Homework #1

Report

- MLP 하이퍼파라미터 튜닝 실험 -

전공 컴퓨터공학전공

이름 한사랑

학번 2271064

1. 서론

다층 퍼셉트론(MLP)은 다양한 기계 학습 문제에 활용되는 핵심 모델이지만, 최적의 성능을 내기 위해서는 적절한 하이퍼파라미터 설정이 필수적입니다. 본 보고서는 MNIST 손글씨 데이터셋을 이용하여, MLP 모델의 주요 하이퍼파라미터 변화가 분류 성능에 미치는 영향을 분석하고자 합니다.

이를 위해 Baseline 모델을 설정하고 다음 요소들을 순차적으로 변경하며 실험을 진행했습니다:

- 네트워크 구조 (깊이 및 너비)
- 활성화 함수 (Sigmoid, Tanh, ReLU)
- 옵티마이저 (SGD, Adam)
- 학습률 (0.1, 1.0, 0.01)

실험에는 28x28 픽셀 크기의 MNIST 그레이스케일 이미지(학습 6만, 테스트 1만)를 사용했습니다. 각 실험 모델의 성능은 1. 테스트 정확도 2. 테스트 손실 3. 총 훈련 시간을 기준으로 평가하고 비교했습니다.

모든 실험은 PyTorch 라이브러리를 활용하였고, Visual Studio Code - 로컬 개발 환경(MacBook Air M2)에서 수행되었습니다.

전체 코드가 담긴 깃허브 저장소 링크:
https://github.com/Sarang-Han/25-1-AI/blob/main/HW1_2271064.ipynb

2. 학습 환경 세팅

2.1. 주요 라이브러리 임포트

실험 수행을 위해 PyTorch 프레임워크를 사용했으며, 데이터 처리 및 결과 시각화를 위해 NumPy, Matplotlib, Seaborn 라이브러리를 함께 활용했습니다. 필요한 라이브러리는 다음과 같이 임포트했습니다.

```
# 데이터 처리 및 시각화를 위한 라이브러리
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# PyTorch 관련 라이브러리
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
import torchvision
from torchvision import datasets, transforms
from torchvision.transforms import ToTensor

# 실행 장치 확인 (CPU/GPU)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f'Using device: {device}')
```

또한 모델 학습 및 평가는 GPU를 우선 사용하도록 설정했습니다. 다만 실제 실험이 진행된 로컬 환경에서는 GPU가 아닌 CPU가 연산 장치로 사용되었습니다.

2.2. 데이터 로드 및 전처리

MNIST 데이터 로드 및 전처리는 과제 가이드라인에서 제공된 코드를 활용하였습니다. 모든 이미지 데이터를 PyTorch가 처리할 수 있는 Tensor 형태로 변환하고, MNIST 데이터셋의 알려진 평균(0.1307)과 표준편차(0.3081)를 사용하여 정규화하는 전처리 과정을 적용했습니다.

```
# 이미지 변환 및 정규화 설정
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])

# MNIST 데이터셋 로드 (학습용/테스트용)
trainval_set = datasets.MNIST(root='./data', train=True, download=True,
transform=transform)
testset = datasets.MNIST(root='./data', train=False, download=True, transform=transform)
```

2.3. 데이터셋 분할 및 DataLoader 설정

원본 학습 데이터셋(trainval_set)은 모델 학습 중 성능 검증을 위해 8:2 비율로 분할하였습니다. 과제 가이드라인에서는 학습 및 테스트 데이터셋만 명시했지만, 본 실험에서는 더 엄밀한 성능 평가를 위해 별도의 검증 데이터셋을 구성하여 사용했습니다.

그 결과, 48,000개의 데이터를 학습용으로, 12,000개의 데이터를 학습 중 성능 검증용으로 사용했습니다. 모델의 최종 성능 평가는 별도로 로드한 10,000개의 테스트 데이터셋으로 수행했습니다.

이렇게 분할된 각 데이터셋의 배치 단위 처리를 위해 `DataLoader`를 생성했으며, 학습 데이터는 매 에폭마다 무작위로 섞어(shuffle=True) 사용했습니다. 배치 크기는 학습/검증 64, 테스트 1000으로 설정했습니다.

```
# 학습/검증 데이터 분할 (8:2)
train_size = int(0.8 * len(trainval_set))
val_size = len(trainval_set) - train_size
train_dataset, val_dataset = random_split(trainval_set, [train_size, val_size])

# DataLoader 생성
batch_size_train_val = 64
batch_size_test = 1000
train_loader = DataLoader(train_dataset, batch_size=batch_size_train_val, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size_train_val, shuffle=False)
test_loader = DataLoader(testset, batch_size=batch_size_test, shuffle=False)

# 분할된 데이터셋 크기 확인
print(f"학습 데이터셋 크기: {len(train_dataset)}")    # 출력: 학습 데이터셋 크기: 48000
print(f"검증 데이터셋 크기: {len(val_dataset)}")      # 출력: 검증 데이터셋 크기: 12000
print(f"테스트 데이터셋 크기: {len(testset)}")        # 출력: 테스트 데이터셋 크기: 10000
```

3. 실험 결과 및 분석

3.1. 베이스라인 모델

이 단계에서는 이후 진행될 실험 결과와 비교하기 위한 기준(Baseline) 모델을 구현하고 성능을 측정했습니다.

3.1.1. 모델 아키텍처

베이스라인 모델은 과제 가이드라인에 따라 다음과 같은 구조로 설계되었습니다:

- 입력층: 784 유닛 (28×28 픽셀 이미지를 1차원으로 펼친 형태)
- 은닉층 1: 32 유닛, 활성화 함수: Sigmoid
- 은닉층 2: 32 유닛, 활성화 함수: Sigmoid
- 출력층: 10 유닛 (0~9 숫자 클래스 분류)

```
class BaselineMLP(nn.Module):
    def __init__(self):
        super(BaselineMLP, self).__init__()

        # 네트워크 구조 정의 (입력 784 -> 은닉층 32 -> 은닉층 32 -> 출력 10)
        self.fc1 = nn.Linear(28*28, 32)  # 입력층 -> 첫 번째 은닉층
        self.fc2 = nn.Linear(32, 32)     # 첫 번째 은닉층 -> 두 번째 은닉층
        self.fc3 = nn.Linear(32, 10)     # 두 번째 은닉층 -> 출력층

        # 시그모이드 활성화 함수
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        # 입력 이미지 평탄화 (배치크기, 1, 28, 28) -> (배치크기, 784)
        x = x.view(-1, 28*28)

       # 순전파
        x = self.sigmoid(self.fc1(x))  # 첫 번째 은닉층 + 활성화 함수
        x = self.sigmoid(self.fc2(x))  # 두 번째 은닉층 + 활성화 함수
        x = self.fc3(x)                # 출력층

        return x

# 모델 인스턴스 생성
model = BaselineMLP().to(device)
print(model)
```

3.1.2. 학습 설정

모델 학습에는 다중 클래스 분류 문제에 표준적으로 사용되는 교차 엔트로피 손실 함수(Cross Entropy Loss)를 사용했습니다. 모델 파라미터 최적화에는 확률적 경사 하강법(SGD) 옵티마이저를 적용했으며, 학습률은 0.1로 설정했습니다. 총 10 epoch 동안 학습을 진행했습니다.

```
# 손실 함수 정의
criterion = nn.CrossEntropyLoss()

# 옵티마이저 정의
learning_rate = 0.1
optimizer = optim.SGD(model.parameters(), lr=learning_rate)
```

3.1.3. 학습 및 평가 실행

앞서 정의한 train_model 함수를 사용하여 48,000개의 학습 데이터와 12,000개의 검증 데이터로 모델을 훈련했으며, 매 에폭마다 검증 성능을 측정했습니다. 10 epoch 학습 완료 후 test_model 함수를 이용해 10,000개의 테스트 데이터셋에서 최종 성능을 평가했습니다. 전체 학습 및 평가 과정과 결과는 baseline_results 딕셔너리에 저장했습니다.

