두산 Rokey Boot Camp

스터디 주간 활동 보고서

팀명	벌꿀오소리	**제출자 성명**	정찬원
참여 명단	정찬원, 임소정, 송주훈, 강인우		
모임 일시	**2025년 03월 06일 21시 ~ 22시 00분(총 1시간)**		
장소	온라인(화상)	**출석 인원**	4/6 정민섭님 개인 사정으로 화상미팅 미참여 (스터디활동에는 참여)
학습목표	교재 7단원 '다중분류', 8단원 'MNIST를 활용한 숫자 인식', 9단원 'CNN을 활용한 이미지 인식'을 파트별로 분배하여 정리하고 발표한다. 7단원: 서준원, 강인우 8단원: 송주훈, 정민섭 9단원: 정찬원, 임소정		
학습내용	**7단원 '다중분류'** (강인우) - 개요 데이터를 정해진 클래스 중 하나로 분류하며 보통 세 가지 이상의 클래스를 분류하는데 쓰이며 각 클래스에 대한 확률값을 출력하고, 가장 높은 확률을 가진 클래스를 선택함. $J(\theta) = -\sum_{i=1}^{n} y_i \log(h_\theta(x_i))$ $where, h_\theta(x_i) = \frac{e^{zi}}{\sum_{j=1}^{k} e^{z_j}}, k = 1, 2, \dots M$		

- 소프트맥스 함수

이진 분류에서 출력층에 시그모이드 함수가 있다면 다중분류에서는 소프트맥스 함수 각 클래스에 대한 확률 제공.

예시) cat, dog, horse클래스 [5, 4, 2] → [0.71, 0.26, 0.04]

- 크로스엔트로피 손실함수

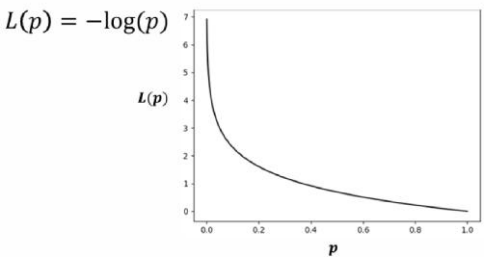
정답 라벨과 모델 예측 확률(소프트맥스 결과)를 비교하는 손실 함수

예시) 정답y를 원핫 벡터로 표현: [1, 0, 0]

J = -(1 * log0.71 + 0*log0.26 + 0*log0.04) = -log0.71 ≈0.34

- NLL (Negative Log-Likelihood)

모델의 출력 확률과 실제 클래스 간의 차이를 측정하는 손실함수를 의미하며 예측이 정확해질수록 NLL 값이 작아짐.



로그함수를 x축(p축)에 대칭한 형태이며 p는 확률이므로 p값의 범위는 0~1.

- 다중 분류 정확도

전체 샘플 중에서 정답을 맞춘 비율

실제값 \ 예측값	Category 1	Category 2	Category 3
Category 1	6	4	2
Category 2	3	6	3
Category 3	2	5	8

- 주요 메서드, 속성

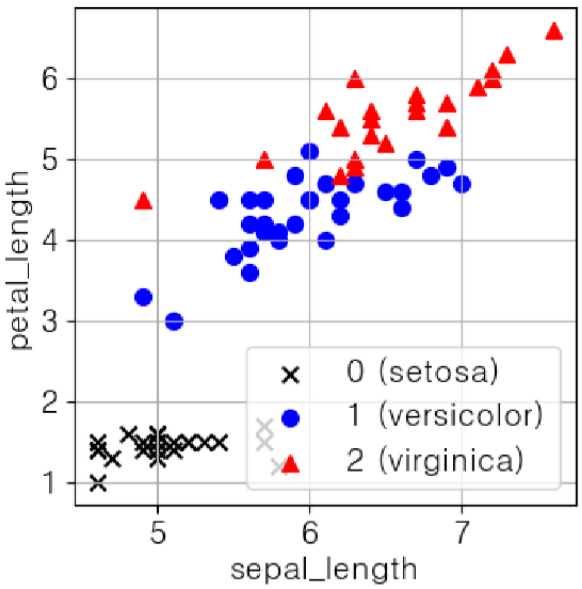
메서드/속성	설명
load_iris() iris.data iris.target	붓꽃(iris) 데이터셋을 불러오는 함수 각 데이터의 정답(품종 레이블, 0, 1, 2)
train_test_split()	학습 데이터와 테스트 데이터를 분할
nn.CrossEntropyLoss()	크로스 엔트로피 손실 함수 객체를 생성
nn.Softmax()	소프트맥스 함수 생성
nn.LogSoftmax()	로그소프트맥스 함수 생성 (Softmax값에 로그 적용)
nn.NLLLoss()	NLL 함수 생성
torch.log()	(자연)로그 함수 생성

