팀프로젝트

결과 보고서 #1

제 목: LeNet5 인공신경망 구성 및 분류기 구현

과 목 명:	**딥러닝프로그래밍및실습**
학 부:	AI융합학부
학 번:	20192917
이 름:	이 경 주
제 출 일:	2024년 4월 26일(금)
담당교수:	한 영 준

<전체 목차>

I. 이경주 (20192917)

1. LeNet-5 기본 모델 구현 (개선전 구현) - 이경주

2. 개선 방안 제시 및 개선 시도 - 이경주

3. 기본 모델과 결과 비교 - 이경주

4. 참고 자료 - 이경주

1. LeNet-5 기본 모델 구현 (개선전 구현) - 이경주

NVIDIA Geforce RTX-2080 GPU를 사용하였다. (device: gpu)
Training 데이터셋으로 60000, Test 데이터셋으로 10000 만큼 사용하였다.
아래 DataLoader 부분을 보면 배치 크기는 32로 설정한 것을 확인할 수 있다.

다음은 LeNet-5 모델 구현이다.

중간에 tanh 활성화 함수를 사용하였으며, 최종 출력에는 Log-softmax 활성화 함수를 사용하였다. 구조는 수업자료에서 제시하는 LeNet-5 구조 그대로 구현하였다.

2. LeNet-5 모델 정의

```python
from torch import nn
import torch.nn.functional as F

class LeNet_5(nn.Module):
    def __init__(self):
        super(LeNet_5,self).__init__()
        self.conv1 = nn.Conv2d(1, 6, kernel_size=5, stride=1, padding=2)
        self.conv2 = nn.Conv2d(6, 16, kernel_size=5, stride=1)
        self.conv3 = nn.Conv2d(16, 120, kernel_size=5, stride=1)
        self.fc1 = nn.Linear(120, 84)
        self.fc2 = nn.Linear(84, 10)

    def forward(self, x):
        x = F.tanh(self.conv1(x))
        x = F.avg_pool2d(x, 2, 2)
        x = F.tanh(self.conv2(x))
        x = F.avg_pool2d(x, 2, 2)
        x = F.tanh(self.conv3(x))
        x = x.view(-1, 120)
        x = F.tanh(self.fc1(x))
        x = self.fc2(x)
        return F.log_softmax(x, dim=1)

model = LeNet_5()
print(model)
```

✓ 0.0s

```
LeNet_5(
  (conv1): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (conv3): Conv2d(16, 120, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=120, out_features=84, bias=True)
  (fc2): Linear(in_features=84, out_features=10, bias=True)
)
```

학습은 아래와 같은 설정으로 진행했다.

에폭 : 30
Loss 함수 : CrossEntropyLoss
옵티마이저 : Adam
학습률(LR) = 0.001 (고정)

아래는 epoch 수만큼 순회하며 학습하는 함수이다.

```python
# train_val 함수 정의
def train_val(model, params):
    num_epochs = params['num_epochs']
    loss_func = params['loss_func']
    opt = params['optimizer']
    train_dl = params['train_dl']
    sanity_check = params['sanity_check']

    loss_history = {
        'train': [],
        'val': [],
    }

    metric_history = {
        'train': [],
        'val': [],
    }

    for epoch in range(num_epochs):
        current_lr = get_lr(opt)
        print('Epoch {}/{}, current lr={}'.format(epoch, num_epochs-1, current_lr))
        epochStartTime = time.time()
        model.train()
        # epoch당 loss와 performance metric을 계산
        train_loss, train_metric = loss_epoch(model, loss_func, train_dl, sanity_check, opt)

        # training 도중 기록한 loss와 accuracy를 기록 (향후 시각화)
        loss_history['train'].append(train_loss)
        metric_history['train'].append(train_metric)

        model.eval()
        epoch_calculation_time = time.time() - epochStartTime

        print('train loss: %.6f, calculation time: %0.3f sec'%(train_loss, epoch_calculation_time))
        print('-'*10)
    return model, loss_history, metric_history
```

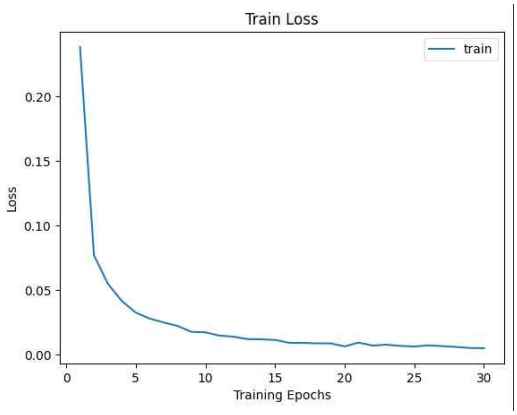
loss_epoch에 대한 구현은 아래와 같다. epoch당 loss와 학습 중의 accuracy를 계산한다.

```python
# epoch당 loss와 performance metric을 계산하는 함수 정의
def loss_epoch(model, loss_func, dataset_dl, sanity_check=False, opt=None):
    running_loss = 0.0
    running_metric = 0.0
    len_data = len(dataset_dl.dataset)

    for xb, yb in dataset_dl:
        xb = xb.type(torch.float).to(device)
        yb = yb.to(device)
        output = model(xb)
        loss_b, metric_b = loss_batch(loss_func, output, yb, opt) # 배치당 loss를 계산
        running_loss += loss_b

        if metric_b is not None:
            running_metric += metric_b

        if sanity_check is True: # sanity_check가 True이면 1epoch만 학습합니다.
            break

    loss = running_loss / float(len_data)
    metric = running_metric / float(len_data)
    return loss, metric
```

학습 중에 1 epoch 단위 training set에 대한 Loss와 수행시간을 출력했다.

```
    # 모델을 학습합니다.
    model,loss_hist,metric_hist=train_val(model,params_train)
 ✓  8m 11.8s

Epoch 20/29, current lr=0.001
train loss: 0.009101, calculation time: 16.424 sec
----------
Epoch 21/29, current lr=0.001
train loss: 0.006818, calculation time: 16.407 sec
----------
Epoch 22/29, current lr=0.001
train loss: 0.007525, calculation time: 16.299 sec
----------
Epoch 23/29, current lr=0.001
train loss: 0.006584, calculation time: 16.483 sec
----------
Epoch 24/29, current lr=0.001
train loss: 0.006065, calculation time: 16.901 sec
----------
Epoch 25/29, current lr=0.001
train loss: 0.007012, calculation time: 16.944 sec
----------
Epoch 26/29, current lr=0.001
train loss: 0.006389, calculation time: 16.961 sec
----------
Epoch 27/29, current lr=0.001
train loss: 0.005735, calculation time: 16.977 sec
----------
Epoch 28/29, current lr=0.001
train loss: 0.004949, calculation time: 16.932 sec
----------
Epoch 29/29, current lr=0.001
train loss: 0.004800, calculation time: 16.150 sec
----------
```

훈련 중의 Loss는 아래 그래프처럼 줄어드는 것을 확인할 수 있었다.



학습을 완료한 후, 기본 LeNet-5의 Test 데이터셋에 대한 정확도를 측정해보았다.
다음은 정확도를 계산하여 출력하고, 옳게 분류된 샘플들과 잘못 분류된 샘플들을 반환하는 함수이다.

```python
def check_accuracy(loader, model):

    correct_list=[]
    wrong_list=[]

    total = 0
    correct = 0  # 정답 개수를 기록하기 위한 변수

    model.eval()
    with torch.no_grad():
        for inputs, labels in loader:
            inputs, labels = inputs.to(device), labels.to(device)

            infer = model(inputs)
            predicted = infer.argmax(dim=1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

            # 올바르게 분류된 샘플과 잘못 분류된 샘플을 구분하여 저장
            for i in range(len(predicted)):
                if predicted[i] == labels[i]:
                    correct_list.append((inputs[i], labels[i], predicted[i]))
                else:
                    wrong_list.append((inputs[i], labels[i], predicted[i]))

    # 정확도 출력
    accuracy = 100 * correct / total
    print(f'Accuracy on the test set: {accuracy:.3f}%')

    # 옳게 분류된 샘플들과 잘못된 분류된 샘플들을 반환
    return correct_list, wrong_list
```

1. 아무런 변형이 없는 Test dataset에 대해 정확도를 측정하였다.
정확도는 **98.75%**가 출력되었다.

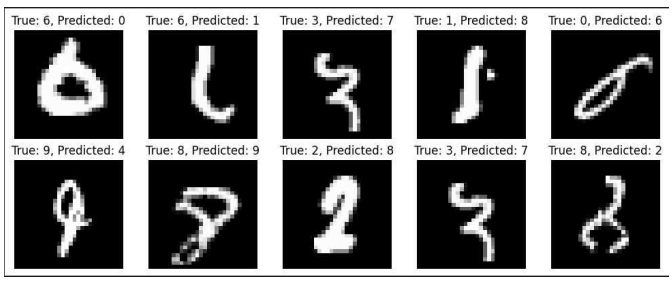
정확도 출력

```
correct_samples, wrong_samples = check_accuracy(test_dl, model)
```

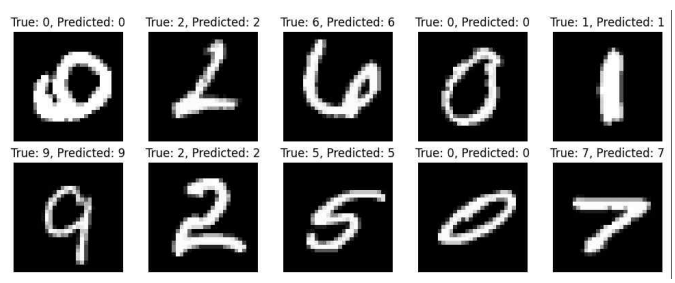
✓ 3.4s

```
Accuracy on the test set: 98.750%
```

아래는 잘못 분류한 샘플 이미지이다. (10개 랜덤 선택)



아래는 옳게 분류한 샘플 이미지이다. (10개 랜덤 선택)



2. 더 나아가, 가우시안 노이즈(**표준편차 0.4**)를 적용한 Test dataset에 대해 정확도를 측정하였다.

