Homework #2

Report

- CNN과 정규화 기법 실험 -

전공 컴퓨터공학전공

이름 한사랑

학번 2271064

1. 서론

본 보고서에서는 CIFAR-10 데이터셋을 활용하여 간단한 이미지 분류용 CNN 모델을 구현합니다. 특히 ResNet(Residual Network) 아키텍처를 직접 구현하고, 다양한 정규화 기법을 관찰하며 각 기법이 모델의 학습에 미치는 영향을 비교 분석하고자 합니다.

구현된 주요 요소는 다음과 같습니다:

- 데이터 증강을 위한 데이터로더
- ResNet 기본 블록 및 전체 모델
- 학습 함수: 모델 파라미터를 최적화하는 훈련 루프
- 평가 함수: 학습된 모델의 성능을 측정하는 검증 루프

실험에는 32x32 픽셀 크기의 CIFAR-10 컬러 이미지(학습 5만, 테스트 1만)이 사용되었습니다.
모든 실험은 PyTorch 라이브러리를 활용하였고, Google Colab - T4 GPU 환경에서 수행되었습니다.

전체 코드가 담긴 깃허브 저장소 링크:

https://github.com/Sarang-Han/AI-HW2

2. 실험 설계

2.1. 데이터 전처리 및 증강

```
# Data augmentation: Not applied
transform_basic = transforms.Compose([
    transforms.ToTensor()
])

# Data augmentation: Random horizontal flit, Random crop
transform_aug = transforms.Compose([
    # TODO
    transforms.RandomHorizontalFlip(),
    transforms.RandomCrop(32, padding=4),
    transforms.ToTensor()
])

# Train set (Use only 10% of the entire samples to induce overfitting)
train_full = torchvision.datasets.CIFAR10(root='./data', train=True, download=True,
transform=transform_basic)
subset_size = int(0.1 * len(train_full))  # 5000 samples
train_subset = Subset(train_full, range(subset_size))
train_loader = DataLoader(train_subset, batch_size=64, shuffle=True)

# Test set (Use full samples)
test_set = torchvision.datasets.CIFAR10(root='./data', train=False, download=True,
transform=transform_basic)
test_loader = DataLoader(test_set, batch_size=64, shuffle=False)
```

CIFAR-10 데이터셋을 로드하고, 모델 학습에 적합한 데이터로 전처리하는 과정입니다.

데이터 변환 (Transform):

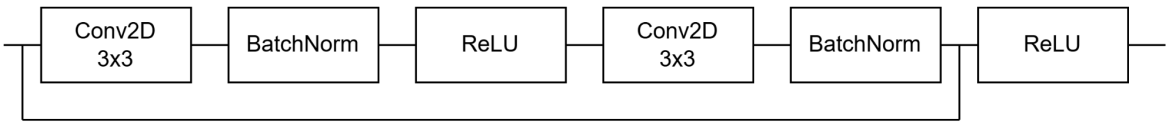
- `transform_basic`: 데이터 증강 없이, 이미지를 텐서로만 변환
- `transform_aug`: `RandomHorizontalFlip`과 `RandomCrop`으로 데이터 증강 적용 (수평 뒤집기, 랜덤 크롭)

데이터셋 분할:

- 학습 데이터셋은 **과적합(overfitting) 현상**을 유도하기 위해 **전체의 10%만 사용**함
- 테스트 데이터셋은 전체 데이터를 사용

2.2. ResNet 기본 블록 구현

```
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, downsample=False):
        super().__init__()

        # TODO
        stride = 2 if downsample else 1

        # Main path
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride,
padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1,
padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)

        # Skip connection (shortcut)
        self.shortcut = nn.Sequential()

        if downsample or in_channels != out_channels:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=stride,
bias=False),
                nn.BatchNorm2d(out_channels)
            )

    def forward(self, x):

        # TODO
        identity = x

        # Main path
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)

        # Skip connection
        out += self.shortcut(identity)
        out = self.relu(out)

        return out
```



먼저, 주어진 구조에 따라 기본적인 Residual Block 클래스를 구현하였습니다.

stride 설정:

- downsample이 True일 때 stride를 2로, 아니면 1로 설정하여 다운샘플링 여부를 결정함.