- 붓꽃 데이터 산포도 코드

```
from sklearn.datasets import load_iris

# 데이터 불러오기
iris = load_iris()

# 입력 데이터와 정답 데이터
x_org, y_org = iris.data, iris.target

# 입력 데이터로 sepal(꽃받침) length(0)와 petal(꽃잎) length(2)를 추출
x_select = x_org[:,[0,2]]
```

```
# 훈련 데이터와 검증 데이터로 분할(셔플도 동시에 실시함)
from sklearn.model_selection import train_test_split

x_train, x_test, y_train, y_test = train_test_split(
    x_select, y_org, train_size=75, test_size=75,
    random_state=123)
```

```
x_t0 = x_train[y_train == 0]
x_t1 = x_train[y_train == 1]
x_t2 = x_train[y_train == 2]


plt.scatter(x_t0[:,0], x_t0[:,1], marker='x', c='k', s=50, label='0 (setosa)')
plt.scatter(x_t1[:,0], x_t1[:,1], marker='o', c='b', s=50, label='1 (versicolor)')
plt.scatter(x_t2[:,0], x_t2[:,1], marker='^', c='r', s=50, label='2 (virginica)')
plt.xlabel('sepal_length')
plt.ylabel('petal_length')
plt.legend()
plt.show()
```



- 다중 분류 모델

```
## 모델 정의 및 생성
class Net(nn.Module):
```

```
    def __init__(self, n_input, n_output):
        super().__init__()
        self.l1 = nn.Linear(n_input, n_output)

    def forward(self, x):
        x1 = self.l1(x)
        return x1

n_input = x_train.shape[1]
n_output = len(list(set(y_train)))

net = Net(n_input, n_output)

## 변수 텐서화
inputs = torch.tensor(x_train).float()
labels = torch.tensor(y_train).long()

inputs_test = torch.tensor(x_test).float()
labels_test = torch.tensor(y_test).long()


lr = 0.01
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=lr)
num_epochs = 10000
history = np.zeros((0,5))
```

```
for epoch in range(num_epochs):

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

    # 예측 라벨 산출
    predicted = torch.max(outputs, 1)[1]
```

```
    # 손실과 정확도 계산
    train_loss = loss.item()
    train_acc = (predicted == labels).sum()  / len(labels)

    outputs_test = net(inputs_test)
    loss_test = criterion(outputs_test, labels_test)
    predicted_test = torch.max(outputs_test, 1)[1]
    # predicted_test= outputs_test.argmax(-1)   # 이렇게 써도 됨

    val_loss =  loss_test.item()
    val_acc =  (predicted_test == labels_test).sum() / len(labels_test)

    if ((epoch) % 10 == 0):
        print (f'Epoch [{epoch}/{num_epochs}], loss: {train_loss:.5f} acc: {tra
        item = np.array([epoch, train_loss, train_acc, val_loss, val_acc])
        history = np.vstack((history, item))
```

```
    ## 각 행에서 가장 높은 점수를 가진 클래스의 인덱스
    # torch.max(outputs, 1) 각 행에서 가장 큰 값과 그 인덱스
    predicted = torch.max(outputs, 1)[1]

    ## 손실과 정확도 계산
    train_loss = loss.item()
    train_acc = (predicted == labels).sum()  / len(labels)

        ## 테스트 데이터에 대한 예측과 평가
    outputs_test = net(inputs_test)
    loss_test = criterion(outputs_test, labels_test)
    predicted_test = torch.max(outputs_test, 1)[1]

        ## 테스트 데이터 손실과 정확도 계산
    val_loss =  loss_test.item()
    val_acc =  (predicted_test == labels_test).sum() / len(labels_test)
```

predicted == labels

```
predicted = torch.tensor([1, 0, 2, 1, 0])
labels = torch.tensor([1, 0, 1, 1, 2])

print(predicted == labels)  # tensor([True, True, False, True, False])
print((predicted == labels).sum())  # 3
print((predicted == labels).sum() / len(labels))  # 0.6 (정확도)
```