정확도는 **91.62%**가 출력되었다. 이는 순수 테스트 데이터셋 분류 문제에 비해 노이즈의 샘플의 분류에 대해서는 성능이 꽤 떨어진 것을 확인할 수 있는 부분이다. 아래는 Test dataset에 가우시안 노이즈를 표준편차 0.4로 가하는 코드이다. **랜덤한 노이즈 값을 추가하는 부분에 랜덤 시드를 0으로 지정하여 동일한 랜덤값을 갖도록 하였다.**

가우시안 노이즈를 적용하기 위한 클래스를 정의 (랜덤 시드를 지정함)

```
# 가우시안 노이즈를 적용하기 위한 클래스를 정의 (랜덤 시드를 지정함)
class AddGaussianNoise(object):
    def __init__(self, mean=0., std=1.):
        self.std = std
        self.mean = mean

    def __call__(self, tensor):
        torch.manual_seed(0)
        return tensor + torch.randn(tensor.size()) * self.std + self.mean

    def __repr__(self):
        return self.__class__.__name__ + '(mean={0}, std={1})'.format(self.mean, self.std)
```

✓ 0.0s

가우시안 노이즈를 추가하고 노이즈를 추가한 Sample 이미지 확인

```
from torchvision import utils

noisy_transform=transforms.Compose([
    #transforms.Resize((32, 32)),
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,)),
    AddGaussianNoise(mean=0., std=0.4),
])

# MNIST test dataset 불러오기
gau_04_test_data = datasets.MNIST(path2data, train=False, download=True, transform=noisy_transform)
gau_04_test_dl = DataLoader(gau_04_test_data, batch_size=128)
# test data를 추출합니다.
x_test, y_test = gau_04_test_data.data, gau_04_test_data.targets
```

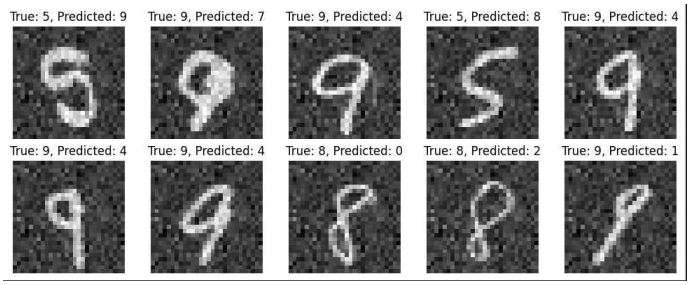
정확도 출력 (표준편차 0.4 가우시안 노이즈 적용)

```
correct_samples, wrong_samples = check_accuracy(gau_04_test_dl, model)
```

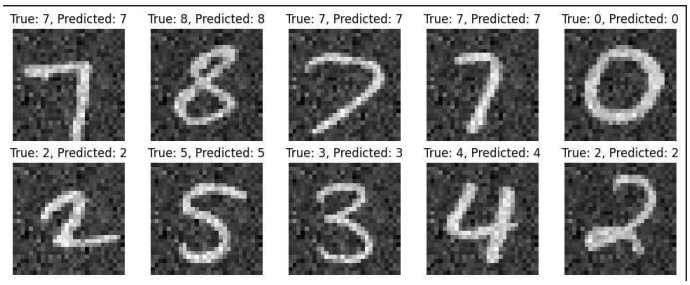
✓ 4.8s

```
Accuracy on the test set: 91.620%
```

아래는 잘못 분류한 샘플 이미지이다. (10개 랜덤 선택)



아래는 옳게 분류한 샘플 이미지이다. (10개 랜덤 선택)



3. 이번에는 가우시안 노이즈(**표준편차 0.6**)를 적용한 Test dataset에 대해 정확도를 측정하였다.
정확도는 **89.04%**가 출력되었다. 노이즈의 표준편차가 커져 표준편차 0.4의 가우시안 노이즈에 대한 테스트보다 정확도가 더 떨어졌다.

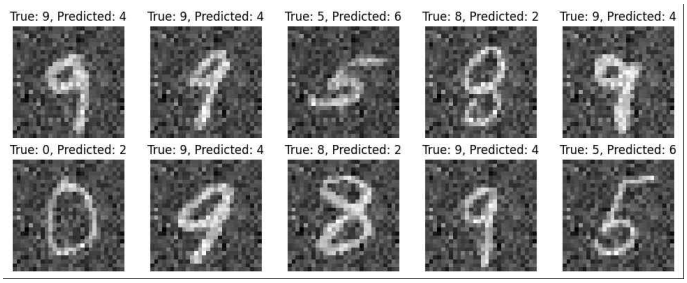
정확도 출력 (표준편차 0.6 가우시안 노이즈 적용)

```
correct_samples, wrong_samples = check_accuracy(gau_06_test_dl, model)
```

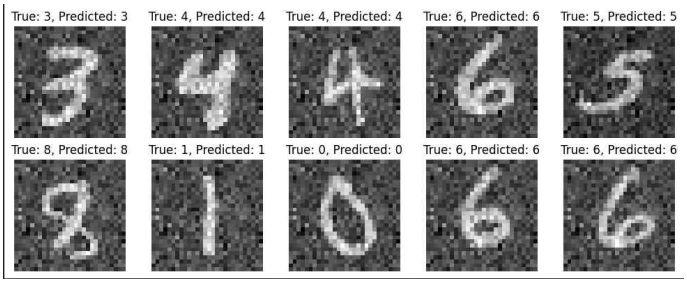
✓ 4.7s

```
Accuracy on the test set: 89.040%
```

아래는 잘못 분류한 샘플 이미지이다. (10개 랜덤 선택)



아래는 옳게 분류한 샘플 이미지이다. (10개 랜덤 선택)



4. 이번에는 가우시안 노이즈(**표준편차 0.8**)를 적용한 Test dataset에 대해 정확도를 측정하였다.
정확도는 **86.43%**가 출력되었다. 노이즈의 표준편차가 커져 표준편차 0.6의 가우시안 노이즈에 대한 테스트보다도 정확도가 더 떨어졌다.

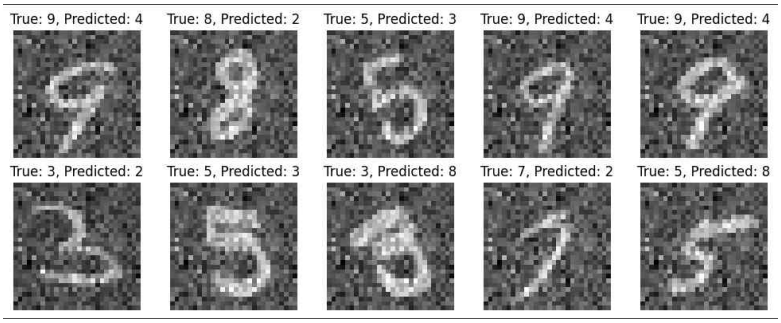
정확도 출력 (표준편차 0.8 가우시안 노이즈 적용)

```
correct_samples, wrong_samples = check_accuracy(gau_08_test_dl, model)
```

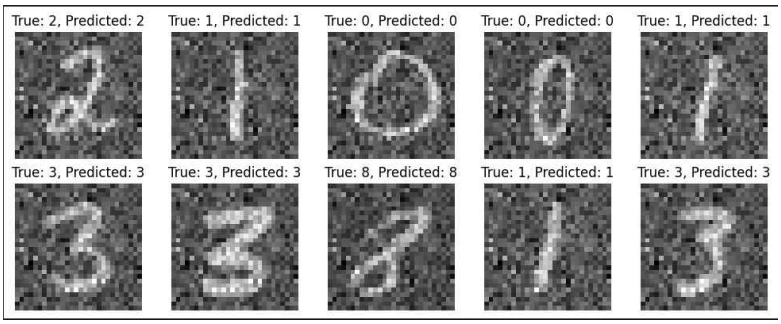
✓ 4.8s

```
Accuracy on the test set: 86.430%
```

아래는 잘못 분류한 샘플 이미지이다. (10개 랜덤 선택)



아래는 옳게 분류한 샘플 이미지이다. (10개 랜덤 선택)



2. 개선 방안 제시 및 개선 시도 - 이경주

위에서 기본적인 LeNet-5 모델을 구현하고, 변형이 없는 테스트 데이터셋과 가우시안 Noise가 있는 테스트 데이터셋에 대한 정확도를 측정하였다.

- **기본형 LeNet-5**의 정확도
  * TEST 1. Pure dataset : 98.75%
  * TEST 2. Gaussian Noise dataset(standard deviation : 0.4) : 91.62%
  * TEST 3. Gaussian Noise dataset(standard deviation : 0.6) : 89.04%
  * TEST 4. Gaussian Noise dataset(standard deviation : 0.8) : 86.43%

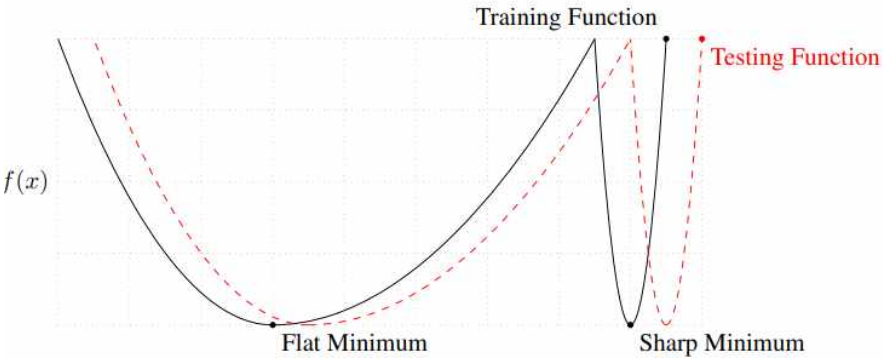
위의 실험을 통해 Noise가 있는 테스트 데이터셋에 대해서는 꽤 성능이 저하됨을 알 수 있었고, 실제 세상에서 숫자 글씨 분류기를 사용한다고 했을 때에는 Noise가 적용된 것과 같은 데이터가 입력으로 들어올 수 있다는 점을 생각해볼 때, 이 성능 저하는 문제가 될 수 있다고 생각된다.

따라서, 일반화(Generalization)도 중요한 모델 성능의 지표로 생각될 수 있다.
일반화(Generalization)은 학습된 모델이 다른 새로운 데이터에 관해서도 잘 작동하도록 하는 것을 말한다. 이와 관련해서 관심을 갖고 논문을 찾아보며 공부하였다. 아래는 정리한 내용.
https://github.com/gjlee0802/publications_summary/blob/main/Sharpness-Aware_Minimization_for_Efficiently_Improving_Generalization.md
https://github.com/gjlee0802/publications_summary/blob/main/generalization_bounds.md
찾아보던 중, Loss landscape의 sharpness(선명도)가 일반화와 연관성이 있다는 것을 새롭게 알게되었고, Google-research 소속 연구원들이 2021년 발표한 아래 논문에 대해 공부하였다.

Sharpness-aware minimization과 관련된 논문
https://arxiv.org/pdf/2010.01412.pdf
Pierre Foret, Ariel Kleiner, Hossein Mobahi, and Behnam Neyshabur. Sharpness-aware minimization for efficiently improving generalization. In International Conference on Learning Representations, 2021.

논문의 내용을 정리하자면 아래와 같다.

Loss landscape가 flat(평평)할수록 generalizability가 강해진다. 즉, Sharpness를 규제하는 것이 일반화 성능에 도움이 된다는 것이다.
Flatness of minima -> the flatter, the more generalizable!!



SAM(Sharpness-aware minimization)은 loss value와 loss sharpness를 동시적으로 최소화 하여 flatter region(더 평평한 지형)의 파라미터들을 찾을 수 있게 되며, 찾은 파라미터 근방의 파라미터들도 비슷한 낮은 loss를 갖게 된다.

그림의 왼쪽은 SGD로 학습할 때의 Loss landscape, 오른쪽은 SGD와 Sharpness-aware minimization으로 학습할 때의 Loss landscape이다.
Sharp한 방향으로 학습되지 않고 평평한쪽으로 모델이 학습되도록 Optimizer를 수정하는 것이 Sharpness-aware minimization이다.