메인 경로(Main path):

- `conv1`: 입력 채널 → 출력 채널 매핑하는 3x3 컨볼루션. stride 값에 따라 다운샘플링 여부 결정.
- 배치 정규화(`bn1`)와 ReLU 활성화(`relu`)를 적용.
- `conv2`: 출력 채널에서 특징을 추가로 추출하는 3x3 컨볼루션(stride=1, 크기 유지).
- conv2 출력에 두 번째 배치 정규화(`bn2`)를 적용.

스킵 연결(Shortcut):

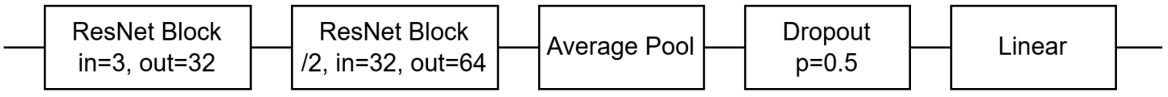
- 입력과 출력의 크기나 채널 수가 다를 때(`downsample=True` 또는 `in_channels != out_channels`) 1x1 컨볼루션과 배치 정규화로 차원을 맞춰줌.
- 그렇지 않으면 입력을 그대로 전달.

forward 함수:

- 입력을 `identity = x`로 저장.
- 메인 경로를 따라 conv1 → bn1 → relu → conv2 → bn2 순서로 연산.
- Skip connection을 통해 identity를 더함. (`out += self.shortcut(identity)`)
- 마지막으로 ReLU 활성화 함수를 적용하여 최종 출력 반환

2.3. ResNet 모델 아키텍처 구현

```
class ResNetSmall(nn.Module):
    def __init__(self, dropout=False):
        super().__init__()

        # ResNet blocks
        self.block1 = ResidualBlock(3, 32, downsample=False)
        self.block2 = ResidualBlock(32, 64, downsample=True)

        # Average pooling
        self.avg_pool = nn.AvgPool2d(kernel_size=8)

        # Dropout
        self.use_dropout = dropout
        if self.use_dropout:
            self.dropout = nn.Dropout(0.5)

        # FC Layer
        self.fc = nn.Linear(64, 10)

    def forward(self, x):
        # ResNet blocks
        x = self.block1(x)
        x = self.block2(x)
        # Average pooling
        x = self.avg_pool(x)
        # Flatten
        x = x.view(x.size(0), -1)
        # Dropout
        if self.use_dropout:
            x = self.dropout(x)
        # FC Layer
        x = self.fc(x)
        return x
```



다음으로, ResNetSmall 클래스를 구현했습니다.

ResNet 블록:

- 첫 번째 블록: 입력 채널 3, 출력 채널 32, 다운샘플링 없음
- 두 번째 블록: 입력 채널 32, 출력 채널 64, 다운샘플링 적용

특징 추출 후 처리:

- 8x8 크기의 평균 풀링(AvgPool2d) 적용하여 특징 맵 압축
- Dropout 옵션: 파라미터로 제어 가능하며, 활성화 시 0.5 확률로 뉴런 비활성화
- 최종 출력을 위한 완전 연결 계층(FC Layer): 256 입력 → 10 출력(CIFAR-10 클래스)

2.4. 학습 및 평가함수 구현

1. 학습 함수 구현

```
def train(model, train_loader, optimizer, criterion):
    model.train()
    correct = total = 0
    loss_total = 0

    for inputs, labels in train_loader:
        # TODO
        # Move data to device
        inputs, labels = inputs.to(device), labels.to(device)
        # Zero the parameter gradients
        optimizer.zero_grad()
        # Forward pass
        outputs = model(inputs)
        # Calculate loss
        loss = criterion(outputs, labels)
        # Backward pass and optimize
        loss.backward()
        optimizer.step()

        loss_total += loss.item()
        _, preds = torch.max(outputs, 1)
        correct += (preds == labels).sum().item()
        total += labels.size(0)

    accuracy = correct / total
    avg_loss = loss_total / len(train_loader)
    return avg_loss, accuracy
```