- 다중 분류 모델

*입력 변수가 두개 일때 → x_select

```
# 입력 데이터와 정답 데이터
x_org, y_org = iris.data, iris.target

# 입력 데이터로 sepal(꽃받침) length(0)와 petal(꽃잎) length(2)를 추출
x_select = x_org[:,[0,2]]

x_train, x_test, y_train, y_test = train_test_split(
    x_select, y_org, train_size=75, test_size=75,
    random_state=123)
```

▼ 전체코드

```
x_train, x_test, y_train, y_test = train_test_split(
    x_org, y_org, train_size=75, test_size=75,
    random_state=123)

n_input = x_train.shape[1]

inputs = torch.tensor(x_train).float()
```

```
labels = torch.tensor(y_train).long()
inputs_test = torch.tensor(x_test).float()
labels_test = torch.tensor(y_test).long()

lr = 0.01

net = Net(n_input, n_output)
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=lr)
num_epochs = 10000

history = np.zeros((0,5))
```

```
for epoch in range(num_epochs):


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

```
# 예측 라벨 산출
predicted = torch.max(outputs, 1)[1]

# 손실과 정확도 계산
train_loss = loss.item()
train_acc = (predicted == labels).sum()  / len(labels)

# 예측 계산
outputs_test = net(inputs_test)
loss_test = criterion(outputs_test, labels_test)
predicted_test = torch.max(outputs_test, 1)[1]

val_loss =  loss_test.item()
val_acc =  (predicted_test == labels_test).sum() / len(labels_test)

if ( epoch % 10 == 0):
```

```
print (f'Epoch [{epoch}/{num_epochs}], loss: {train_loss:.5f} acc: {tra
item = np.array([epoch , train_loss, train_acc, val_loss, val_acc])
history = np.vstack((history, item))
```

- NLL Loss 함수

입력데이터 / 정답레이블 생성 - 텐서화 - NLL손실함수 정의 -

```
outputs_np = np.array(range(1, 13)).reshape((4,-1))

# 더미 정답 데이터(샘플의 정답 클래스 인덱스)
labels_np = np.array([0, 1, 2, 0])

# 텐서화
outputs_dummy = torch.tensor(outputs_np).float()    #강의자료는 .없이 출력됨(버
labels_dummy = torch.tensor(labels_np).long()

# 결과 확인
print(outputs_dummy.data)
print(labels_dummy.data)

nllloss = nn.NLLLoss()
loss = nllloss(outputs_dummy, labels_dummy) # -(1 + 5 + 9 + 10)/4 = -6.25
print(loss.item())

<출력>
tensor([[ 1.,  2.,  3.],
        [ 4.,  5.,  6.],
        [ 7.,  8.,  9.],
        [10., 11., 12.]])
tensor([0, 1, 2, 0])
-6.25
```

- 모델 클래스 측에 LogSoftmax 함수 포함

```
class Net(nn.Module):
  def __init__(self, n_input, n_output):
    super().__init__()
    self.l1 = nn.Linear(n_input, n_output)

    # logsoftmax 함수 정의
    self.logsoftmax = nn.LogSoftmax(dim=1) # 열 방향으로 연산)


  def forward(self, x):
    x1 = self.l1(x)
    x2 = self.logsoftmax(x1)
    return x2
```

- 모델 클래스 측에 소프트맥스 함수 포함

```
class Net(nn.Module):
  def __init__(self, n_input, n_output):
    super().__init__()
    self.l1 = nn.Linear(n_input, n_output)
    self.softmax = nn.Softmax(dim=1)

  def forward(self, x):
    x1 = self.l1(x)
    x2 = self.softmax(x1)
    return x2

lr = 0.01
net = Net(n_input, n_output)
```

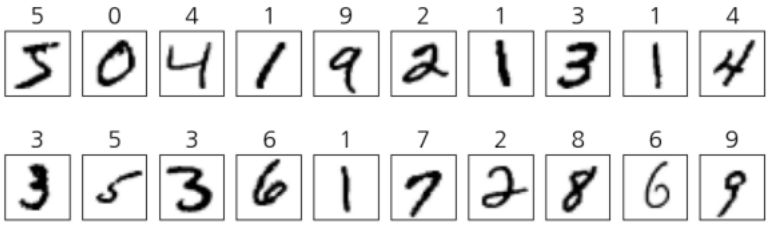
```
criterion = nn.NLLLoss()
optimizer = optim.SGD(net.parameters(), lr=lr)
num_epochs = 10000
history = np.zeros((0,5))
```