아래는 PAC(probably approximately correct) Bayesian generalization bound 기반으로 generalization bound 산정을 하여 population loss에 대한 upper bound(상한)를 정하고, 식을 유도했을 때, 결과적으로 나오는 gradient 전개식이다.

$$\nabla_{\boldsymbol{w}} L_{\mathcal{S}}^{SAM}(\boldsymbol{w}) \approx \nabla_{\boldsymbol{w}} L_{\mathcal{S}}(w)|_{\boldsymbol{w}+\hat{\boldsymbol{\epsilon}}(\boldsymbol{w})}.$$

여기서 두 번째 항의 ϵ (perturbation, 섭동을 의미)는 따로 아래와 같이 계산한다.

$$\hat{\boldsymbol{\epsilon}}(\boldsymbol{w}) = \rho \, \text{sign}\left(\nabla_{\boldsymbol{w}} L_{\mathcal{S}}(\boldsymbol{w})\right) |\nabla_{\boldsymbol{w}} L_{\mathcal{S}}(\boldsymbol{w})|^{q-1} / \left(\|\nabla_{\boldsymbol{w}} L_{\mathcal{S}}(\boldsymbol{w})\|_q^q\right)^{1/p}$$

아래는 SAM 알고리즘이다.

Input: Training set $\mathcal{S} \triangleq \cup_{i=1}^{n}\{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}$, Loss function $l : \mathcal{W} \times \mathcal{X} \times \mathcal{Y} \to \mathbb{R}_+$, Batch size b, Step size $\eta > 0$, Neighborhood size $\rho > 0$.
Output: Model trained with SAM
Initialize weights $\boldsymbol{w}_0$, $t = 0$;
while *not converged* **do**
  Sample batch $\mathcal{B} = \{(\boldsymbol{x}_1, \boldsymbol{y}_1), ...(\boldsymbol{x}_b, \boldsymbol{y}_b)\}$;
  Compute gradient $\nabla_{\boldsymbol{w}} L_{\mathcal{B}}(\boldsymbol{w})$ of the batch's training loss;
  Compute $\hat{\boldsymbol{\epsilon}}(\boldsymbol{w})$ per equation 2;
  Compute gradient approximation for the SAM objective (equation 3): $\boldsymbol{g} = \nabla_{\boldsymbol{w}} L_{\mathcal{B}}(\boldsymbol{w})|_{\boldsymbol{w}+\hat{\boldsymbol{\epsilon}}(\boldsymbol{w})}$;
  Update weights: $\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \eta \boldsymbol{g}$;
  $t = t + 1$;
end
return $\boldsymbol{w}_t$

Algorithm 1: SAM algorithm

주목할 것은 배치의 training loss의 gradient를 먼저 계산한 후, 이를 이용해서 두 번째 항의 ϵ를 계산하고, 구한 첫 번째 항과 두 번째 항 ϵ 계산 결과를 이용하여 SAM 최종 gradient 전개식을 계산한다. 그리고 이 SAM의 최종 gradient 값으로 가중치를 업데이트한다.

SAM을 적용했을 때, SVHN과 Fasion-MNIST에서도 성능 개선을 보였으며,
ImageNet에서도 SAM을 적용했을 때 더 높은 성능을 보였고, 과적합이 덜 발생하면서 training을 진행할 수 있었다고 한다.
이 SAM 알고리즘을 LeNet-5 MNIST에 적용했을 때도 일반화 측면의 성능향상을 기대할 수 있으리라 믿고, Sharpness-aware minimization을 적용한 LeNet-5 모델 학습을 시도했다.

이번에는 Training 데이터셋에서 0.2 비율을 split하여 Validation 데이터셋으로 활용했다.
Training 데이터셋과 Validation 데이터셋 비율은 8:2이다.
Training dataset : 48000
Validation dataset : 12000
Test 데이터셋은 10000개 샘플로 이전에 개선전 LeNet-5 모델로 테스트했을 때의 테스트 데이터셋 크기와 같다.

LeNet-5 모델은 어떠한 변형 없이 구현하였다.

```
from torch import nn
import torch.nn.functional as F

class LeNet_5(nn.Module):
    def __init__(self):
        super(LeNet_5,self).__init__()
        self.conv1 = nn.Conv2d(1, 6, kernel_size=5, stride=1, padding=2)
        self.conv2 = nn.Conv2d(6, 16, kernel_size=5, stride=1)
        self.conv3 = nn.Conv2d(16, 120, kernel_size=5, stride=1)
        self.fc1 = nn.Linear(120, 84)
        self.fc2 = nn.Linear(84, 10)

    def forward(self, x):
        x = F.tanh(self.conv1(x))
        x = F.avg_pool2d(x, 2, 2)
        x = F.tanh(self.conv2(x))
        x = F.avg_pool2d(x, 2, 2)
        x = F.tanh(self.conv3(x))
        x = x.view(-1, 120)
        x = F.tanh(self.fc1(x))
        x = self.fc2(x)
        return F.log_softmax(x, dim=1)
```

SAM + SGD + momentum 조합으로 학습시켰다.
LearningRate 함수로는 LR이 계단형으로 증가하는 StepLR을 사용하였다.

배치 크기 : 128
에폭 수 : 80
초기 LR : 0.1
모멘텀 : 0.9
아래는 하이퍼파라미터 설정이다.

```
args = dict()
args['adaptive'] = True
args['batch_size'] = 128
args['epochs'] = 80
# 라벨 스무딩, 레이블을 그대로 사용하는 것이 아니라 조금 smooth하게 만들어서 정규화를 시키는 것
args['label_smoothing'] = 0.1
# 초기 학습률
args['learning_rate'] = 0.1
# Momentum은 Local Minimum에 빠지는 경우를 대처하기 위해 적용.
args['momentum'] = 0.9
# SAM의 gradient 전개식 계산에 쓰일 파라미터, 논문에 따르면 경험적으로 ρ = 2로 하여 L2-norm 계산한 것이 최적의 결과를 냄.
args['rho'] = 2.0
args['weight_decay'] = 0.0005

initialize(args, seed=42)
```

rho값은 SAM의 계산식에 쓰이는 파라미터이며, SAM 계산식 내에서는 `rho`차-norm 계산을 수행하게 되는데, SAM 논문에서는 rho값은 2로 설정하여 2차-norm을 계산할 때 경험적으로 가장 성능이 좋다고 하였다.
아래와 같이 LeNet-5 모델, SAM 옵티마이저, LR 스케줄러를 선언하였다.

```
model = LeNet_5().to(device)
print(model)

base_optimizer = torch.optim.SGD
optimizer = SAM(model.parameters(), base_optimizer, rho=args['rho'], adaptive=args['adaptive'], lr=args['learning_rate'], momentum=args['momentum'],
scheduler = StepLR(optimizer, args['learning_rate'], args['epochs'])
```

SAM Optimizer는 첫 번째 Step과 두 번째 Step이 있다.
- First Step : 로컬 `rho`-neighbor에서 손실이 가장 큰 가중치를 찾는 첫 번째 최적화 단계를 수행.
- Second Step : Loss landscape에서 (로컬적으로) 가장 높은 지점의 기울기로 원래 가중치를 업데이트하는 두 번째 최적화 단계를 수행.

아래는 에폭 수만큼 순회하며 학습시키는 코드이다.

```
for epoch in range(args['epochs']):
    model.train()
    log.train(len_dataset=len(train_dl))

    for batch in train_dl:
        inputs, targets = (b.to(device) for b in batch)

        # first forward-backward step
        enable_running_stats(model)
        predictions = model(inputs)
        loss = smooth_crossentropy(predictions, targets, smoothing=args['label_smoothing'])
        loss.mean().backward()
        optimizer.first_step(zero_grad=True)

        # second forward-backward step
        disable_running_stats(model)
        smooth_crossentropy(model(inputs), targets, smoothing=args['label_smoothing']).mean().backward()
        optimizer.second_step(zero_grad=True)

        with torch.no_grad():
            correct = torch.argmax(predictions.data, 1) == targets
            log(model, loss.cpu(), correct.cpu(), scheduler.lr())
            scheduler(epoch)

    model.eval()
    log.eval(len_dataset=len(val_dl))

    with torch.no_grad():
        for batch in val_dl:
            inputs, targets = (b.to(device) for b in batch)

            predictions = model(inputs)
            loss = smooth_crossentropy(predictions, targets)
            correct = torch.argmax(predictions, 1) == targets

            log(model, loss.cpu(), correct.cpu())
```

smooth_cross entropy는 일반화를 향상시키기 위한 정규화 테크닉인 라벨 스무딩을 적용한 크로스엔트로피이다.

아래는 학습시킬 때의 출력이다. 이러한 로그 출력 관련 코드는 utility 폴더로 불러와서 살짝 변형하여 사용했다.

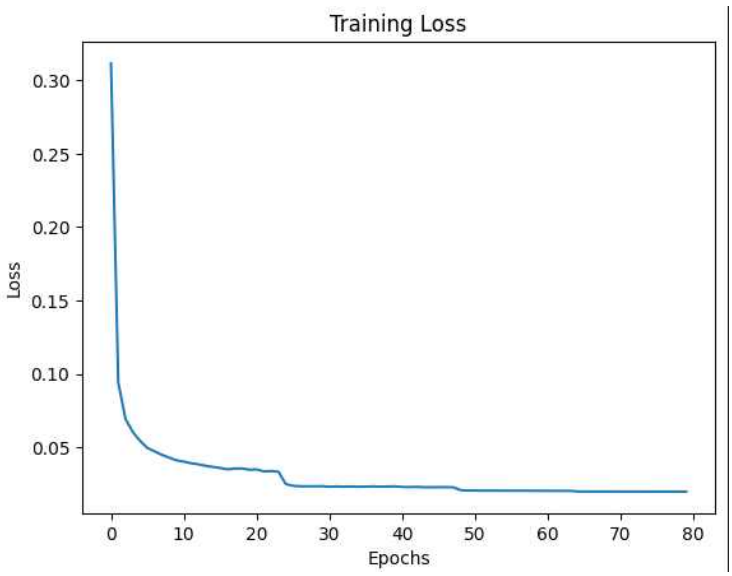
```
LeNet_5(
  (conv1): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (conv3): Conv2d(16, 120, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=120, out_features=84, bias=True)
  (fc2): Linear(in_features=84, out_features=10, bias=True)
)
```

	T-R-A-I-N		S-T-A-T-S		L-O-A-D-I-N-G	V-A-L-I-D	
epoch	loss	accuracy	l.r.	elapsed	bar	loss	accuracy
0	0.3116	88.41 %	1.000e-01	00:11 min		0.1192	96.27 %
1	0.0942	97.09 %	1.000e-01	00:09 min		0.0789	97.58 %
2	0.0693	97.98 %	1.000e-01	00:09 min		0.0677	98.07 %
3	0.0605	98.22 %	1.000e-01	00:09 min		0.0553	98.24 %
4	0.0544	98.43 %	1.000e-01	00:09 min		0.0504	98.34 %
5	0.0496	98.58 %	1.000e-01	00:09 min		0.0535	98.48 %
6	0.0473	98.64 %	1.000e-01	00:09 min		0.0490	98.54 %
7	0.0449	98.71 %	1.000e-01	00:09 min		0.0448	98.58 %
8	0.0430	98.80 %	1.000e-01	00:09 min		0.0467	98.47 %
9	0.0412	98.87 %	1.000e-01	00:09 min		0.0461	98.60 %
10	0.0403	98.90 %	1.000e-01	00:09 min		0.0497	98.49 %
11	0.0392	98.89 %	1.000e-01	00:09 min		0.0425	98.57 %
12	0.0385	98.93 %	1.000e-01	00:09 min		0.0411	98.70 %
13	0.0375	98.99 %	1.000e-01	00:09 min		0.0408	98.81 %
...							
76	0.0199	99.59 %	8.000e-04	00:09 min		0.0291	99.17 %
77	0.0199	99.57 %	8.000e-04	00:09 min		0.0291	99.17 %
78	0.0199	99.58 %	8.000e-04	00:09 min		0.0291	99.17 %
79	0.0199	99.59 %	8.000e-04	00:09 min		0.0291	99.16 %

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...