1. **학습 및 검증 함수 정의**

```
import time
from datetime import timedelta

# 학습 및 검증 함수
def train_model(model, train_loader, val_loader, criterion, optimizer, device,
num_epochs=10):
    # 결과 저장용 리스트
    train_losses, train_accs = [], []
    val_losses, val_accs = [], []

    # 전체 학습 시간 측정 시작
    total_start_time = time.time()

    print("학습 시작...")
    print("-" * 50)

    for epoch in range(num_epochs):
        # 학습 모드
        model.train()
        running_loss = 0.0
        correct = 0
        total = 0

        # 학습 단계
        for inputs, labels in train_loader:
```

```
            inputs, labels = inputs.to(device), labels.to(device)

            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            running_loss += loss.item() * inputs.size(0)
            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        # 학습 평균 손실 및 정확도 계산
        epoch_train_loss = running_loss / len(train_loader.dataset)
        epoch_train_acc = 100 * (correct / total)
        train_losses.append(epoch_train_loss)
        train_accs.append(epoch_train_acc)

        # 검증 모드
        model.eval()
        running_loss = 0.0
        correct = 0
        total = 0

        # 검증 단계
        with torch.no_grad():
            for inputs, labels in val_loader:
                inputs, labels = inputs.to(device), labels.to(device)

                outputs = model(inputs)
                loss = criterion(outputs, labels)

                running_loss += loss.item() * inputs.size(0)
                _, predicted = torch.max(outputs, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()

        # 검증 평균 손실 및 정확도 계산
        epoch_val_loss = running_loss / len(val_loader.dataset)
        epoch_val_acc = 100 * (correct / total)
        val_losses.append(epoch_val_loss)
        val_accs.append(epoch_val_acc)

        # 결과 출력
        print(f"Epoch [{epoch+1}/{num_epochs}]")
        print(f"Train - Loss: {epoch_train_loss:.4f}, Accuracy: {epoch_train_acc:.2f}%")
        print(f"Valid - Loss: {epoch_val_loss:.4f}, Accuracy: {epoch_val_acc:.2f}%")
        print("-" * 50)

    # 총 학습 시간 계산 및 출력
    total_time = time.time() - total_start_time
    print(f"총 학습 시간: {str(timedelta(seconds=total_time))}")

    return model, train_losses, train_accs, val_losses, val_accs, total_time
```

2. 평가 함수 정의

```
def test_model(model, test_loader, criterion, device):
    model.eval()
    running_loss = 0.0
    correct = 0
    total = 0

    with torch.no_grad():
        for inputs, labels in test_loader:
            inputs, labels = inputs.to(device), labels.to(device)

            outputs = model(inputs)
            loss = criterion(outputs, labels)

            running_loss += loss.item() * inputs.size(0)
            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    test_loss = running_loss / len(test_loader.dataset)
    test_acc = 100 * (correct / total)

    print("-" * 50)
    print(f"테스트 손실: {test_loss:.4f}")
    print(f"테스트 정확도: {test_acc:.2f}%")

    return test_loss, test_acc
```

3. 모델 학습 및 평가 실행

```
num_epochs = 10
model, train_losses, train_accs, val_losses, val_accs, total_time = train_model(
    model, train_loader, val_loader, criterion, optimizer, device, num_epochs)

# 테스트 세트에 대한 최종 평가
test_loss, test_acc = test_model(model, test_loader, criterion, device)

# 실험 결과 저장
baseline_results = {
    'model': 'Baseline MLP',
    'hidden_layers': '2 layers, 32 units each',
    'activation': 'Sigmoid',
    'optimizer': 'SGD',
    'learning_rate': learning_rate,
    'train_losses': train_losses,
    'train_accs': train_accs,
    'val_losses': val_losses,
    'val_accs': val_accs,
    'test_loss': test_loss,
    'test_acc': test_acc,
    'training_time': total_time
}
print("-" * 50)
print("베이스라인 모델 학습 완료!")
```

3.1.4. 베이스라인 모델 학습 결과

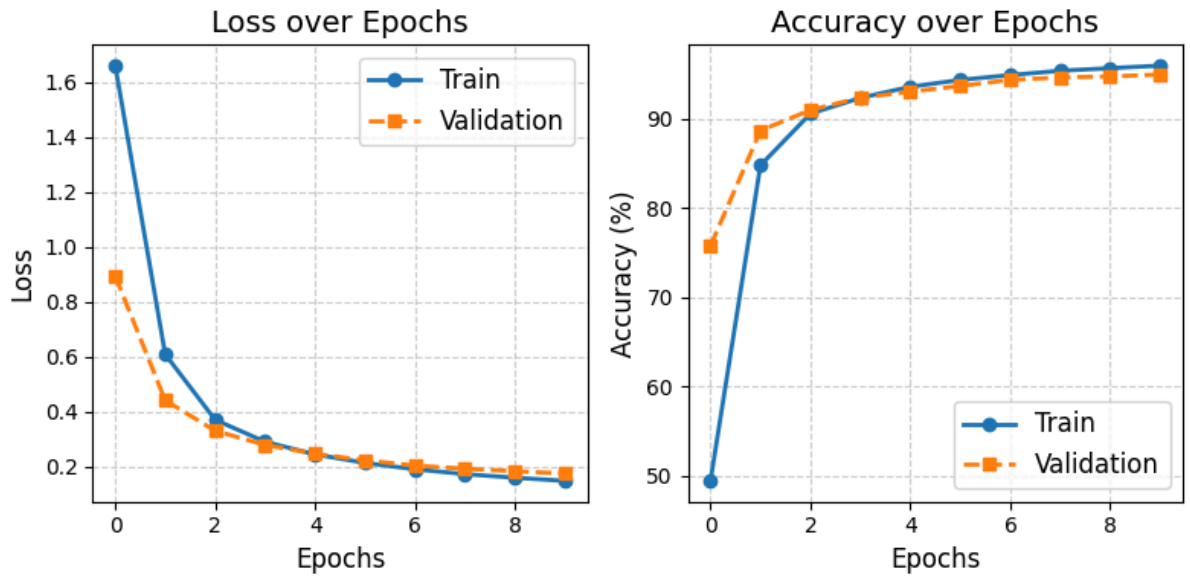
베이스라인 모델의 주요 성능 지표는 다음과 같습니다:

- 최종 테스트 정확도: **95.39%**
- 최종 테스트 손실: **0.1662**
- 총 훈련 시간: **약 28.06초**

아래 표는 베이스라인 모델의 핵심 성능 지표를 요약한 것입니다.