데이터 처리

- inputs와 labels을 지정된 계산 장치(CPU 또는 GPU)로 이동

그래디언트 초기화

- optimizer.zero_grad()로 파라미터 그래디언트를 0으로 초기화

순방향 전파

- 모델에 입력 데이터를 전달하여 예측값 생성
- outputs = model(inputs)를 통해 모델의 출력을 계산

손실 계산

- criterion(손실 함수)를 사용하여 모델의 예측과 실제 레이블 간의 오차 계산

역전파 및 최적화

- loss.backward()로 손실에 대한 그래디언트를 계산
- optimizer.step()으로 계산된 그래디언트를 사용하여 모델 파라미터 업데이트

2. 평가 함수 구현

```
def evaluate(model, test_loader, criterion):
    model.eval()
    correct = total = 0
    loss_total = 0

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

            # Forward pass
            outputs = model(inputs)

            # Calculate loss
            loss = criterion(outputs, labels)

            loss_total += loss.item()
            _, preds = torch.max(outputs, 1)
            correct += (preds == labels).sum().item()
            total += labels.size(0)

    accuracy = correct / total
    avg_loss = loss_total / len(test_loader)
    return avg_loss, accuracy
```

테스트 데이터 처리

- 입력 데이터와 레이블을 지정된 계산 장치로 이동 `# Move data to device`
- 모델에 입력 데이터를 전달하여 예측값 생성 `# Forward pass`

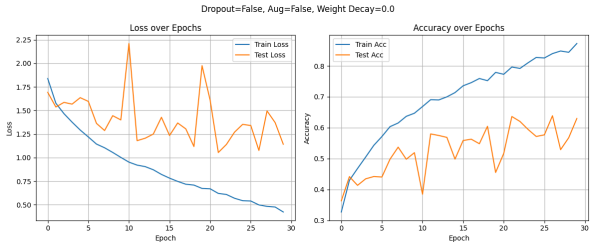
손실 계산

- 모델의 출력과 실제 레이블 간의 손실 계산 `# Calculate loss`
- 배치별 손실값을 누적하여 전체 데이터셋에 대한 평균 손실 계산

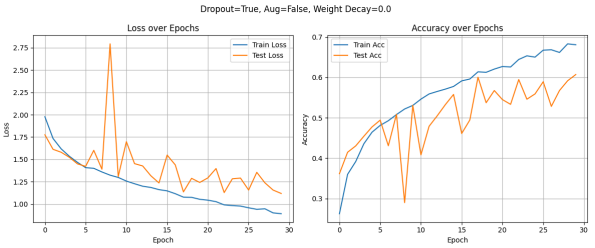
3. 실험 결과

3.1. 각 모델의 학습 결과 그래프

실험 A : Baseline

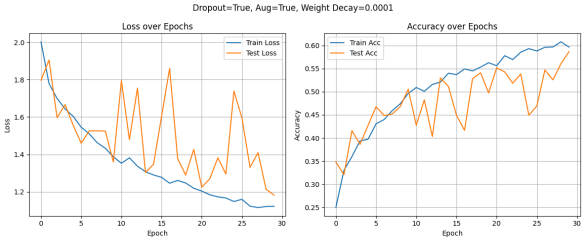


실험 B : Dropout only



실험 C : Data augmentation only

실험 D : Dropout + Augment + WD



3.2. 실험 결과 비교

실험	설명	train Acc	Test Acc	정확도 차이	최고 테스트 정확도	과적합 정도
A	베이스라인	87.24%	62.97%	24.27%	63.86%	매우 강함
B	드롭아웃만 적용	68.08%	60.71%	7.37%	60.71%	중간
C	데이터 증강만 적용	70.12%	52.67%	17.45%	62.89%	강함
D	드롭아웃 + 데이터 증강 + 가중치 감소	59.66%	58.65%	1.01%	58.65%	약함

실험 A :

- 훈련 정확도는 계속 증가하여 87.24%에 도달
- 테스트 정확도는 크게 변동하며 최대 63.86%에 그침
- 훈련과 테스트 정확도 간 차이가 24.27%로 강한 과적합 발생

실험 B :

- 훈련 정확도 상승이 억제되어 68.08%에 머무름
- 테스트 정확도는 60.71%로 실험 A보다 조금 낮음
- 훈련-테스트 정확도 차이가 7.37%로 과적합이 상당히 감소

실험 C :

- 훈련 정확도는 70.12%로 실험 A보다 낮음
- 테스트 정확도는 불안정하며 최종적으로 52.67%에 그침
- 17.45%의 정확도 차이로 여전히 과적합 문제 존재