Training loss를 그래프로 출력해보았다.

LR 함수로 StepLR을 사용하여 아래의 loss 그래프를 보면 계단형으로 살짝 꺾이는 부분을 확인할 수 있다.



학습을 완료한 후, SAM을 이용한 LeNet-5의 Test 데이터셋에 대한 정확도를 측정해보았다.

1. 아무런 변형이 없는 Test dataset에 대해 정확도를 측정하였다.
정확도는 **99.00%**가 출력되었다.

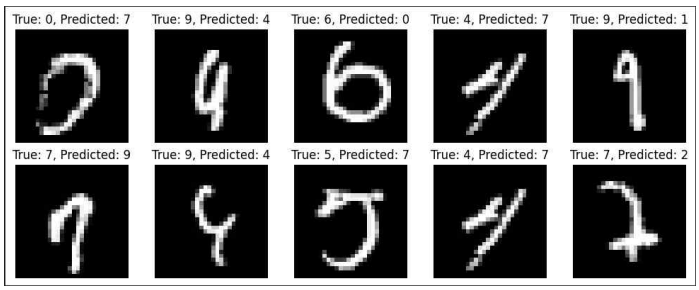
정확도 출력

```
correct_samples, wrong_samples = check_accuracy(test_dl, model)
```

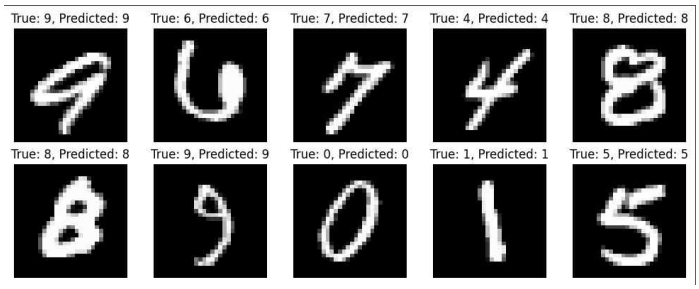
✓ 2.4s

```
Accuracy on the test set: 99.000%
```

아래는 잘못 분류한 샘플 이미지이다. (10개 랜덤 선택)



아래는 옳게 분류한 샘플 이미지이다. (10개 랜덤 선택)



2. 가우시안 노이즈(**표준편차 0.4**)를 적용한 Test dataset에 대해 정확도를 측정하였다.
정확도는 **98.60%**가 출력되었다. 마찬가지로 랜덤 노이즈 생성 시에 랜덤 시드를 지정하였다.

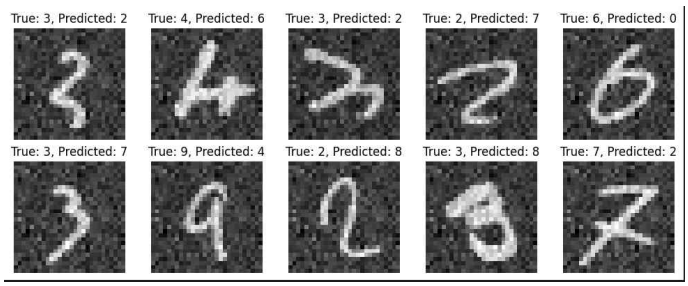
정확도 출력 (표준편차 0.4 가우시안 노이즈 적용)

```
correct_samples, wrong_samples = check_accuracy(gau_04_test_dl, model)
```

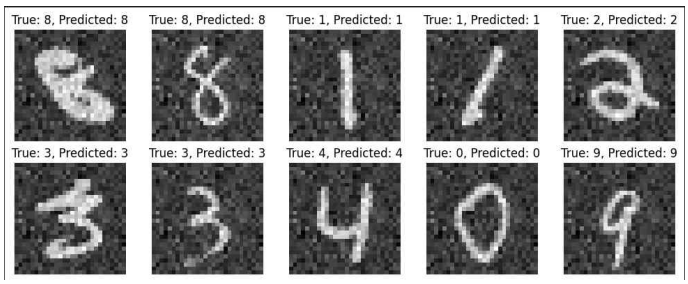
✓ 4.8s

```
Accuracy on the test set: 98.600%
```

아래는 잘못 분류한 샘플 이미지이다. (10개 랜덤 선택)

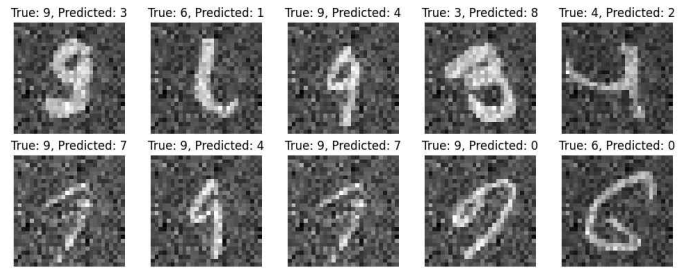


아래는 옳게 분류한 샘플 이미지이다. (10개 랜덤 선택)

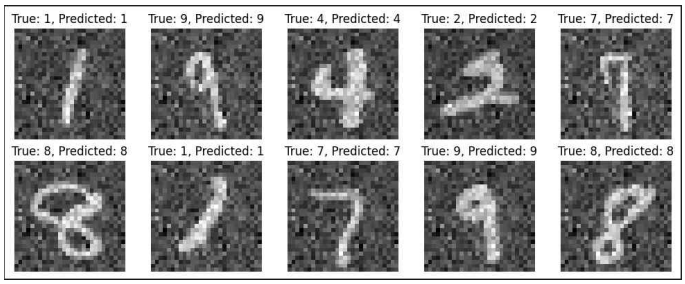


3. 가우시안 노이즈(**표준편차 0.6**)를 적용한 Test dataset에 대해 정확도를 측정하였다. 정확도는 **98.11%**가 출력되었다.

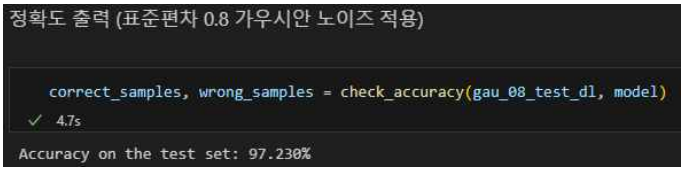
아래는 잘못 분류한 샘플 이미지이다. (10개 랜덤 선택)



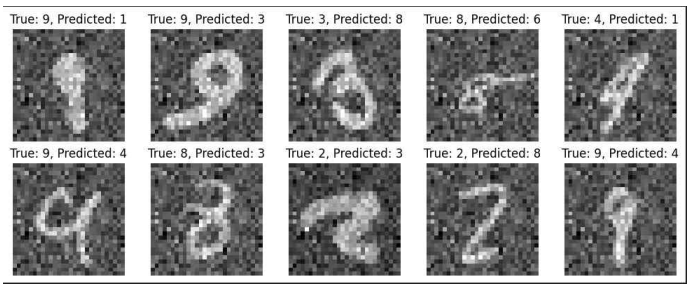
아래는 옳게 분류한 샘플 이미지이다. (10개 랜덤 선택)



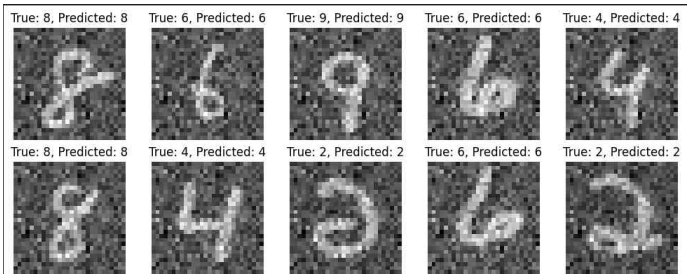
4. 가우시안 노이즈(**표준편차 0.8**)를 적용한 Test dataset에 대해 정확도를 측정하였다. 정확도는 **97.23%**가 출력되었다.



아래는 잘못 분류한 샘플 이미지이다. (10개 랜덤 선택)



아래는 옳게 분류한 샘플 이미지이다. (10개 랜덤 선택)



3. 기본 모델과 결과 비교 - 이경주

테스트 결과, 정확도는 아래와 같다.

- **기본형 LeNet-5**의 정확도
  * TEST 1. Pure dataset : 98.75%
  * TEST 2. Gaussian Noise dataset(standard deviation : 0.4) : 91.62%
  * TEST 3. Gaussian Noise dataset(standard deviation : 0.6) : 89.04%
  * TEST 4. Gaussian Noise dataset(standard deviation : 0.8) : 86.43%
- **개선한 LeNet-5 with SAM(Sharpness-aware minimization)**의 정확도
  * TEST 1. Pure dataset : 99%
  * TEST 2. Gaussian Noise dataset(standard deviation : 0.4) : 98.60%
  * TEST 3. Gaussian Noise dataset(standard deviation : 0.6) : 98.11%
  * TEST 4. Gaussian Noise dataset(standard deviation : 0.8) : 97.23%

Sharpness-aware minimization을 적용한 LeNet-5은 개선전 모델인 LeNet-5 기본형보다 약간의 정확도가 높게 나타났지만 크게 차이가 나진 않는다.

그런데 주목해야할 부분은, 일반화(Generalization) 측면에서의 성능을 보여주는 Noise 적용 Test 데이터셋에 대한 정확도이다.
기본형 LeNet-5의 경우에는 Noise가 적용된 Test 데이터셋에 대해서는 심한 성능저하를 보여준다. 표준편차 값이 커지는 노이즈 데이터셋일수록 더 심한 성능 저하를 보여준다.
반면에, Sharpness-aware minimization을 적용한 개선 버전 LeNet-5는 Noise가 적용된 Test 데이터셋에 대해서 기본형에 비해 약간의 성능저하를 보여주며, **일반화(Generalization) 성능에서 더 뛰어남을 보여준다**.