	Model	Hidden Layers	Activation	Optimizer	Learning Rate	Test Accuracy (%)	Training Time (s)
0	Baseline MLP	2 layers, 32 units each	Sigmoid	SGD	0.1	95.39	28.06

학습 과정 중 epoch별 훈련/검증 손실 및 정확도 변화는 visualize_training_results 함수를 통해 시각화했습니다. 아래 그래프에서 볼 수 있듯이, 학습이 진행됨에 따라 Loss는 꾸준히 감소하고 Accuracy는 상승하여 10 epoch 내에 안정적으로 수렴하는 양상을 보였습니다. 검증 손실과 정확도 역시 훈련 데이터와 유사한 추세를 보이며 과적합이 크게 발생하지 않았습니다.



성능 시각화는 아래 코드를 사용하였습니다:

```
def visualize_training_results(results):
    plt.figure(figsize=(8, 4))

    # Loss 그래프
    plt.subplot(1, 2, 1)
    plt.plot(results['train_losses'], label='Train', marker='o', markersize=6,
linestyle='-', linewidth=2)
    plt.plot(results['val_losses'], label='Validation', marker='s', markersize=6,
linestyle='--', linewidth=2)
    plt.title('Loss over Epochs', fontsize=14)
    plt.xlabel('Epochs', fontsize=12)
    plt.ylabel('Loss', fontsize=12)
    plt.grid(True, linestyle='--', alpha=0.6)
    plt.legend(fontsize=12)

    # Accuracy 그래프
    plt.subplot(1, 2, 2)
    plt.plot(results['train_accs'], label='Train', marker='o', markersize=6,
linestyle='-', linewidth=2)
    plt.plot(results['val_accs'], label='Validation', marker='s', markersize=6,
linestyle='--', linewidth=2)
    plt.title('Accuracy over Epochs', fontsize=14)
    plt.xlabel('Epochs', fontsize=12)
    plt.ylabel('Accuracy (%)', fontsize=12)
    plt.grid(True, linestyle='--', alpha=0.6)
    plt.legend(fontsize=12)

    plt.tight_layout()
    plt.show()

    # 테스트 결과 출력
    print(f"Test Accuracy: {results['test_acc']:.2f}%")
    print(f"Test Loss: {results['test_loss']:.4f}")
    print(f"Training Time: {timedelta(seconds=results['training_time'])}")

# 베이스라인 모델 학습 결과 시각화
visualize_training_results(baseline_results)

# 성능 요약 표시
performance_summary = {
    'Model': [baseline_results['model']],
    'Hidden Layers': [baseline_results['hidden_layers']],
    'Activation': [baseline_results['activation']],
    'Optimizer': [baseline_results['optimizer']],
    'Learning Rate': [baseline_results['learning_rate']],
    'Test Accuracy (%)': [f"{baseline_results['test_acc']:.2f}"],
    'Training Time (s)': [f"{baseline_results['training_time']:.2f}"]
}

# 결과 표 생성
import pandas as pd
summary_df = pd.DataFrame(performance_summary)
display(summary_df)
```

3.2. 실험 1: 더 깊은 네트워크 모델 (Deeper Model)

첫번째 실험에서는 네트워크 깊이(depth)가 모델 성능에 미치는 영향을 확인하기 위해, 베이스라인 모델의 다른 하이퍼파라미터는 동일하게 유지하면서 은닉층의 개수만 늘려 성능 변화를 관찰했습니다.

3.2.1. 모델 아키텍처 수정

베이스라인 모델(은닉층 2개)에서 은닉층 개수를 **6개**로 늘렸습니다. 각 은닉층의 유닛 수는 32개, 활성화 함수는 Sigmoid로 베이스라인과 동일하게 유지했습니다.

- **입력층:** 784 유닛
- **은닉층 1~6:** 각 32 유닛, 활성화 함수: Sigmoid
- **출력층:** 10 유닛

```
class DeeperMLP(nn.Module):
    def __init__(self):
        super(DeeperMLP, self).__init__()

        # 네트워크 구조 정의 (입력 784 -> 6개 은닉층(각 32 유닛) -> 출력 10)
        self.fc1 = nn.Linear(28*28, 32)  # 입력층 -> 첫 번째 은닉층
        self.fc2 = nn.Linear(32, 32)     # 첫 번째 은닉층 -> 두 번째 은닉층
        self.fc3 = nn.Linear(32, 32)     # 두 번째 은닉층 -> 세 번째 은닉층
        self.fc4 = nn.Linear(32, 32)     # 세 번째 은닉층 -> 네 번째 은닉층
        self.fc5 = nn.Linear(32, 32)     # 네 번째 은닉층 -> 다섯 번째 은닉층
        self.fc6 = nn.Linear(32, 32)     # 다섯 번째 은닉층 -> 여섯 번째 은닉층
        self.fc7 = nn.Linear(32, 10)     # 여섯 번째 은닉층 -> 출력층

        # 시그모이드 활성화 함수
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        # 입력 이미지 평탄화 (배치크기, 1, 28, 28) -> (배치크기, 784)
        x = x.view(-1, 28*28)

        # 순전파
        x = self.sigmoid(self.fc1(x))  # 첫 번째 은닉층 + 활성화 함수
        x = self.sigmoid(self.fc2(x))  # 두 번째 은닉층 + 활성화 함수
        x = self.sigmoid(self.fc3(x))  # 세 번째 은닉층 + 활성화 함수
        x = self.sigmoid(self.fc4(x))  # 네 번째 은닉층 + 활성화 함수
        x = self.sigmoid(self.fc5(x))  # 다섯 번째 은닉층 + 활성화 함수
        x = self.sigmoid(self.fc6(x))  # 여섯 번째 은닉층 + 활성화 함수
        x = self.fc7(x)                # 출력층

        return x

# 모델 인스턴스 생성 및 출력
deeper_model = DeeperMLP().to(device)
print(deeper_model)
```