실험 D :

- 훈련 정확도는 가장 낮은 59.66%에 머무름
- 테스트 정확도 58.65%로 훈련 정확도와 거의 비슷함
- 훈련-테스트 정확도 차이가 1.01%로 과적합이 거의 없음

3.3. 각 정규화 방법의 효과 비교

드롭아웃 (실험 B)

- 훈련 정확도 상승을 억제하여 과적합 감소에 효과적
- 테스트 정확도의 변동성을 줄여 안정적인 성능
- 기본 모델 대비 과적합이 24.27% → 7.37%로 감소

데이터 증강 (실험 C)

- 훈련 정확도는 억제되었으나 테스트 정확도의 변동성이 높음
- 최고 테스트 정확도는 좋으나(62.89%) 일관성이 부족
- 과적합 감소에는 효과적이나 드롭아웃보다는 덜 효과적

복합 정규화 (실험 D)

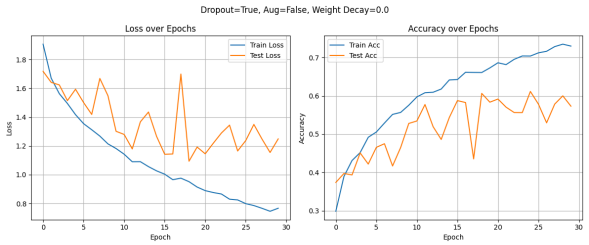
- 모든 정규화 기법이 결합되어 과적합이 거의 없음(1.01%)
- 전체적인 모델 성능(테스트 정확도)은 다소 저하됨
- 훈련과 테스트 성능이 가장 균형 잡힌 모델로 보여짐

3.4. 정규화 하이퍼파라미터 변경 및 비교

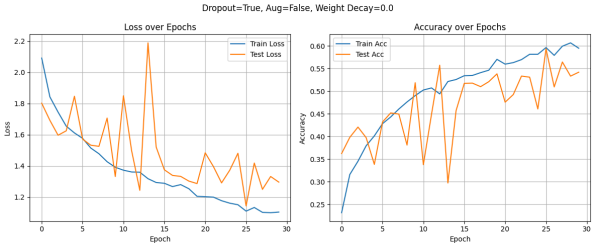
이 실험에서는 각 정규화 기법의 하이퍼파라미터 조정에 따른 효과를 비교하였다. 이를 드롭아웃 확률 변경 / 데이터 증강 강도 변경 / 가중치 감소 강도 변경 세가지 조정을 진행하였고, 각각 기본 모델은 모델 B / 모델 C / 모델 A (WD 인자만 변경)으로 고정하여 확인하였다.

1. 드롭아웃 확률 변경

Dropout(0.3)



Dropout(0.7)



지표	Dropout(0.3)	Dropout(0.5) - 실험 B	Dropout(0.7)
최종 훈련 정확도	72.98%	68.08%	59.50%
최종 테스트 정확도	57.29%	60.71%	54.20%
과적합 정도 (훈련-테스트 차이)	15.69%	7.37%	5.30%
최고 테스트 정확도	61.13% (25 에폭)	60.71% (30 에폭)	59.37% (26 에폭)
훈련 속도 (마지막 10 에폭 정확도 증가)	4.7%p	2.7%p	0.5%p
학습 안정성	중간	양호	불안정

학습/테스트 정확도

- 드롭아웃 확률이 높을수록 훈련 정확도가 현저히 감소 (73% → 68% → 60%)
- 높은 드롭아웃은 모델의 표현력을 크게 제한하여 훈련 데이터에서도 낮은 성능을 보임
- 드롭아웃 0.5가 가장 높은 최종 테스트 정확도(60.71%)를 보임
- 드롭아웃 0.3은 일부 에폭에서 더 높은 테스트 정확도(61.13%)를 기록했으나 최종 성능은 떨어짐
- 드롭아웃 0.7은 과도한 규제로 인해 테스트 성능이 가장 낮음

과적합 방지:

- 드롭아웃 확률이 높을수록 과적합이 더 효과적으로 방지됨 (15.7% → 7.4% → 5.3%)
- 그러나 0.7의 경우 모델의 학습 자체가 제한되어 전반적인 성능이 저하됨

학습 안정성:

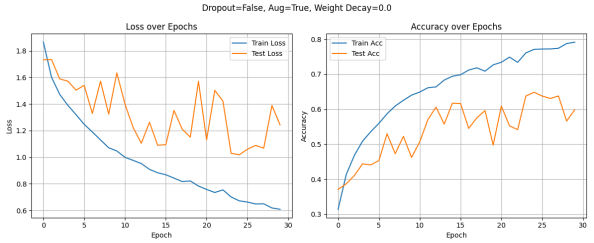
- 드롭아웃 0.5가 가장 안정적인 학습 패턴을 보여줌
- 드롭아웃 0.7은 테스트 정확도의 변동성이 크고 학습이 불안정함
- 드롭아웃 0.3은 과적합 위험은 있으나 상대적으로 안정적인 학습 보임

2. 데이터 증강 강도 변경

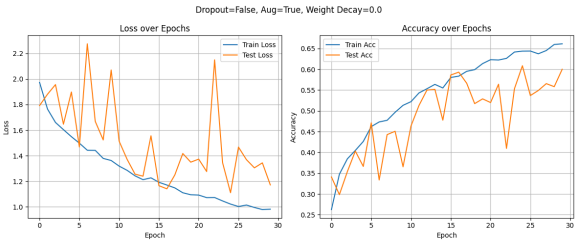
데이터 증강 실험을 위해 transform 함수를 조정하였다. 아래와 같이, 세가지 케이스로 나누어 비교하였다.

- 약한 증강 (RandomHorizontalFlip만 사용)
- 현재 증강 (RandomHorizontalFlip, RandomCrop 사용)
- 강한 증강 ((RandomHorizontalFlip, RandomCrop, ColorJitter 사용)

약한 증강



강한 증강



증강 방법	사용된 기법	최종 Train Acc	최종 Test Acc	최고 Test Acc	Train-Test 격차
약한 증강	RandomHorizontalFlip	79.16%	59.87%	64.84%	19.29%
기본 증강 (C)	RandomHorizontalFlip + RandomCrop	70.12%	52.67%	62.89%	17.45%
강한 증강	RandomHorizontalFlip + RandomCrop + ColorJitter	66.12%	59.99%	60.84%	6.13%

학습 정확도

- 약한 증강이 가장 빠르게 학습되며, 초기 epoch부터 높은 정확도를 보임
- 강한 증강은 초기 epoch에서 학습 속도가 느리지만, 후반에 안정적으로 수렴
- 기본 증강은 중간 정도의 학습 속도를 보이나 테스트 정확도의 변동이 심함

과적합 방지

- 강한 증강이 Train-Test 격차 6.13%로 가장 효과적으로 과적합 방지
- 기본 증강은 17.45%의 격차로 중간 정도 효과
- 약한 증강은 19.29%의 격차로 상대적으로 과적합 경향이 높음

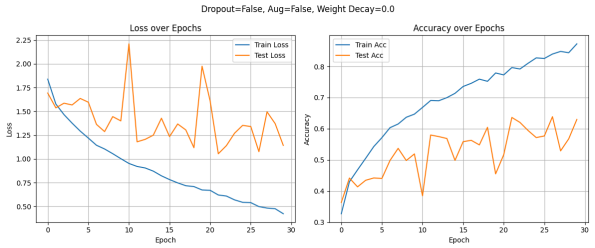
최종 성능

- 약한 증강이 최고 테스트 정확도 64.84%로 가장 우수
- 기본 증강은 최고 62.89%로 중간
- 강한 증강은 안정적이지만 최고 성능은 60.84%로 다소 낮음

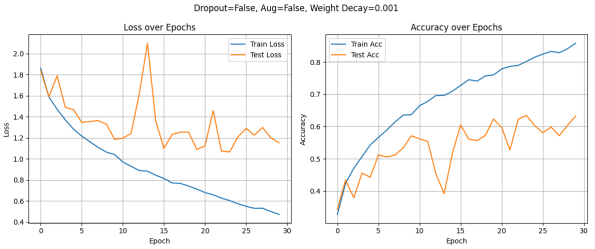
3. 가중치 감소 강도 변경

이번 실험에서는 기본 모델 A에서 `weight_decay`의 값을 각각 0.0 (A모델), `1e-3`, `1e-4`, `1e-5`로 변경하며 결과를 확인하였다.