개인 깃허브 레포지토리에 코드와 설명을 업로드하였다.
https://github.com/gjlee0802/LeNet-5

4. 참고 자료 - 이경주

https://arxiv.org/pdf/2010.01412.pdf
Pierre Foret, Ariel Kleiner, Hossein Mobahi, and Behnam Neyshabur. Sharpness-aware minimization for efficiently improving generalization. In International Conference on Learning Representations, 2021.

https://github.com/davda54/sam
SAM: Sharpness-Aware Minimization (PyTorch)

II. 김인우 (20201791)

1. LeNet-5 기본 모델 구현 (개선전 구현) - 김인우

2. 개선 방안 제시 및 개선 시도 - 김인우

3. 기본 모델과 결과 비교 - 김인우

4. 참고 자료 - 김인우

1. LeNet-5 기본 모델 구현 (개선전 구현) - 김인우

```python
# LeNet5 모델 정의;

class LeNet5(nn.Module):
    def __init__(self):
        super(LeNet5, self).__init__()
        #input image size에 맞게끔 convolution 연산 진행 모델구현
        self.conv1 = nn.Conv2d(1, 6, kernel_size = 5, padding = 2, stride = 1)
        self.conv2 = nn.Conv2d(6, 16, kernel_size = 5, padding = 0, stride = 1)
        self.conv3 = nn.Conv2d(16, 120, kernel_size = 5, padding = 0, stride = 1)

        #풀링연산 (커널사이즈2x2, 보폭 2)
        self.pool = nn.MaxPool2d(kernel_size = 2, stride = 2, padding = 0)

        self.fc1 = nn.Linear(120, 84)
        self.fc2 = nn.Linear(84, 10)

        #출력층의 활성화함수로 log_softmax함수 사용
        self.logsoft = nn.LogSoftmax()
        #컨볼루션과 풀링 연산 사이에 ReLU함수 사용
        self.relu = nn.ReLU()

    def forward(self, x):
        #컨볼루션-풀링-컨볼루션-풀링-컨볼루션 구조의 기본적인 LeNet5모델 구현,
        x = self.conv1(x)
        x = self.relu(x)
        x = self.pool(x)

        x = self.conv2(x)
        x = self.relu(x)
        x = self.pool(x)

        x = self.conv3(x)
        x = self.relu(x)

        x = torch.flatten(x, start_dim=1)

        x = self.fc1(x)
        x = self.fc2(x)
        x = self.logsoft(x)

        return x
```

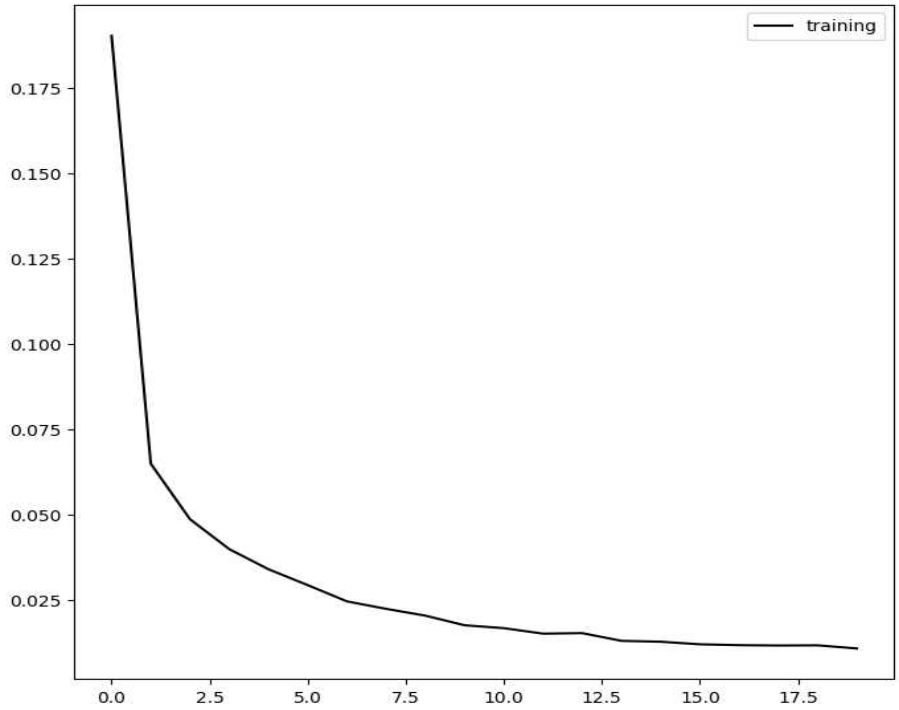
기본적인 LeNet5모델을 구현하였다. Input image size에 맞게끔 컨볼루션, 풀링 등에서 커널 사이즈 등을 고려하여 구현하였다. 은닉층 사이의 활성함수로는 ReLU함수를 사용하였고, 출력층에서는 log_softmax함수를 사용하였다.

```python
## 모델 학습과정
train_losses = []

for e in range(epoch):
    starttime =time.time()  #에폭 시간 측정을 위한 시작 시간
    epoch_loss = 0  #에폭당 로스값 저장.

    for inputs, labels in train_loader:  #data loader를 통한 학습데이터 load
        inputs, labels = inputs.to(device), labels.to(device)
        optimizer.zero_grad()     #0으로 초기화 후 진행,
        predicts = model(inputs)

        loss = nn.CrossEntropyLoss()(predicts, labels)  #손실함수로 crossEntropy 사용
        loss.backward()       # 오류역전파를 통한 옵티마이저 학습 진행,
        optimizer.step()
        epoch_loss += loss.item()

    epoch_loss /= len(train_loader)
    train_losses.append(epoch_loss)

    finishtime = time.time() - starttime  #에폭 소요시간 계산

    #에폭 손실값, 소요시간 출력,
    print(f"epoch : {e+1}, loss : {epoch_loss}, time per epoch : {finishtime}")


torch.save(model.state_dict(), "./lenetmodel.pth")
```

모델 학습과정 : 손실함수로는 CrossEntropy를 사용하여 지정한 에폭수만큼 오류역전파 등을 사용하여 최적화하는 코드이다. 각 에폭별로 loss값, 소요시간등을 출력한다. 다음은 출력결과이다.

```
epoch : 1, loss : 0.19021087537991505, time per epoch : 31.426242351531982
epoch : 2, loss : 0.06482819666867144, time per epoch : 30.002444744110107
epoch : 3, loss : 0.048549111404427095, time per epoch : 40.161439180374146
epoch : 4, loss : 0.03979700125887369, time per epoch : 34.377853870391846
epoch : 5, loss : 0.033897634581821816, time per epoch : 36.691229820251465
epoch : 6, loss : 0.029247968364345917, time per epoch : 35.07482957839966
epoch : 7, loss : 0.024486128172293925, time per epoch : 37.48995780944824
epoch : 8, loss : 0.022294487469384814, time per epoch : 38.92436337471008
epoch : 9, loss : 0.020301710967313556, time per epoch : 36.6633780002594
epoch : 10, loss : 0.01748068964108655, time per epoch : 27.130366802215576
epoch : 11, loss : 0.016634677096075697, time per epoch : 23.766648530960083
epoch : 12, loss : 0.015030243795257987, time per epoch : 24.23704433441162
epoch : 13, loss : 0.015177333124401124, time per epoch : 27.055598735809326
epoch : 14, loss : 0.012916772114430751, time per epoch : 24.17375874519348
epoch : 15, loss : 0.012663987977153379, time per epoch : 24.33322811126709
epoch : 16, loss : 0.011905347170843735, time per epoch : 26.536707639694214
epoch : 17, loss : 0.011654252550602442, time per epoch : 24.213143825531006
epoch : 18, loss : 0.01155155143219735, time per epoch : 24.09988260269165
epoch : 19, loss : 0.011604942509742757, time per epoch : 23.76769208908081
epoch : 20, loss : 0.010683375694008904, time per epoch : 23.847839832305908
```

이를 시각화하면 다음과 같다.

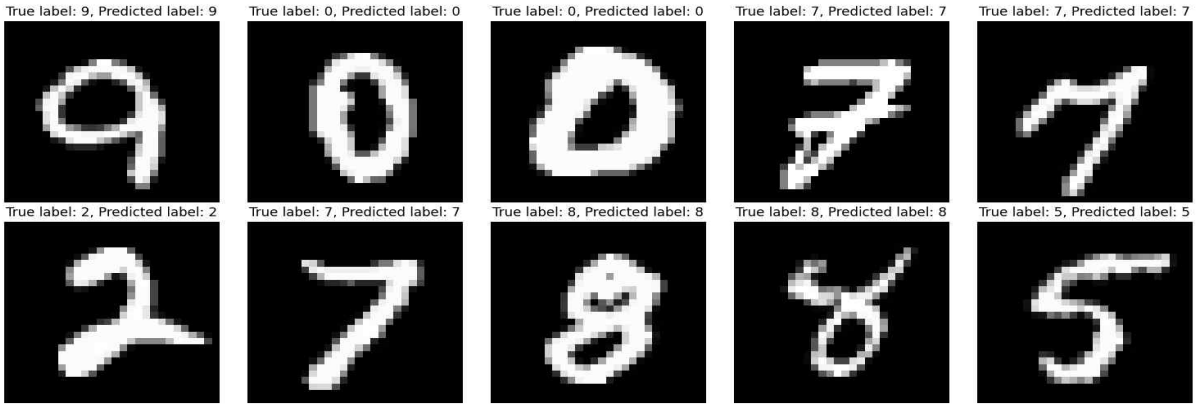


다음은 testdata를 활용한 평가코드이다.

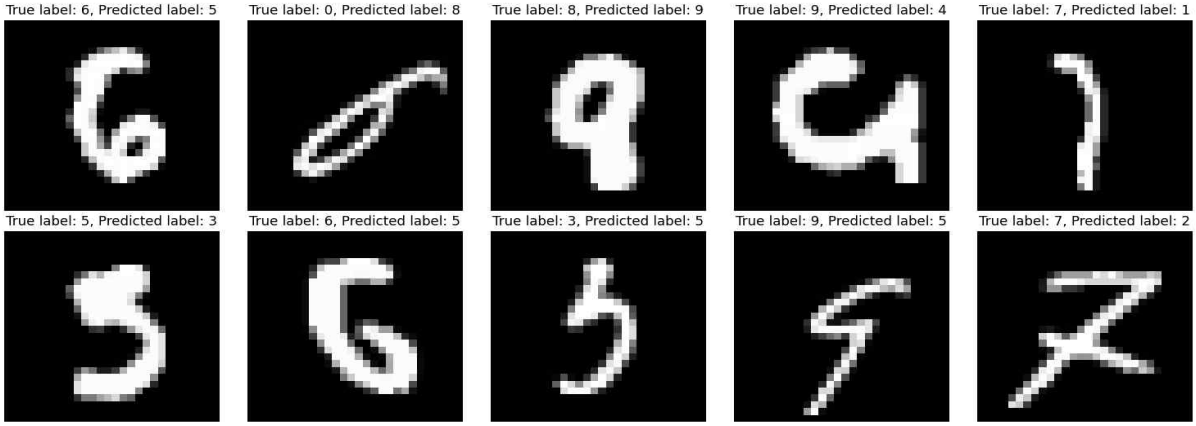
```python
#모델 평가 코드
model.eval()
total = 0
correct = 0  # 정답 개수를 기록하기 위한 변수를 초기화합니다.

with torch.no_grad():  # 그래디언트 변화 x (평가이므로)
    for inputs, labels in test_loader:
        inputs, labels = inputs.to(device), labels.to(device)

        infer = model(inputs)  # 모델에 인풋으로 test data를 넣어 평가를 진행하고
        predicted = infer.argmax(dim=1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()  # 맞는지 틀렸는지를 평가하고 맞은 경우에 추가한다.

        # 올바르게 분류된 샘플과 잘못 분류된 샘플을 구분하여 저장
        for i in range(len(predicted)):
            if predicted[i] == labels[i]:
                correct_lst.append((inputs[i], labels[i], predicted[i]))
            else:
                wrong_lst.append((inputs[i], labels[i], predicted[i]))

# 정확도 출력
accuracy = 100 * correct / total
print(f'Accuracy on the test set: {accuracy:.3f}%')
```

해당 코드를 통해 모델의 testdata를 넣어 추론을 하고, 결과와 라벨값을 비교해 정확도를 평가한다. 이 때, 맞은 샘플과 틀린 샘플을 구분하여 list에 저장하게 된다.

