간 단계의 RNN출력 :  torch.Size([2, 4, 16])
RNN의 마지막 시간 단계에서의 은닉 상태 : torch.Size([1, 2, 16])
입력 임베딩 :  torch.Size([2, 4, 10])
각 시간 단계의 RNN출력 :  torch.Size([2, 4, 16])
RNN의 마지막 시간 단계에서의 은닉 상태 : torch.Size([1, 2, 16])
Epoch 1, Loss: 0.6748062670230865
입력 임베딩 :  torch.Size([2, 4, 10])
각 시간 단계의 RNN출력 :  torch.Size([2, 4, 16])
RNN의 마지막 시간 단계에서의 은닉 상태 : torch.Size([1, 2, 16])
입력 임베딩 :  torch.Size([2, 4, 10])
각 시간 단계의 RNN출력 :  torch.Size([2, 4, 16])
RNN의 마지막 시간 단계에서의 은닉 상태 : torch.Size([1, 2, 16])
Epoch 2, Loss: 0.6635642051696777
입력 임베딩 :  torch.Size([2, 4, 10])
각 시간 단계의 RNN출력 :  torch.Size([2, 4, 16])
RNN의 마지막 시간 단계에서의 은닉 상태 : torch.Size([1, 2, 16])
입력 임베딩 :  torch.Size([2, 4, 10])
각 시간 단계의 RNN출력 :  torch.Size([2, 4, 16])
RNN의 마지막 시간 단계에서의 은닉 상태 : torch.Size([1, 2, 16])
Epoch 3, Loss: 0.6533559262752533
입력 임베딩 :  torch.Size([2, 4, 10])
각 시간 단계의 RNN출력 :  torch.Size([2, 4, 16])
RNN의 마지막 시간 단계에서의 은닉 상태 : torch.Size([1, 2, 16])
입력 임베딩 :  torch.Size([2, 4, 10])
각 시간 단계의 RNN출력 :  torch.Size([2, 4, 16])
RNN의 마지막 시간 단계에서의 은닉 상태 : torch.Size([1, 2, 16])
Epoch 4, Loss: 0.6433069705963135
입력 임베딩 :  torch.Size([2, 4, 10])
각 시간 단계의 RNN출력 :  torch.Size([2, 4, 16])
RNN의 마지막 시간 단계에서의 은닉 상태 : torch.Size([1, 2, 16])
입력 임베딩 :  torch.Size([2, 4, 10])
각 시간 단계의 RNN출력 :  torch.Size([2, 4, 16])
RNN의 마지막 시간 단계에서의 은닉 상태 : torch.Size([1, 2, 16])
Epoch 5, Loss: 0.6333291232585907
입력 임베딩 :  torch.Size([2, 4, 10])
각 시간 단계의 RNN출력 :  torch.Size([2, 4, 16])
RNN의 마지막 시간 단계에서의 은닉 상태 : torch.Size([1, 2, 16])
입력 임베딩 :  torch.Size([2, 4, 10])
각 시간 단계의 RNN출력 :  torch.Size([2, 4, 16])
RNN의 마지막 시간 단계에서의 은닉 상태 : torch.Size([1, 2, 16])
Epoch 6, Loss: 0.623369574546814
입력 임베딩 :  torch.Size([2, 4, 10])
각 시간 단계의 RNN출력 :  torch.Size([2, 4, 16])
RNN의 마지막 시간 단계에서의 은닉 상태 : torch.Size([1, 2, 16])
입력 임베딩 :  torch.Size([2, 4, 10])
각 시간 단계의 RNN출력 :  torch.Size([2, 4, 16])
RNN의 마지막 시간 단계에서의 은닉 상태 : torch.Size([1, 2, 16])
Epoch 7, Loss: 0.6133836209774017
입력 임베딩 :  torch.Size([2, 4, 10])
각 시간 단계의 RNN출력 :  torch.Size([2, 4, 16])
RNN의 마지막 시간 단계에서의 은닉 상태 : torch.Size([1, 2, 16])
입력 임베딩 :  torch.Size([2, 4, 10])
각 시간 단계의 RNN출력 :  torch.Size([2, 4, 16])
RNN의 마지막 시간 단계에서의 은닉 상태 : torch.Size([1, 2, 16])
Epoch 8, Loss: 0.6033314168453217
```