3.2.2. 더 깊은 네트워크 모델 학습 결과

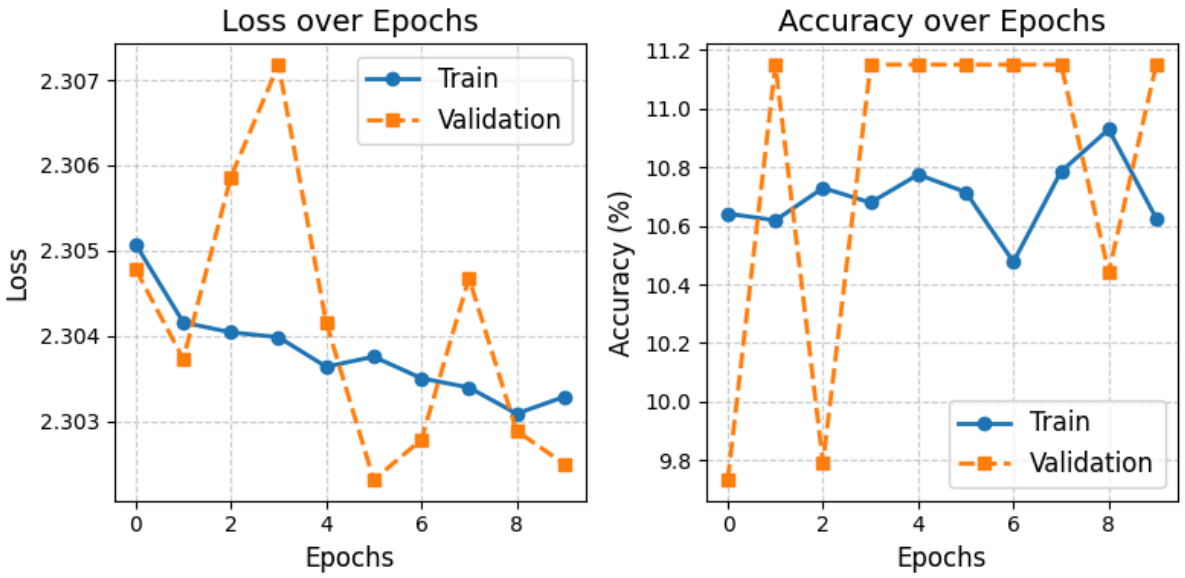
6개의 은닉층으로 깊어진 모델의 주요 성능 지표는 다음과 같습니다:

- 최종 테스트 정확도: **11.35%**
- 최종 테스트 손실: **2.3019**
- 총 훈련 시간: **약 29.46초**

아래 표는 베이스라인 모델과 깊어진 모델의 성능을 비교한 결과입니다.

	Model	Hidden Layers	Activation	Test Accuracy (%)	Test Loss	Training Time (s)
0	Baseline MLP	2 layers, 32 units each	Sigmoid	95.39	0.1662	28.06
1	Deeper MLP	6 layers, 32 units each	Sigmoid	11.35	2.3019	29.46

표에서 드러나듯이, 은닉층을 6개로 늘린 모델은 베이스라인 모델(테스트 정확도 95.39%)에 비해 성능이 급격하게 저하되었습니다. 테스트 정확도는 11.35%로, 10개 클래스를 무작위로 찍는 수준(10%)과 거의 차이가 없으며, 테스트 손실 역시 2.3 수준으로 매우 높게 나타났습니다. 훈련 시간은 베이스라인보다 약 1.4초 정도 소폭 증가했습니다.



위 학습 과정 그래프를 보면, 훈련 및 검증 정확도 모두 약 10-11% 수준에서 정체되는 모습을 확인할 수 있습니다. 손실 값 역시 epoch이 진행되어도 거의 변화 없이 높은 값(약 2.3)을 유지했습니다. 이는 Sigmoid 활성화 함수를 사용하는 깊은 네트워크에서 흔히 발생하는 기울기 소실 문제가 주된 원인으로 추정됩니다. 결과적으로, 단순히 네트워크를 깊게 쌓는 것만으로는 성능 향상을 보장할 수 없으며 오히려 학습이 실패로 이어질 수 있음을 확인했습니다.

3.3. 실험 2: 더 넓은 네트워크 모델 (Wider Model)

두 번째 실험에서는 네트워크의 너비(width), 즉 은닉층의 유닛 수가 모델 성능에 미치는 영향을 확인했습니다. 베이스라인 모델의 은닉층 수(2개)는 유지하되, 각 은닉층의 유닛 수를 늘려 성능을 비교 분석했습니다.

3.3.1. 모델 아키텍처 수정

모델 아키텍처는 베이스라인의 각 은닉층 유닛 수를 32개에서 512개로 크게 늘렸습니다. 은닉층의 개수(2개)와 활성화 함수(Sigmoid)는 베이스라인 모델과 동일하게 유지했습니다. 수정된 아키텍처는 다음과 같습니다:

- **입력층:** 784 유닛
- **은닉층 1: 512 유닛**, 활성화 함수: Sigmoid
- **은닉층 2: 512 유닛**, 활성화 함수: Sigmoid
- **출력층:** 10 유닛

```
class WiderMLP(nn.Module):
    def __init__(self):
        super(WiderMLP, self).__init__()

        # 네트워크 구조 정의 (입력 784 -> 은닉층 512 -> 은닉층 512 -> 출력 10)
        self.fc1 = nn.Linear(28*28, 512)  # 입력층 -> 첫 번째 은닉층 (512 유닛)
        self.fc2 = nn.Linear(512, 512)    # 첫 번째 은닉층 -> 두 번째 은닉층 (512 유닛)
        self.fc3 = nn.Linear(512, 10)     # 두 번째 은닉층 -> 출력층

        # 시그모이드 활성화 함수 (베이스라인과 동일)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        # 입력 이미지 평탄화 (배치크기, 1, 28, 28) -> (배치크기, 784)
        x = x.view(-1, 28*28)

        # 순전파
        x = self.sigmoid(self.fc1(x))  # 첫 번째 은닉층 + 활성화 함수
        x = self.sigmoid(self.fc2(x))  # 두 번째 은닉층 + 활성화 함수
        x = self.fc3(x)                # 출력층

        return x

# 모델 인스턴스 생성 및 출력
wider_model = WiderMLP().to(device)
print(wider_model)
```

3.3.2. 더 넓은 네트워크 모델 학습 결과

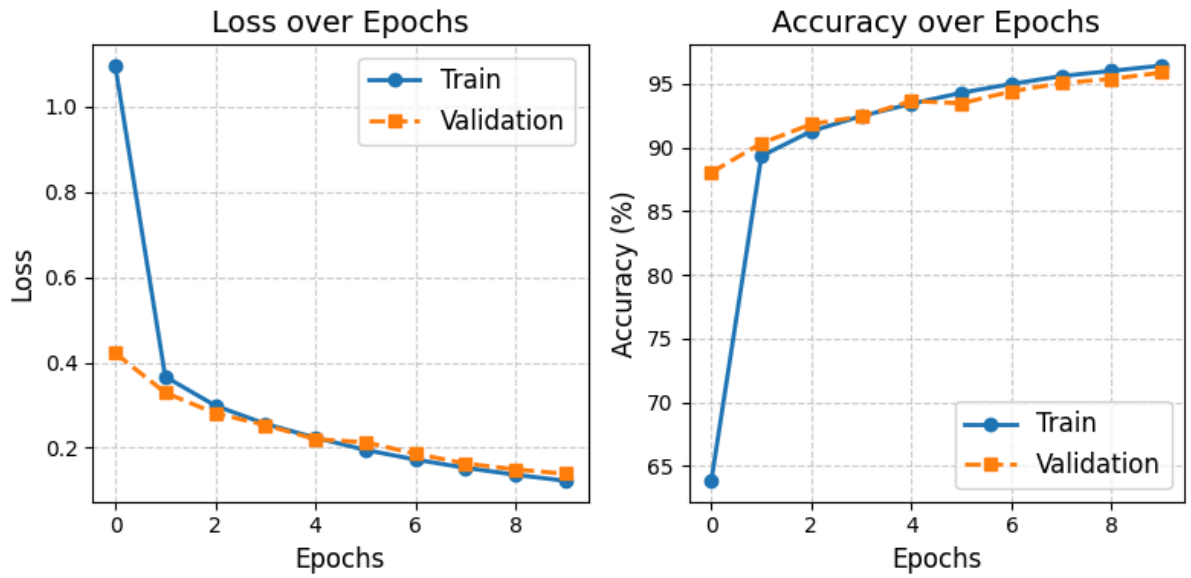
은닉 유닛 수를 512개로 늘린 넓은 모델의 주요 성능 지표는 다음과 같습니다:

- 최종 테스트 정확도: **96.15%**
- 최종 테스트 손실: **0.1289**
- 총 훈련 시간: **약 32.97초**

아래 표는 지금까지 진행한 세 가지 모델의 성능을 비교한 결과입니다.

	Model	Hidden Layers	Activation	Test Accuracy (%)	Test Loss	Training Time (s)
0	Baseline MLP	2 layers, 32 units each	Sigmoid	95.39	0.1662	28.06
1	Deeper MLP	6 layers, 32 units each	Sigmoid	11.35	2.3019	29.46
2	Wider MLP	2 layers, 512 units each	Sigmoid	96.15	0.1289	32.97

은닉층의 유닛 수를 512개로 늘린 넓은 모델은 베이스라인 모델(95.39%)보다 테스트 정확도가 0.76%p 소폭 상승하여 96.15%를 기록했으며, 테스트 손실 또한 0.1662에서 0.1289로 개선되었습니다. 이는 현재까지 실험한 모델 중 가장 좋은 성능입니다. 반면, 파라미터 수가 크게 늘어남에 따라 훈련 시간은 약 32.97초로 베이스라인(28.06초)보다 약 4.9초 증가했습니다.



위 그래프를 보면, 넓은 모델 역시 안정적으로 학습이 진행되어 수렴하는 것을 확인할 수 있습니다. 특히, 첫 번째 에폭(Epoch 1)에서의 학습 및 검증 성능을 베이스라인과 비교해보면, 넓은 모델이 더 낮은 손실과 높은 정확도로 시작하는 것을 관찰할 수 있습니다.
결론적으로, 네트워크 너비를 늘리는 것은 성능 향상에 소폭 긍정적인 영향을 주었고 초기 학습 속도가 개선되는 효과가 있었습니다. 다만 파라미터 수가 크가 늘어남에 따라 훈련 시간도 함께 증가했습니다.

3.4. 실험 3: 다양한 활성화 함수 (Activation Function Variation)

세 번째 실험에서는 은닉층에서 사용하는 활성화 함수가 MLP 모델의 성능에 미치는 영향을 비교 분석했습니다.

3.4.1. 모델 아키텍처 수정

```
# 1. Tanh 활성화 함수를 사용하는 모델 정의
class TanhMLP(nn.Module):
    def __init__(self):
        super(TanhMLP, self).__init__()

        # 네트워크 구조 정의 (입력 784 -> 은닉층 32 -> 은닉층 32 -> 출력 10)
        self.fc1 = nn.Linear(28*28, 32)  # 입력층 -> 첫 번째 은닉층
        self.fc2 = nn.Linear(32, 32)     # 첫 번째 은닉층 -> 두 번째 은닉층
        self.fc3 = nn.Linear(32, 10)     # 두 번째 은닉층 -> 출력층

        # Tanh 활성화 함수
        self.tanh = nn.Tanh()

    def forward(self, x):
        # 입력 이미지 평탄화 (배치크기, 1, 28, 28) -> (배치크기, 784)
        x = x.view(-1, 28*28)

        # 순전파
        x = self.tanh(self.fc1(x))  # 첫 번째 은닉층 + Tanh 활성화 함수
        x = self.tanh(self.fc2(x))  # 두 번째 은닉층 + Tanh 활성화 함수
        x = self.fc3(x)             # 출력층
        return x

# 2. ReLU 활성화 함수를 사용하는 모델 정의
class ReLUMLP(nn.Module):
    def __init__(self):
        super(ReLUMLP, self).__init__()

        # 네트워크 구조 정의 (입력 784 -> 은닉층 32 -> 은닉층 32 -> 출력 10)
        self.fc1 = nn.Linear(28*28, 32)  # 입력층 -> 첫 번째 은닉층
        self.fc2 = nn.Linear(32, 32)     # 첫 번째 은닉층 -> 두 번째 은닉층
        self.fc3 = nn.Linear(32, 10)     # 두 번째 은닉층 -> 출력층

        # ReLU 활성화 함수
        self.relu = nn.ReLU()

    def forward(self, x):
        # 입력 이미지 평탄화 (배치크기, 1, 28, 28) -> (배치크기, 784)
        x = x.view(-1, 28*28)

        # 순전파
        x = self.relu(self.fc1(x))  # 첫 번째 은닉층 + ReLU 활성화 함수
        x = self.relu(self.fc2(x))  # 두 번째 은닉층 + ReLU 활성화 함수
        x = self.fc3(x)             # 출력층
        return x
```

3.4.2. 다양한 활성화 함수 모델 학습 결과

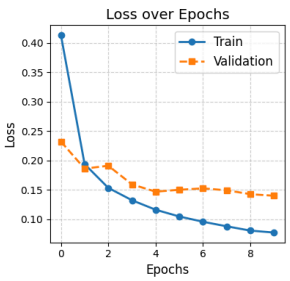
Tanh 및 ReLU 활성화 함수를 사용한 모델의 주요 성능 지표는 다음과 같습니다.

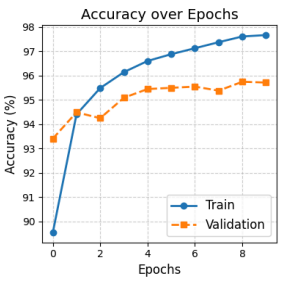
- **Tanh 모델:** 테스트 정확도 **95.87%**, 테스트 손실 **0.1380**, 훈련 시간 약 **28.29초**
- **ReLU 모델:** 테스트 정확도 **96.93%**, 테스트 손실 **0.1068**, 훈련 시간 약 **28.96초**

아래 표는 세 가지 활성화 함수(Sigmoid, Tanh, ReLU)를 적용한 모델들의 성능을 비교한 결과입니다.