다음은 맞은 샘플 10개에 대해서 랜덤하게 출력한 결과이다.



다음은 틀린 샘플 10개에 대해서 랜덤하게 출력한 결과이다.



2. 개선 방안 제시 및 개선 시도 - 김인우

학습데이터에 대해 전처리 작업을 진행하여 모델의 성능을 높이려 시도하였다. 따라서, 배치사이즈, 에폭, 학습률, 옵티마이저 함수 등에 대해서는 고정하여 진행하고, 전처리 방법에 대해서만 변형을 주어 모델의 성능변화를 평가하였다. 전처리 방법으로는 Normalize 정규화를 통해 성능을 높이는 방법을 시도하여 보았고, RandomRotate등 학습데이터에 변형을 주어서 원본이미지를 크게 훼손시키지 않는 선에서 다양한 학습데이터로 모델을 학습하는 것을 시도하였다. 또한, 두 방법을 동시에 시도하여 보거나, 원본데이터 60000장 + RandomRotate등을 통한 변형데이터 60000장을 합쳐 총 120000장의 증강된 데이터를 통한 학습 또한 시도하여 보았다.

```python
#데이터 전처리 방법
augmented_transform = transforms.Compose([
    transforms.RandomRotation(5),  # 0 ~ 5도 사이의 랜덤한 값으로 이미지를 회전시키는 전처리
    transforms.ToTensor(),            #텐서화
    transforms.Normalize((0.5,),(0.5,))      # 정규화코드 (평균, 표준편차 지정)
])

transform = transforms.Compose([
    transforms.ToTensor(),#텐서화
    transforms.Normalize((0.5,),(0.5,)) # 정규화코드 (평균, 표준편차 지정)
])



# 학습데이터 > 60000장
train_data = datasets.MNIST(root='./data', train=True, transform=augmented_transform, download=True)
# 테스트데이터 > 10000장
test_data = datasets.MNIST(root='./data', train=False, transform=transform, download=True)
```

3. 기본 모델과 결과 비교 - 김인우

LeNet5 model accuracy 평가지표

공통 : learning rate = 0.001, batch_size = 32, epoch = 20 고정 학습

- 기본 LeNet5 모델 : 98.89,
- Normalize(0.5, 0.5) : 99.04,
- Rotation(5) : 99.16,
- Rotation(5) + Normalize(0.5, 0.5) : 98.96,
- Rotation(5)를 통한 12만장 학습 : 99.16

학습데이터에 RandomRotation 또는 정규화 작업을 진행한 후의 결과가 기본적인 LeNet5모델 보다 성능적인 부분에서 이점을 갖는 것을 확인할 수 있었다. 다만, random seed값을 주고 모델 학습을 진행하여도, 딥러닝 내부적인 학습과정에서 파라미터 설정 값등 값이 조금씩 달라지는 이유로 인해 매번 같은 값의 정확도를 보장해주지 않았다. 전체적인 성능평가를 위해서는 여러 번의 모델학습을 해보아야 정확한 성능평가를 할 수 있음을 알 수 있었다. 그러나, 기본적인 LeNet5구조의 모델을 학습했을 때보다, Rotation등의 전처리 과정을 추가하여 다양한 데이터셋에 대해서 학습을 진행하였을 때 모델 성능 부분에서 이점을 갖는 것은 확인할 수 있었다.

4. 참고 자료 - 김인우

Survey paper
Open access
Published: 06 July 2019
A survey on Image Data Augmentation for Deep Learning

https://journalofbigdata.springeropen.com/articles/10.1186/s40537-019-0197-0

데이터 증강에 따른 딥러닝 학습에 관한 논문 자료 및 preprocessing for CNN에 관한 자료들을 찾아보았다.

III. 최현우 (20223527)

1. LeNet-5 기본 모델 구현 (개선전 구현) - 최현우

2. 개선 방안 제시 및 개선 시도 - 최현우

3. 기본 모델과 결과 비교 - 최현우

4. 참고 자료 - 최현우

1. LeNet-5 기본 모델 구현 (개선전 구현) - 최현우

```python
class LeNet5(nn.Module):
    def __init__(self):
        super(LeNet5, self).__init__()
        # 첫 번째 합성곱 계층: 1개의 입력 채널, 6개의 출력 채널, 5x5 커널 크기
        self.conv1 = nn.Conv2d(1, 6, kernel_size=5, padding=2)
        self.pool1 = nn.AvgPool2d(kernel_size=2, stride=2)
        # 두 번째 합성곱 계층: 6개의 입력 채널, 16개의 출력 채널, 5x5 커널 크기
        self.conv2 = nn.Conv2d(6, 16, kernel_size=5)
        self.pool2 = nn.AvgPool2d(kernel_size=2, stride=2)
        # 완전 연결 계층
        self.fc1 = nn.Linear(16*5*5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # 첫 번째 합성곱 및 풀링 계층을 통과
        x = torch.tanh(self.conv1(x)) # 첫 번째 계층에서 활성화 함수로 tanh 사용
        x = self.pool1(x)
        # 두 번째 합성곱 및 풀링 계층을 통과
        x = torch.tanh(self.conv2(x))
        x = self.pool2(x)
        x = x.view(-1, 16*5*5)
        x = torch.tanh(self.fc1(x))
        x = torch.tanh(self.fc2(x))
        x = self.fc3(x)
        return torch.log_softmax(x, dim=1) # 분류 결과를 log_softmax로 출력하여 확률 계산
```

컨볼루션, 풀링, FC를 image에 맞게끔 설정하고 활성화 함수로 tanh를 사용하였다. 양식에 맞게 출력층에 log_sofrmax 활성 함수를 넣어주었다.

```python
# 모델, 손실 함수, 최적화 알고리즘 설정
model = LeNet5()
criterion = nn.NLLLoss()  # 음의 로그 가능도 손실 함수
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)  # 옵티마이저 SGD, 학습률 0.01


epoch_losses = []  # 에포크별 손실을 저장할 리스트
for epoch in range(1, 11):
    model.train()
    total_loss = 0
    start_time = time.time()
    for batch_idx, (data, target) in enumerate(train_loader):
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    end_time = time.time()
    epoch_loss = total_loss / len(train_loader)
    epoch_losses.append(epoch_loss)  # 에포크별 손실을 리스트에 추가
    print(f'Epoch {epoch}, Training loss: {epoch_loss:.4f}, Time: {end_time - start_time:.2f}s')
```

```python
# MNIST 데이터셋 로드
train_dataset = datasets.MNIST(root='./data', train=True, download=True, transform=transform)
test_dataset = datasets.MNIST(root='./data', train=False, download=True, transform=transform)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)
```

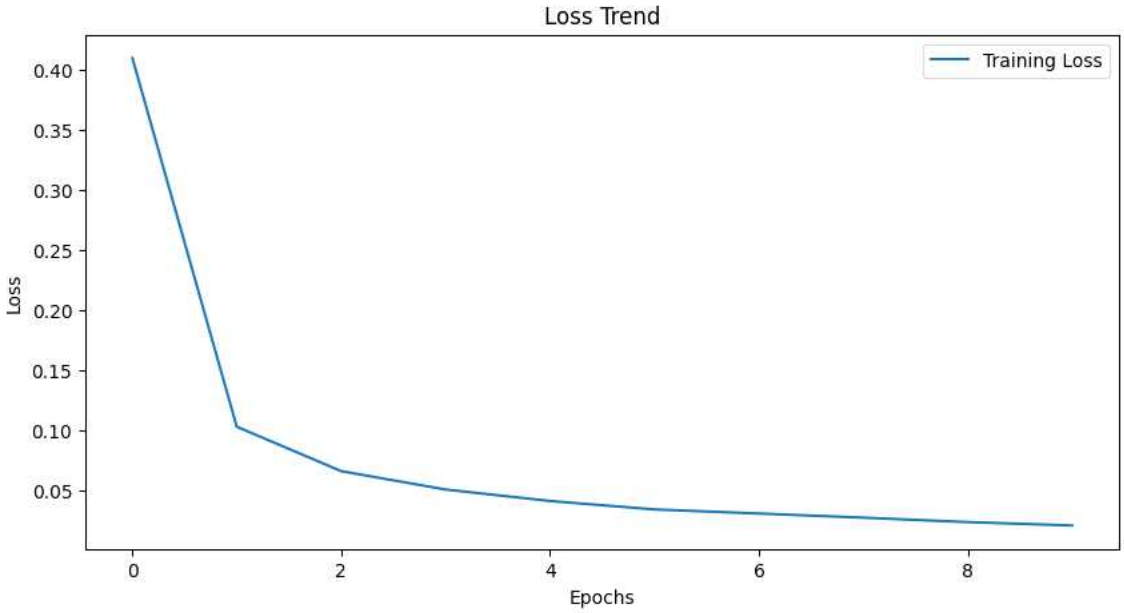
- epoch: 10
- Loss 함수: NLLLoss
- 옵티마이저: SGD
- 학습률 = 0.01
- batch size = 64

이 설정으로 초기 LeNet-5를 구현하였다. 기본적인 LeNet-5의 정확도를 측정하기 위해 성능을 향상 시키는 부분들은 제외하였다.

```
epoch_losses = []  # 에포크별 손실을 저장할 리스트
for epoch in range(1, 31):
    model.train()
    total_loss = 0
    start_time = time.time()
    for batch_idx, (data, target) in enumerate(train_loader):
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    scheduler.step()
    end_time = time.time()
    epoch_loss = total_loss / len(train_loader)
    epoch_losses.append(epoch_loss)  # 에포크별 손실을 리스트에 추가
    print(f'Epoch {epoch}, Training loss: {epoch_loss:.4f}, Time: {end_time - start_time:.2f}s')
```