```
입력 임베딩 :  torch.Size([2, 4, 10])
각 시간 단계의 RNN출력 :  torch.Size([2, 4, 16])
RNN의 마지막 시간 단계에서의 은닉 상태 : torch.Size([1, 2, 16])
입력 임베딩 :  torch.Size([2, 4, 10])
각 시간 단계의 RNN출력 :  torch.Size([2, 4, 16])
RNN의 마지막 시간 단계에서의 은닉 상태 : torch.Size([1, 2, 16])
Epoch 9, Loss: 0.5931777060031891
입력 임베딩 :  torch.Size([2, 4, 10])
각 시간 단계의 RNN출력 :  torch.Size([2, 4, 16])
RNN의 마지막 시간 단계에서의 은닉 상태 : torch.Size([1, 2, 16])
입력 임베딩 :  torch.Size([2, 4, 10])
각 시간 단계의 RNN출력 :  torch.Size([2, 4, 16])
RNN의 마지막 시간 단계에서의 은닉 상태 : torch.Size([1, 2, 16])
Epoch 10, Loss: 0.5828917920589447
```

실습 과제

- [실습 1] hidden_size를 8,32로 변경해 보자. 모델의 성능에 어떤 영향을 미칠까? RNN 모델의 하이퍼파라미터(예: hidden_size, embedding_dim, batch_size, learning_rate 등)를 변경해보기. 모델의 성능에 어떤 영향을 미칠까?

*[실습 2] 다층 RNN 실습 레이어 수를 늘려 다층 RNN으로 실습해 보자. num_layers=2로 설정하여 성능 차이 분석

*[실습 3] 활성화 함수 변경 활성화 함수로 ReLU, Tanh, Sigmoid 등을 실험해 보자 어떤 성능 변화가 있을까?

*[실습 4] 손실 함수 변경 실습 CrossEntropyLoss 대신 MSELoss를 적용해보고, 이진 분류에서 손실 함수 선택의 중요성을 실습해보자. 어떤 성능 변화가 있을까?

03. [실습 2] 다층 RNN 실습: num_layers=2로 설정하여 성능 차이를 분석

목차로 이동하기

- self.rnn = nn.RNN(embedding_dim, hidden_size, num_layers=num_layers, batch_first=True)

In [19]:
```python
import torch
import torch.nn as nn
import torch.optim as optim

# 다층 RNN 모델 정의
class MultiLayerRNN(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_size, output_size, num_
        super(MultiLayerRNN, self).__init__()
        # 단어 임베딩 레이어
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        # 다층 RNN 레이어
        self.rnn = nn.RNN(embedding_dim, hidden_size, num_layers=num_layers, bat
        # 출력 레이어
        self.fc = nn.Linear(hidden_size, output_size)
```

```
    def forward(self, x):
        x = self.embedding(x)  # 입력을 임베딩으로 변환
        rnn_out, hidden = self.rnn(x)  # 다층 RNN 통과
        out = self.fc(rnn_out[:, -1, :])  # 마지막 타임스텝의 출력만 사용
        return out

# 파라미터 설정
vocab_size = 10000
embedding_dim = 10
hidden_size = 16
output_size = 2

# 모델 초기화 (다층 RNN)
model = MultiLayerRNN(vocab_size, embedding_dim, hidden_size, output_size, num_l

# 최적화 알고리즘과 손실 함수 설정
optimizer = optim.Adam(model.parameters(), lr=0.001)
criterion = nn.CrossEntropyLoss()

# 학습 루프
for epoch in range(10):  # 10 에포크 동안 학습
    model.train()
    total_loss = 0
    for text_tensor, labels in dataloader:
        optimizer.zero_grad()
        output = model(text_tensor)  # 모델에 입력
        loss = criterion(output, labels)  # 손실 계산
        loss.backward()  # 역전파
        optimizer.step()  # 가중치 업데이트
        total_loss += loss.item()

    print(f'Epoch {epoch+1}, Loss: {total_loss/len(dataloader)}')
```

```
Epoch 1, Loss: 0.7444477081298828
Epoch 2, Loss: 0.723354160785675
Epoch 3, Loss: 0.7046844959259033
Epoch 4, Loss: 0.6870203614234924
Epoch 5, Loss: 0.6702257990837097
Epoch 6, Loss: 0.6541889607906342
Epoch 7, Loss: 0.6387788951396942
Epoch 8, Loss: 0.623837798833847
Epoch 9, Loss: 0.6091840863227844
Epoch 10, Loss: 0.594628095626831
```

04. [실습 3] 활성화 함수 변경 실습: ReLU, Tanh, Sigmoid 등을 실험

목차로 이동하기

In [22]:
```
import torch
import torch.nn as nn
import torch.optim as optim

# 활성화 함수를 포함한 RNN 모델 정의
class RNNWithActivation(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_size, output_size, acti
        super(RNNWithActivation, self).__init__()
        # 단어 임베딩 레이어
```