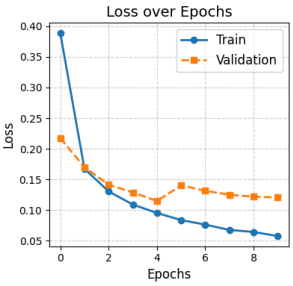
	Model	Activation	Test Accuracy (%)	Test Loss	Training Time (s)
0	Baseline MLP	Sigmoid	95.39	0.1662	28.06
1	Tanh MLP	Tanh	95.87	0.1380	28.29
2	ReLU MLP	ReLU	96.93	0.1068	28.96

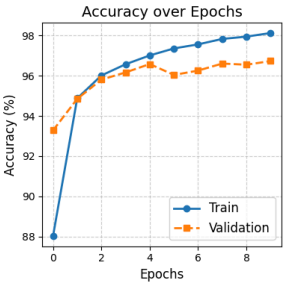
표에서 볼 수 있듯이, 활성화 함수를 Tanh 또는 ReLU로 변경했을 때 베이스라인(Sigmoid) 모델보다 더 높은 테스트 정확도와 낮은 테스트 손실을 얻었습니다. 특히 **ReLU** 활성화 함수를 사용했을 때 96.93%로 가장 좋은 성능을 보였습니다. 훈련 시간은 세 모델 간에 큰 차이가 없었습니다.





Tanh 적용 모델





ReLU 적용 모델

위 그래프들을 살펴보면, Tanh와 ReLU 모델 모두 성공적으로 학습이 진행되었음을 알 수 있습니다. 두 모델 모두 베이스라인보다 빠르게 손실을 줄이고 높은 훈련 정확도에 도달했습니다. 하지만 훈련이 진행됨에 따라 Train 정확도는 계속 상승하는 반면 Validation 정확도의 상승 폭은 둔화되어, 두 정확도 간의 간격이 베이스라인 모델보다 다소 크게 나타나는 경향을 보였습니다. 즉, 약간의 과적합 현상이 발생하고 있음을 확인했습니다.

3.5. 실험 4: 다양한 옵티마이저 (Optimizer Variation)

네 번째 실험에서는 옵티마이저 변경이 모델 성능에 미치는 영향을 확인했습니다. 이를 위해 이전 실험에서 가장 좋은 성능을 보였던 **ReLU 활성화 함수 모델**을 기반 아키텍처로 사용하되, 옵티마이저만 SGD에서 **Adam**으로 변경하여 성능을 비교 분석했습니다.

3.5.1. 모델 학습 설정 변경

- 아키텍처: 실험 3.4의 ReLU MLP와 동일
- 옵티마이저 변경: 기존 **optim.SGD** 에서 **optim.Adam** 으로 변경
- 학습률: 베이스라인 및 이전 실험들과 동일하게 **0.1**로 유지
- 기타 설정: 손실 함수, epoch 수 등은 동일하게 유지

```
# 손실 함수 정의
criterion = nn.CrossEntropyLoss()

# Adam 옵티마이저 정의
adam_learning_rate = 0.1
adam_optimizer = optim.Adam(adam_relu_model.parameters(), lr=adam_learning_rate)

# 모델 학습 실행
num_epochs = 10
adam_model, adam_train_losses, adam_train_accs, adam_val_losses, adam_val_accs,
adam_total_time = train_model(
   adam_relu_model, train_loader, val_loader, criterion, adam_optimizer, device,
num_epochs)

# 테스트 세트에 대한 최종 평가
adam_test_loss, adam_test_acc = test_model(adam_relu_model, test_loader, criterion,
device)

# 실험 결과 저장
adam_results = {
   'model': 'Adam ReLU MLP',
   'hidden_layers': '2 layers, 32 units each',
   'activation': 'ReLU',
   'optimizer': 'Adam',
   'learning_rate': adam_learning_rate,
   'train_losses': adam_train_losses,
   'train_accs': adam_train_accs,
   'val_losses': adam_val_losses,
   'val_accs': adam_val_accs,
   'test_loss': adam_test_loss,
   'test_acc': adam_test_acc,
   'training_time': adam_total_time
}

print("-" * 50)
print("Adam 옵티마이저 + ReLU 모델 학습 완료!")
```

3.5.2. 다양한 옵티마이저 모델 학습 결과

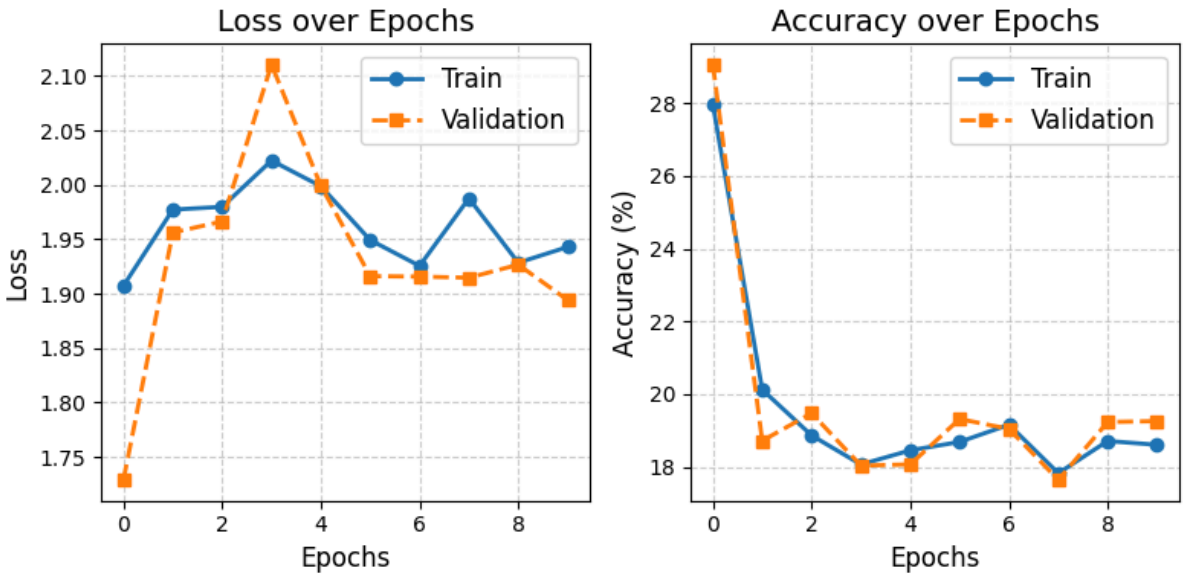
Adam 옵티마이저(학습률 0.1)를 사용한 모델의 주요 성능 지표는 다음과 같습니다.

- **최종 테스트 정확도: 19.35%**
- **최종 테스트 손실: 1.8900**
- **총 훈련 시간: 약 29.52초**

아래 표는 동일한 ReLU 활성화 함수 모델에 대해 SGD 옵티마이저(lr=0.1)를 사용했을 때와 Adam 옵티마이저(lr=0.1)를 사용했을 때의 성능을 비교한 결과입니다.