학습 중에 1 epoch 단위로 수행시간을 출력하였다.

```
Epoch 1, Training loss: 0.3808, Time: 41.86s
Epoch 2, Training loss: 0.0921, Time: 37.99s
Epoch 3, Training loss: 0.0606, Time: 37.51s
Epoch 4, Training loss: 0.0482, Time: 38.54s
Epoch 5, Training loss: 0.0409, Time: 37.96s
Epoch 6, Training loss: 0.0346, Time: 37.48s
Epoch 7, Training loss: 0.0307, Time: 37.90s
Epoch 8, Training loss: 0.0269, Time: 38.24s
Epoch 9, Training loss: 0.0250, Time: 37.05s
Epoch 10, Training loss: 0.0216, Time: 37.48s
```



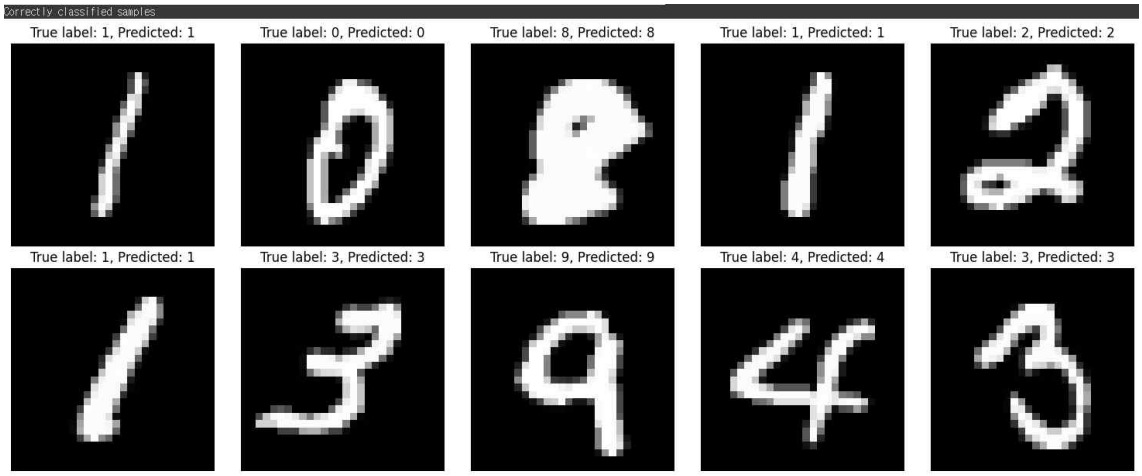
학습이 끝난 후에 cost function 추세를 그래프로 출력한 화면이다.
위 그래프처럼 학습 중 Loss가 줄어드는 것을 알 수 있다.

```
# 임의로 선택한 10개의 올바르게 분류된 샘플 표시
if len(correct_lst) >= 10:
    random_correct_samples = random.sample(correct_lst, 10)
    visualize_samples(random_correct_samples, "Correctly classified samples")

# 임의로 선택한 잘못 분류된 표본 10개 표시
if len(wrong_lst) >= 10:
    random_wrong_samples = random.sample(wrong_lst, 10)
    visualize_samples(random_wrong_samples, "Wrongly classified samples")
```

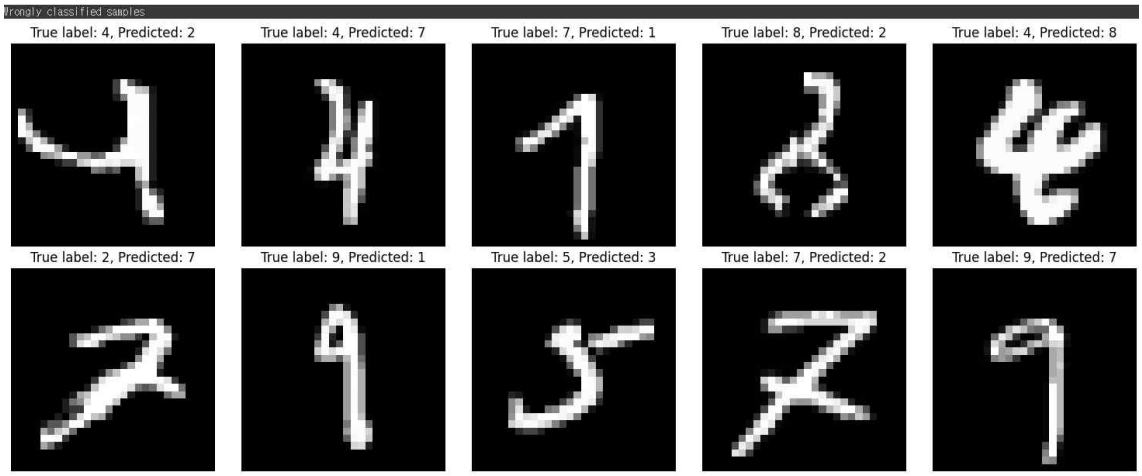
```
def visualize_samples(samples, title):
    print(title)
    plt.figure(figsize=(15, 6))
    for i in range(10):
        input_img, true_label, predicted_label = samples[i]
        plt.subplot(2, 5, i + 1)
        plt.imshow(input_img.squeeze().cpu().numpy(), cmap='gray')
        plt.title(f'True label: {true_label}, Predicted: {predicted_label}')
        plt.axis('off')
    plt.tight_layout()
    plt.show()
```

정확하게 분류한 10개 sample과 잘못 분류된 10개의 sample을 랜덤하게 선택해서 화면에 출력하는 코드이다.

다음 사진은 정확하게 분류한 10개의 sample을 랜덤하게 출력한 결과이다.



다음 사진은 잘못 분류한 10개의 sample을 랜덤하게 출력한 결과이다.



2. 개선 방안 제시 및 개선 시도 - 최현우

```
class LeNet5(nn.Module):
    def __init__(self):
        super(LeNet5, self).__init__()
        # 합성곱 계층 및 풀링 계층을 통해 이미지의 특징을 추출
        self.conv1 = nn.Conv2d(1, 6, kernel_size=5, padding=2)
        self.bn1 = nn.BatchNorm2d(6) # 배치 정규화
        self.pool1 = nn.AvgPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(6, 16, kernel_size=5)
        self.bn2 = nn.BatchNorm2d(16)
        self.pool2 = nn.AvgPool2d(kernel_size=2, stride=2)
        self.fc1 = nn.Linear(16*5*5, 120)
        self.dropout1 = nn.Dropout(0.5) # 드롭아웃 추가
        self.fc2 = nn.Linear(120, 84)
        self.dropout2 = nn.Dropout(0.5)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = torch.tanh(self.bn1(self.conv1(x))) # 첫 번째 계층에서 활성화 함수로 tanh 사용
        x = self.pool1(x)
        x = torch.tanh(self.bn2(self.conv2(x)))
        x = self.pool2(x)
        x = x.view(-1, 16*5*5)
        x = torch.tanh(self.fc1(x))
        x = self.dropout1(x)
        x = torch.tanh(self.fc2(x))
        x = self.dropout2(x)
        x = self.fc3(x)
        return torch.log_softmax(x, dim=1) # 분류 결과를 log_softmax로 출력하여 확률 계산
```

기존의 LeNet-5를 구현하는 코드에 추가적으로 배치 정규화와 드롭아웃을 추가해주었다.

```
def add_noise(data, noise_factor=0.3): # 노이즈 추가
    noise = torch.randn_like(data) * noise_factor
    noisy_data = data + noise
    return torch.clamp(noisy_data, 0., 1.)
```

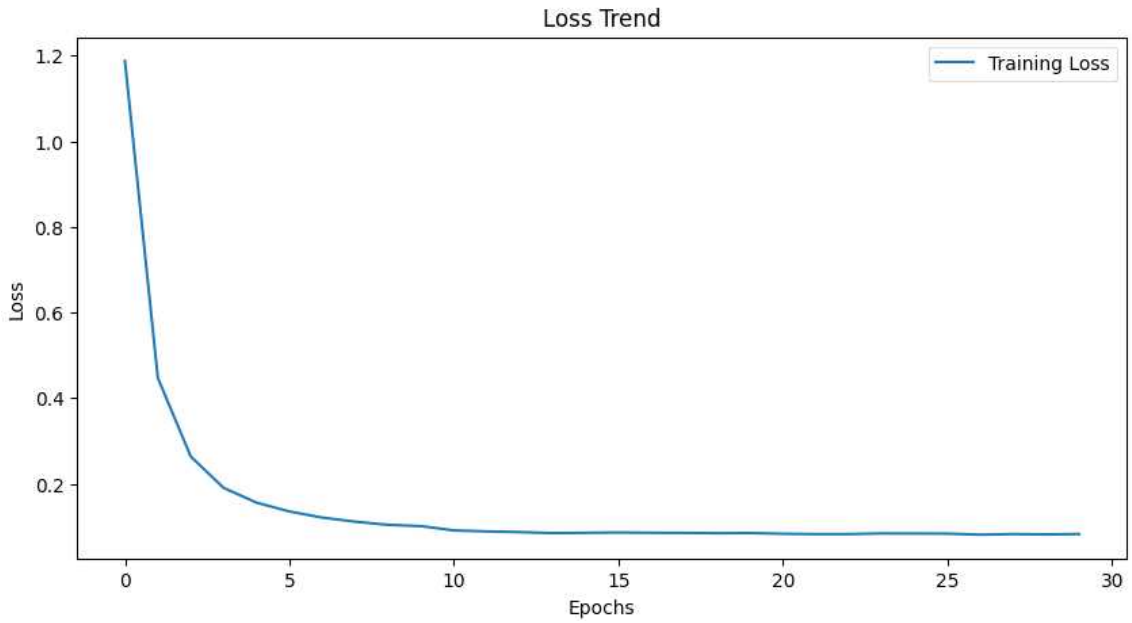
노이즈를 추가하여 단순 정확도만 올리는 것이 아닌 일반화 능력을 향상 시켰다.

```
model = LeNet5()
criterion = nn.NLLLoss() # 손실 함수: 음의 로그 우도 손실(Negative Log Likelihood Loss)
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9, weight_decay=0.0005) # 가중치 감소 추가, SGD 옵티마이저, 학습률과 모멘텀 설정
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.1) # 학습률 스케줄러
```

학습률을 0.01에서 0.001로 변형 하였고 가중치를 추가하여 손실율을 낮추었다.
또한 학습률 스케줄러를 통해 손실 함수의 미세한 조정을 가능하게 하여 성능 향상에 도움을 주었다.