```
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        # RNN 레이어
        self.rnn = nn.RNN(embedding_dim, hidden_size, batch_first=True)
        # 출력 레이어
        self.fc = nn.Linear(hidden_size, output_size)

        # 활성화 함수 선택
        if activation_function == 'ReLU':
            self.activation = nn.ReLU()
        elif activation_function == 'Tanh':
            self.activation = nn.Tanh()
        elif activation_function == 'Sigmoid':
            self.activation = nn.Sigmoid()
        else:
            raise ValueError("Invalid activation function")

    def forward(self, x):
        x = self.embedding(x)  # 입력을 임베딩으로 변환
        rnn_out, hidden = self.rnn(x)  # RNN 통과
        out = self.fc(rnn_out[:, -1, :])  # 마지막 타임스텝의 출력만 사용
        out = self.activation(out)  # 활성화 함수 적용
        return out

# 파라미터 설정
vocab_size = 10000
embedding_dim = 10
hidden_size = 16
output_size = 2

act_list = ['ReLU', 'Tanh', 'Sigmoid']

for one_act in act_list:
  # 모델 초기화 (활성화 함수 ReLU 사용)
  model = RNNWithActivation(vocab_size, embedding_dim, hidden_size, output_size,

  # 최적화 알고리즘과 손실 함수 설정
  optimizer = optim.Adam(model.parameters(), lr=0.001)
  criterion = nn.CrossEntropyLoss()

  print("활성화 함수 :", one_act)
  # 학습 루프
  for epoch in range(10):  # 10 에포크 동안 학습
      model.train()
      total_loss = 0
      for text_tensor, labels in dataloader:
          optimizer.zero_grad()
          output = model(text_tensor)  # 모델에 입력
          loss = criterion(output, labels)  # 손실 계산
          loss.backward()  # 역전파
          optimizer.step()  # 가중치 업데이트
          total_loss += loss.item()

      print(f'Epoch {epoch+1}, Loss: {total_loss/len(dataloader)}')
```

```
활성화 함수 : ReLU
Epoch 1, Loss: 0.6931471824645996
Epoch 2, Loss: 0.6931471824645996
Epoch 3, Loss: 0.6931471824645996
Epoch 4, Loss: 0.6931471824645996
Epoch 5, Loss: 0.6931471824645996
Epoch 6, Loss: 0.6931471824645996
Epoch 7, Loss: 0.6931471824645996
Epoch 8, Loss: 0.6931471824645996
Epoch 9, Loss: 0.6931471824645996
Epoch 10, Loss: 0.6931471824645996
활성화 함수 : Tanh
Epoch 1, Loss: 0.6978185176849365
Epoch 2, Loss: 0.6881099939346313
Epoch 3, Loss: 0.6795485019683838
Epoch 4, Loss: 0.6715735197067261
Epoch 5, Loss: 0.6641144156455994
Epoch 6, Loss: 0.6571040153503418
Epoch 7, Loss: 0.6504657864570618
Epoch 8, Loss: 0.6441184878349304
Epoch 9, Loss: 0.6379789710044861
Epoch 10, Loss: 0.631966233253479
활성화 함수 : Sigmoid
Epoch 1, Loss: 0.6977046728134155
Epoch 2, Loss: 0.6950433850288391
Epoch 3, Loss: 0.6924891769886017
Epoch 4, Loss: 0.6899415850639343
Epoch 5, Loss: 0.6874087452888489
Epoch 6, Loss: 0.6848956942558289
Epoch 7, Loss: 0.682404488325119
Epoch 8, Loss: 0.6799349188804626
Epoch 9, Loss: 0.6774855852127075
Epoch 10, Loss: 0.6750539839267731
```

성능 차이 분석

- ReLU 활성화 함수는 첫 번째 에포크에서 다소 높은 손실을 보였으나 이후 큰 개선 없이 유지되었습니다.
- Tanh는 손실 값이 꾸준히 감소하는 모습을 보이며, 마지막 에포크에서 가장 낮은 손실을 기록했습니다.
- Sigmoid 활성화 함수도 손실이 꾸준히 감소했지만, Tanh만큼 좋은 성능을 보이진 않았습니다.

결론:

- 이 실험에서는 Tanh 활성화 함수가 가장 빠른 손실 감소를 보이며 성능이 가장 좋은 것으로 확인.
- Sigmoid와 ReLU도 안정적인 학습을 보였지만, Tanh가 이 데이터셋과 모델 구조에 가장 적합한 것으로 보임.

05. [실습 4] 손실 함수 변경 실습: CrossEntropyLoss 대신 MSELoss

목차로 이동하기

In [24]:
```python
import torch
import torch.nn as nn
import torch.optim as optim