	Model	Optimizer	Learning Rate	Activation	Test Accuracy (%)	Test Loss	Training Time (s)
0	ReLU MLP	SGD	0.1000	ReLU	96.93	0.1068	28.96
1	Adam ReLU MLP	Adam	0.1000	ReLU	19.35	1.8900	29.52

일반적으로 좋은 성능을 보이는 것으로 알려진 Adam 옵티마이저를 사용했음에도 불구하고, 테스트 정확도가 19.35%로 매우 낮게 측정되었으며, 테스트 손실 역시 1.89로 매우 높게 나타나 모델이 전혀 학습되지 않았음을 알 수 있었습니다.



위 그래프를 보면, 학습이 진행될수록 오히려 성능이 저하되거나 불안정해지는 현상이 뚜렷하게 나타났습니다. 첫 epoch에서 약 28%의 정확도를 기록한 이후, 정확도가 20% 미만으로 떨어져 정체되거나 오르내렸습니다. 손실 값 역시 감소하지 않고 1.9 ~ 2.0 부근의 높은 값에서 불안정하게 변동하거나 오히려 증가하는 양상을 보였습니다. 주된 원인은 Adam 옵티마이저에 적절하지 않은 학습률(0.1)을 사용했기 때문으로 추정됩니다. 결과적으로, 특정 옵티마이저가 모든 조건에서 항상 우수한 성능을 보장하는 것은 아니며 각 옵티마이저의 특성에 맞는 적절한 학습률 설정이 중요함을 확인했습니다. 동일한 학습률 0.1 조건에서는 SGD가 Adam보다 훨씬 안정적인 학습 결과를 제공했습니다.

3.6. 실험 5: 다양한 학습률 (Learning rate Variation)

마지막 실험에서는 학습률이 모델 학습 과정과 최종 성능에 미치는 영향을 확인했습니다.

3.6.1. 모델 학습 설정 변경

각 학습률(1.0, 0.1, 0.01)에 대해 독립적으로 모델을 초기화하고 10 epoch 동안 학습 및 평가를 진행하여, 결과를 lr_results 딕셔너리에 저장했습니다.

- 아키텍처: 실험 3.4의 ReLU MLP와 동일
- 옵티마이저: SGD (고정)
- 학습률 변경: **1.0, 0.1, 0.01** 세 가지 경우로 실험
- 기타 설정: 손실 함수, epoch 수 등은 동일하게 유지

```
# 다양한 학습률로 모델 학습 및 평가
learning_rates = [1.0, 0.1, 0.01]
lr_results = {}

# 각 학습률에 대해 실험 수행
for lr in learning_rates:
    print(f"\n===== 학습률 {lr} 실험 =====")

    # 모델 초기화
    lr_model = LearningRateMLP().to(device)

    criterion = nn.CrossEntropyLoss()
    lr_optimizer = optim.SGD(lr_model.parameters(), lr=lr)

    num_epochs = 10
    lr_model, lr_train_losses, lr_train_accs, lr_val_losses, lr_val_accs, lr_total_time =
train_model(
        lr_model, train_loader, val_loader, criterion, lr_optimizer, device, num_epochs)

    lr_test_loss, lr_test_acc = test_model(lr_model, test_loader, criterion, device)

    lr_results[lr] = {
        'model': f'ReLU MLP (LR={lr})',
        'hidden_layers': '2 layers, 32 units each',
        'activation': 'ReLU',
        'optimizer': 'SGD',
        'learning_rate': lr,
        'train_losses': lr_train_losses,
        'train_accs': lr_train_accs,
        'val_losses': lr_val_losses,
        'val_accs': lr_val_accs,
        'test_loss': lr_test_loss,
        'test_acc': lr_test_acc,
        'training_time': lr_total_time
    }

    print(f"학습률 {lr} 실험 완료!")
```

3.6.2. 다양한 학습률 모델 학습 결과

각 학습률 설정에 따른 주요 성능 지표는 다음과 같습니다.

- **학습률 1.0:** 테스트 정확도 11.37%, 테스트 손실 2.3048, 훈련 시간 약 28.17초
- **학습률 0.1:** 테스트 정확도 **96.45%**, 테스트 손실 **0.1264**, 훈련 시간 약 28.68초
- **학습률 0.01:** 테스트 정확도 95.30%, 테스트 손실 0.1671, 훈련 시간 약 29.41초

아래 표는 세 가지 학습률에 대한 성능 비교 결과입니다.

	Model	Learning Rate	Activation	Test Accuracy (%)	Test Loss	Training Time (s)
0	ReLU MLP (LR=1.0)	1.00	ReLU	11.37	2.3048	28.17
1	ReLU MLP (LR=0.1)	0.10	ReLU	96.45	0.1264	28.68
2	ReLU MLP (LR=0.01)	0.01	ReLU	95.30	0.1671	29.41

결과에서 볼 수 있듯이, 학습률 설정은 모델 성능에 매우 큰 영향을 미쳤습니다.

- **학습률 1.0:** 이전 Adam(lr=0.1) 실험과 마찬가지로 학습이 전혀 이루어지지 않고 발산했습니다.
- **학습률 0.1:** 96.45%로 세 가지 학습률 중 가장 높은 테스트 정확도를 달성했습니다.
- **학습률 0.01:** 95.30%의 테스트 정확도를 기록하여 0.1보다는 약간 낮지만 우수한 성능을 보였습니다.

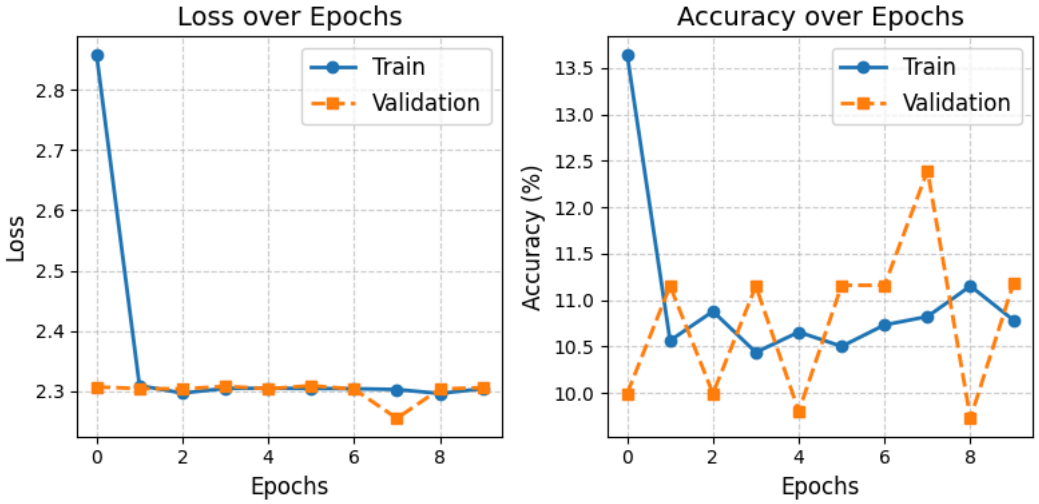
(다음 페이지) 다음으로 학습 과정 그래프를 비교해보았을때, 흥미로운 점을 발견할 수 있었습니다. 학습률 0.1은 가장 높은 최종 테스트 정확도를 기록했지만, 학습 후반부로 갈수록 훈련 정확도(~98.2%)와 검증 정확도(~96.4%) 사이의 간격이 상대적으로 벌어져 과적합의 경향을 보였습니다.