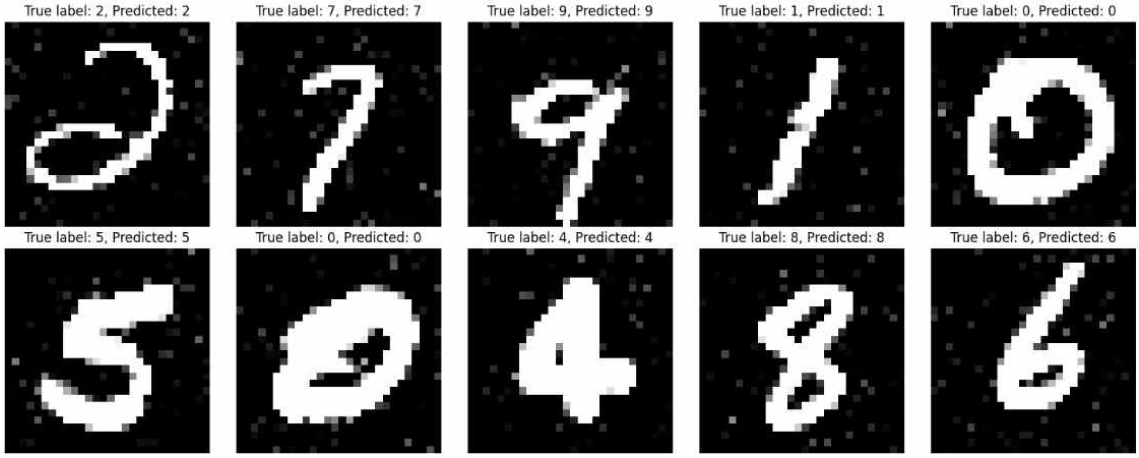
```
Epoch 1, Training loss: 1.1876, Time: 42.12s
Epoch 2, Training loss: 0.4471, Time: 41.61s
Epoch 3, Training loss: 0.2633, Time: 41.16s
Epoch 4, Training loss: 0.1900, Time: 41.09s
Epoch 5, Training loss: 0.1556, Time: 41.97s
Epoch 6, Training loss: 0.1349, Time: 41.10s
Epoch 7, Training loss: 0.1206, Time: 41.53s
Epoch 8, Training loss: 0.1110, Time: 41.08s
Epoch 9, Training loss: 0.1037, Time: 41.76s
Epoch 10, Training loss: 0.1006, Time: 41.66s
Epoch 11, Training loss: 0.0907, Time: 41.24s
Epoch 12, Training loss: 0.0883, Time: 41.06s
Epoch 13, Training loss: 0.0867, Time: 42.10s
Epoch 14, Training loss: 0.0845, Time: 41.01s
Epoch 15, Training loss: 0.0851, Time: 40.79s
Epoch 16, Training loss: 0.0859, Time: 41.26s
Epoch 17, Training loss: 0.0851, Time: 39.93s
Epoch 18, Training loss: 0.0850, Time: 39.77s
Epoch 19, Training loss: 0.0841, Time: 39.55s
Epoch 20, Training loss: 0.0845, Time: 39.60s
Epoch 21, Training loss: 0.0828, Time: 39.66s
Epoch 22, Training loss: 0.0820, Time: 39.75s
Epoch 23, Training loss: 0.0821, Time: 39.39s
Epoch 24, Training loss: 0.0836, Time: 39.76s
Epoch 25, Training loss: 0.0834, Time: 39.74s
Epoch 26, Training loss: 0.0831, Time: 39.39s
Epoch 27, Training loss: 0.0807, Time: 39.62s
Epoch 28, Training loss: 0.0821, Time: 40.13s
Epoch 29, Training loss: 0.0813, Time: 39.60s
Epoch 30, Training loss: 0.0823, Time: 39.45s
Test Accuracy: 98.72%
```

또한 epoch 수를 10에서 30으로 늘려 충분한 학습을 시켰고 성능을 향상 시켰다.

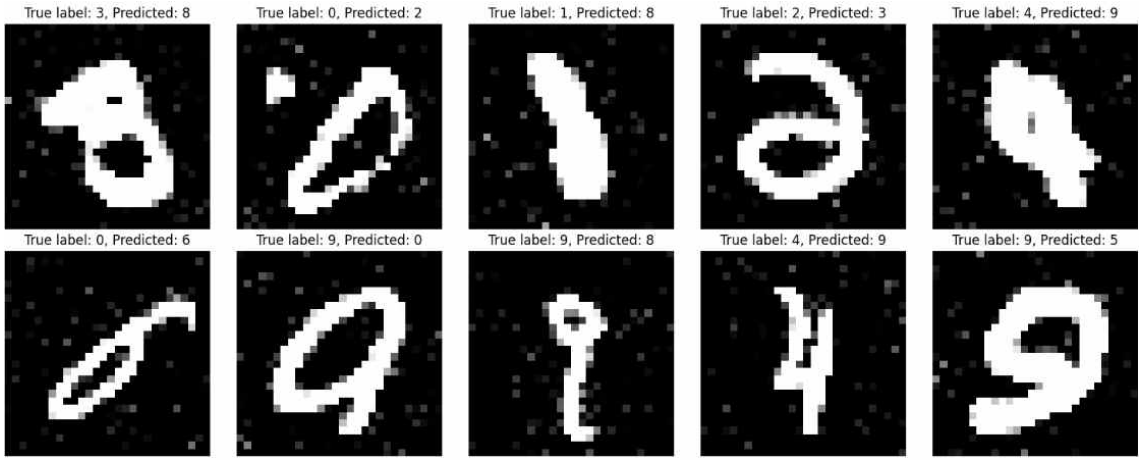


다음 사진은 hyperparameter를 조정하여 정확하게 분류한 10개 sample들을 랜덤하게 출력

한 화면이다.



다음 사진은 hyperparameter를 조정하여 잘못 분류한 10개 sample들을 랜덤하게 출력한 화면이다.



3. 기본 모델과 결과 비교 - 최현우

```
batch = 64
학습률 = 0.01
에폭 = 10
옵티마이저 = SGD
정확도 = 96.82%
------------------------

batch = 128
학습률 = 0.001
에폭 = 20
옵티마이저 = ADM
정확도 = 98.35%
------------------------

batch = 128
학습률 = 0.001
에폭 = 20
옵티마이저 = ADM
노이즈 + 전처리
정확도 = 98.31%
        = 98.42%
------------------------

batch = 128
학습률 = 0.001
에폭 = 20
옵티마이저 = ADM
노이즈 + 전처리
배치 정규화 + 드롭아웃 => 손실율 감소
정확도 = 98.68%
        = 98.84%
------------------------
```

```
batch = 128
학습률 = 0.001
에폭 = 30
옵티마이저 = ADM
노이즈 추가 + 전처리
스케줄러 조정
배치 정규화 + 드롭아웃 => 손실율 감소
정확도 = 96.59%
         = 96.84%
------------------------

batch = 128
학습률 = 0.001
에폭 = 30
옵티마이저 = PMSProp
노이즈 감소 + 전처리
스케줄러 조정
배치 정규화 + 드롭아웃 + 가중치 감소 => 손실율 감소
정확도 = 95.74%
         = 97.04%

------------------------
**********
batch = 64
학습률 = 0.001
에폭 = 30
옵티마이저 = SGD
손실함수: 음의 로그 우도 손실(Negative Log Likelihood Loss)
노이즈 감소 + 전처리
스케줄러 조정
배치 정규화 + 드롭아웃 + 가중치 감소 => 손실율 감소
정확도 = 98.72%
**********
------------------------
batch = 64
학습률 = 0.001
에폭 = 30
옵티마이저 = SGD
손실함수: 음의 로그 우도 손실(Negative Log Likelihood Loss)
정확도 = 98.47%
------------------------
```

- 기본형 LeNet-5의 정확도(accuracy): 96.82%
- 개선한 LeNet-5의 정확도(accuracy): 98.72%

hyperparameter 조정 및 성능 향상 조건을 추가했을 때의 정확도가 더 높아진 것을 확인할 수 있다. 또한 출력 그래프의 개형 및 sample들의 개형 및 정확도가 기본의 모델과 비교해봤을 때 세밀한 걸 시각적으로 확인할 수 있다.

4. 참고 자료 - 최현우

1. https://limepencil.tistory.com/4 [LeNet-5으로 더욱더 정확한 손글씨 분류기 만들기: 연두색연필]
2. https://brave-greenfrog.tistory.com/45 [[딥러닝 |CNN] - LeNet-5: kkulzi]
3. https://resultofeffort.tistory.com/103 [[pytorch] 이미지 분류를 위한 LeNet-5 구현: 독립성이 강한 ISFP]

IV. Summary

사전에 각자 분류기의 성능을 높이기 위한 방안을 생각하는 방향성을 다르게 정하여 역할을 분담하고 실습을 진행했다. 역할 분담은 아래와 같이 하였다.

<역할 분담>

- **이경주: 일반화 성능**에 초점을 맞추어 **Sharpness-aware minimization을 활용**한 일반화 (generalization) 성능 개선 시도.

- **최현우:** 딥러닝 모델 학습 과정 중 사용자가 결정해야하는 **하이퍼파라미터 (배치사이즈, 에폭, 학습률) 등을 조정**하여 최적의 하이퍼파라미터를 찾는것에 초점을 맞추어 개선 시도.

- **김인우:** preprocessing 방법 중 정규화 + 이미지 **변형등을 사용하여 학습**, **데이터 증강**을 통해 더 많은 데이터셋으로 개선 시도.

우리 나반 4조는 위처럼 다른 방향성으로 실험을 해보기로 역할 분담을 적절히 한 덕분에 다양한 실험이 가능했다.

이경주 조원은

Sharpness-aware minimization을 적용한 개선 버전 LeNet-5는 Noise가 적용된 Test 데이터셋에 대해서 약간의 성능저하를 보여주며, **일반화(Generalization) 성능에서 기본 LeNet-5 모델보다 더 뛰어남을 보여준다는 것을 확인했다.** (노이즈 생성 시 랜덤 시드 지정함)

- **기본형 LeNet-5**의 정확도
  * TEST 1. Pure dataset : 98.75%
  * TEST 2. Gaussian Noise dataset(standard deviation : 0.4) : 91.62%
  * TEST 3. Gaussian Noise dataset(standard deviation : 0.6) : 89.04%
  * TEST 4. Gaussian Noise dataset(standard deviation : 0.8) : 86.43%
- **개선한 LeNet-5 with SAM(Sharpness-aware minimization)**의 정확도
  * TEST 1. Pure dataset : 99%
  * TEST 2. Gaussian Noise dataset(standard deviation : 0.4) : 98.60%
  * TEST 3. Gaussian Noise dataset(standard deviation : 0.6) : 98.11%
  * TEST 4. Gaussian Noise dataset(standard deviation : 0.8) : 97.23%

최현우 조원은

다양한 하이퍼파라미터를 조정해보며 여러번의 성능 테스트를 해보았으며, **에폭 수나 손실함수, 옵티마이저를 다양하게 테스트해보고 정확도가 어떻게 달라지는지 확인했다.**

- 기본형 LeNet-5의 정확도(accuracy): 96.82%
- 개선한 LeNet-5의 정확도(accuracy): 98.72% (SGD + NLL + epoch 30 + batch 64)

김인우 조원은

기본적인 LeNet5구조의 모델을 학습했을 때보다, **Rotation등의 전처리 과정을 추가하여 다양한 데이터셋에 대해서 학습을 진행하였을 때 모델 성능 부분에서 이점을 갖는 것을 확인했다.**

- Normalize 전처리 과정으로 : 98.89% (전처리 전) -> 99.04% (전처리 후)
- Rotation의 전처리 과정으로 : 98.89% (전처리 전) -> 99.16% (전처리 후)