# 기본 RNN 모델 정의
class SimpleTextRNN(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_size, output_size):
        super(SimpleTextRNN, self).__init__()
        # 단어 임베딩 레이어
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        # RNN 레이어
        self.rnn = nn.RNN(embedding_dim, hidden_size, batch_first=True)
        # 출력 레이어
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = self.embedding(x)  # 입력을 임베딩으로 변환
        rnn_out, hidden = self.rnn(x)  # RNN 통과
        out = self.fc(rnn_out[:, -1, :])  # 마지막 타임스텝의 출력만 사용
        return out

# 파라미터 설정
vocab_size = 10000
embedding_dim = 10
hidden_size = 16
output_size = 1  # MSELoss를 사용할 때 출력 크기를 1로 변경 (회귀 문제처럼 다룸)

# 모델 초기화
model = SimpleTextRNN(vocab_size, embedding_dim, hidden_size, output_size)

# 최적화 알고리즘과 MSELoss 설정
optimizer = optim.Adam(model.parameters(), lr=0.001)
criterion = nn.MSELoss()  # MSELoss 사용

# 학습 루프
for epoch in range(10):  # 10 에포크 동안 학습
    model.train()
    total_loss = 0
    for text_tensor, labels in dataloader:
        labels = labels.float()  # 타겟 값을 float형으로 변환
        optimizer.zero_grad()
        output = model(text_tensor)  # 모델에 입력
        loss = criterion(output, labels)  # 손실 계산
        loss.backward()  # 역전파
        optimizer.step()  # 가중치 업데이트
        total_loss += loss.item()

    print(f'Epoch {epoch+1}, Loss: {total_loss/len(dataloader)}')
```

```
Epoch 1, Loss: 0.815737396478653
Epoch 2, Loss: 0.7514109015464783
Epoch 3, Loss: 0.6974431574344635
Epoch 4, Loss: 0.6481395065784454
Epoch 5, Loss: 0.60284623503685
Epoch 6, Loss: 0.5613270699977875
Epoch 7, Loss: 0.5234307050704956
Epoch 8, Loss: 0.48902156949043274
Epoch 9, Loss: 0.45796026289463043
Epoch 10, Loss: 0.43009747564792633
```

결과 해석

1. MSELoss 사용 시 학습 결과: 손실 값이 첫 에포크에서 0.8157로 시작하여, 마지막 10번째 에포크에서 0.4301로 줄어듭니다. 손실 값의 감소율이 전반적으로 빠르고 꾸준하게 감소하고 있습니다. 최종 손실 값이 10번째 에포크에서 0.4301으로, 상당히 낮은 수준으로 감소했습니다.

2. CrossEntropyLoss 사용 시 학습 결과: 손실 값이 첫 에포크에서 0.6748로 시작하여, 10번째 에포크에서 0.5829로 줄어듭니다. 손실 값의 감소율이 상대적으로 완만합니다. 특히, 손실 값의 차이가 크게 줄지 않고 서서히 감소하는 양상을 보입니다. 최종 손실 값은 10번째 에포크에서 0.5829로, 여전히 어느 정도 손실이 남아있습니다.

손실 값의 감소율:

MSELoss는 손실 값이 빠르게 감소하며, 10번째 에포크에서 0.4301까지 낮아집니다. 반면, CrossEntropyLoss는 손실 값이 더 서서히 감소하며, 최종적으로 0.5829에 도달합니다. MSELoss는 더 빠른 학습 속도를 보이는 반면, CrossEntropyLoss는 더 안정적으로 손실 값을 줄이는 경향이 있습니다.

성능이 좋은 것처럼 보이는 이유:

MSE 손실 값이 더 빠르게 감소하고 작은 값으로 수렴하는 것처럼 보일 수 있습니다. 하지만 이진 분류 문제에서 중요한 것은 정확도나 F1 스코어 같은 분류 성능 지표이지, 손실 값의 절대적인 크기만은 아닙니다. CrossEntropyLoss는 모델이 0과 1 사이에서 더 나은 확률 분포를 예측할 수 있도록 도와줍니다. 반면, MSE는 그 차이를 절대적인 값으로 계산할 뿐, 확률적 해석에는 적합하지 않습니다.

- 분류 문제에서는 CrossEntropyLoss를 사용하는 것이 좋습니다. MSELoss는 회귀 문제나 실수형 출력을 예측할 때 사용해야 한다.