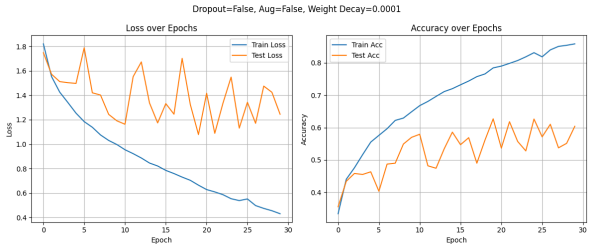
`weight_decay = 0.0`



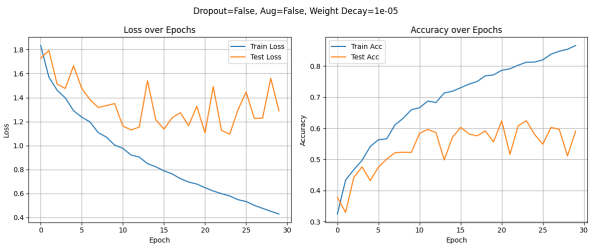
`weight_decay = 1e-3`



`weight_decay = 1e-4`



`weight_decay = 1e-5`



Weight Decay	최종 Train Acc	최종 Test Acc	Train Loss	Test Loss	Train-Test Acc
0.0 (기본 A)	0.8724	0.6297	0.4229	1.1416	0.2427
1e-3	0.8580	0.6321	0.4709	1.1520	0.2259
1e-4	0.8584	0.6038	0.4304	1.2449	0.2546
1e-5	0.8660	0.5910	0.4292	1.2887	0.2750

과적합 제어:

- 1e-3 적용 시 과적합이 가장 효과적으로 제어됨 (Train-Test Acc 차이 0.2259로 가장 작음)
- 기본 모델(A)과 비교했을 때, 1e-3 적용 시 훈련 정확도는 약간 감소했지만(0.8724→0.8580) 테스트 정확도는 소폭 향상됨(0.6297→0.6321)

학습 패턴:

- 모든 모델이 비슷한 학습 패턴을 보이지만, Weight Decay 값이 클수록(1e-3) 훈련 손실이 상대적으로 높게 유지됨
- 이는 가중치가 과도하게 커지는 것을 제한하여 일반화 성능을 향상시키는 Weight Decay의 목적과 일치함

성능 균형:

- Weight Decay 1e-3은 훈련 정확도와 테스트 정확도 간의 균형이 가장 좋음
- 너무 작은 값(1e-5)은 기본 모델(A)과 거의 차이가 없고, 오히려 테스트 정확도가 낮아짐

4. 결론

본 실험에서는 CIFAR-10 이미지 분류 작업에서 과적합을 방지하고 일반화 성능을 향상시키기 위한 다양한 정규화 기법을 적용하고 그 효과를 비교 분석했습니다. 주요 정규화 기법으로 드롭아웃(Dropout), 데이터 증강(Data Augmentation), 가중치 감소(Weight Decay)를 사용했으며, 각 기법의 강도와 조합에 따른 성능 변화를 관찰했습니다.

실험의 주요 관찰은 다음과 같습니다:

정규화 기법의 효과

- 드롭아웃: 가장 효과적인 과적합 방지 기법으로, Train-Test 격차를 크게 줄임
- 데이터 증강: 약한 증강이 가장 높은 테스트 정확도를 달성, 강한 증강은 과적합을 더 잘 방지함
- 가중치 감소: 1e-3의 강한 가중치 감소가 가장 높은 테스트 정확도를 보임

정규화 강도와 성능의 관계

- 드롭아웃: 확률이 높을수록 과적합이 감소하나, 너무 높으면(p=0.7) 학습이 부족해짐
- 데이터 증강: 과도한 증강은 학습을 어렵게 만들지만 일반화 능력은 향상시킴
- 가중치 감소: 감소 강도가 너무 낮으면 효과가 미미하고, 적절한 강도(1e-3)에서 최적의 성능 달성

정규화 기법 조합의 효과

- 모든 정규화 기법을 조합한 모델이 가장 낮은 Train-Test 격차(1.01%)를 보여 과적합이 거의 없음
- 하지만 최고 성능 측면에서는 개별 기법(특히 약한 데이터 증강, 강한 가중치 감소)이 더 효과적인 경우도 있음