반면, 학습률 0.01은 최종 정확도는 0.1보다 약간 낮았지만, 학습 그래프에서 훈련 정확도(~95.4%)와 검증 정확도(~94.7%)가 매우 비슷한 수준으로 함께 증가하며 그 간격이 매우 작게 유지되었습니다. 이는 가장 안정적인 학습 과정을 보였습니다.

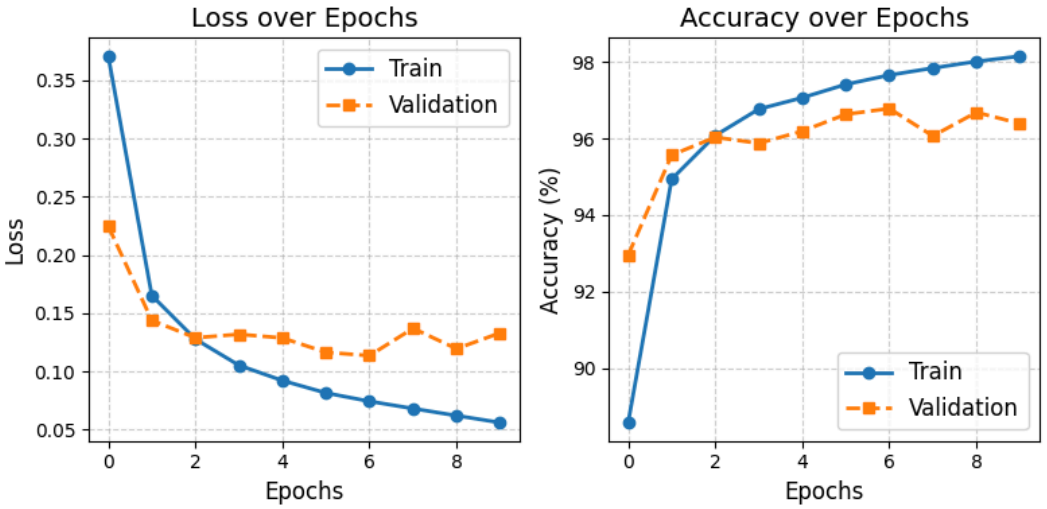
결과적으로 이 실험을 통해 학습률 설정의 중요성과 trade-off를 확인할 수 있었습니다. 너무 높은 학습률은 학습 실패를 초래하고, 적절히 높은 학습률은 빠른 학습과 좋은 성능을 가져올 수 있지만 과적합의 위험이 따릅니다. 반면 낮은 학습률은 학습 속도가 느릴 수 있지만 안정적인 수렴과 낮은 과적합 위험이라는 장점을 가집니다.

따라서 최종 성능 지표뿐만 아니라 학습 과정의 안정성과 과적합 정도를 함께 고려하여 최적의 학습률을 선택하는 것이 중요함을 알 수 있었습니다.

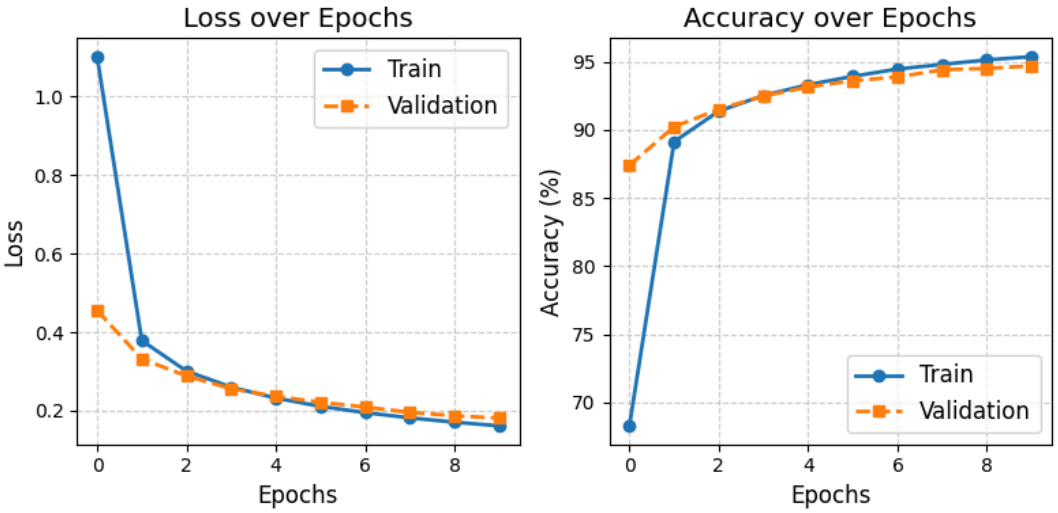
학습률 1.0 적용 그래프



학습률 0.1 적용 그래프



학습률 0.01 적용 그래프



4. 결론

본 보고서에서는 MNIST 데이터셋 기반 MLP 모델의 주요 하이퍼파라미터(네트워크 구조, 활성화 함수, 옵티마이저, 학습률)가 성능에 미치는 영향을 실험을 통해 분석했습니다.

주요 실험 결과 및 핵심 요약은 다음과 같습니다.

- **네트워크 구조:** 은닉층 깊이를 늘리는 것(Sigmoid 사용 시)은 기울기 소실 문제로 성능 저하를 야기한 반면, 너비를 늘리는 것은 소폭의 성능 향상을 가져왔으나 훈련 시간이 증가했습니다.
- **활성화 함수: ReLU** > **Tanh** > **Sigmoid** 순으로 높은 테스트 정확도를 보였으며, ReLU 사용 시 가장 좋은 성능을 기록했습니다. 다만, ReLU와 Tanh는 Sigmoid 대비 약간의 과적합 경향을 나타냈습니다.
- **옵티마이저 & 학습률:** Adam 옵티마이저는 SGD에 적합했던 높은 학습률(0.1)에서 학습에 실패하여, 옵티마이저별 적절한 학습률 설정의 중요성을 보여주었습니다. SGD 옵티마이저의 학습률 조정 실험에서는 0.1이 가장 높은 테스트 정확도를 달성했지만 과적합 위험이 있었고, 0.01은 안정성은 가장 높았으나 정확도는 다소 낮았습니다.

결론적으로, MLP 모델의 성능은 특정 요소가 아닌 다양한 하이퍼파라미터 간의 상호작용에 크게 의존함을 확인했습니다. 본 실험의 제한된 조건 하에서는 ReLU 활성화 함수 - SGD 옵티마이저 - 학습률 0.1 조합이 가장 우수한 테스트 정확도(약 96-97%)를 달성했습니다. 하지만 과적합 가능성이나 학습 안정성까지 고려한다면 다른 설정(학습률 0.01)이 더 적합할 수도 있습니다.