```
for epoch in range(num_epochs):


    optimizer.zero_grad()
    outputs = net(inputs)

    # 여기서 로그 함수를 적용함
    outputs2 = torch.log(outputs)

    loss = criterion(outputs2, labels)
    loss.backward()
    optimizer.step()
    predicted = torch.max(outputs, 1)[1]

    train_loss = loss.item()
    train_acc = (predicted == labels).sum()  / len(labels)

    outputs_test = net(inputs_test)

    # 여기서 로그 함수를 적용함
    outputs2_test = torch.log(outputs_test)

    loss_test = criterion(outputs2_test, labels_test)
    predicted_test = torch.max(outputs_test, 1)[1]

    val_loss =  loss_test.item()
    val_acc =  (predicted_test == labels_test).sum() / len(labels_test)

    if epoch % 10 == 0:
        print (f'Epoch [{epoch}/{num_epochs}], loss: {train_loss:.5f} acc: {train_a
        item = np.array([epoch , train_loss, train_acc, val_loss, val_acc])
        history = np.vstack((history, item))
```

8단원 ‘MNIST를 활용한 숫자 인식’ (송주훈, 정민섭)

- 개요

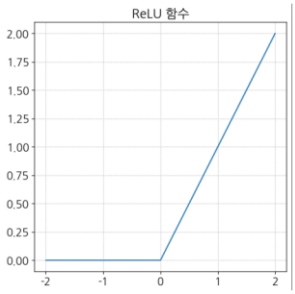
MNIST Dataset을 전처리하여 다중분류하는 것이 목표이며 MNIST Dataset이란 간단한 컴퓨터 비전 데이터셋이며 손으로 쓰여진 숫자 이미지로 구성되어 있다. 6만 개의 training set과 1만 개의 test set으로 구성되어 있으며 0~1까지의 값을 갖는 고정 크기(28x28) 이미지로 크기가 표준화 되었다.



- 은닉층: ReLU 함수

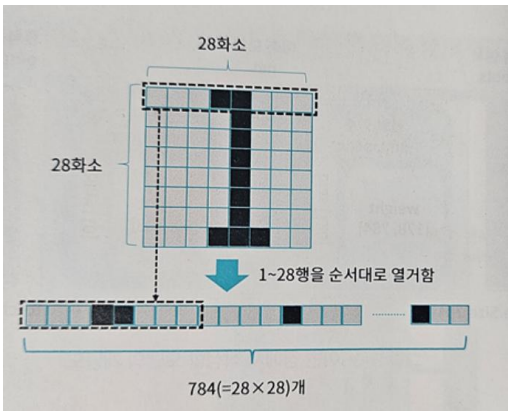
활성화 함수 중 하나이며 층이 깊은 신경망을 유의미하게 해주는 비선형 함수이다. 이번 모델부터 은닉층을 포함하게 된다. 다시 말해 2개 층 이상의 선형 함수로 이뤄진 모델을 사용하게 된다.

층을 깊게 만들면 범용성이 높아지고 높은 정확도의 모델이 만들어지게 되며 그리고 이 깊이를 유효하게 하기 위해 선형 함수들 사이에 활성화 함수를 사용한다.



- 데이터 전처리

학습 데이터를 모델에 입력하기 전 필요에 따라 가공하는 과정이다.



해당 모델에서는 [1,28,28]의 데이터를 784개의 요소를 1차원 배열로 바꿔줘야하며 전처리에서 transform을 이용했다.

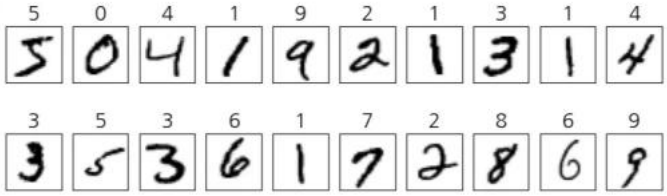
1. 데이터 로드 및 확인

```
# 파이토치 관련 라이브러리
import torch
from torch import nn, optim
```

```
import torch.nn.functional as F
from torchviz import make_dot
from torchinfo import summary
from tqdm.notebook import tqdm

import torchvision.transforms as transforms
import torchvision.datasets as datasets
```

```
# 라이브러리 임포트
import torchvision.datasets as datasets

# 다운로드받을 디렉터리명
data_root = './data'

train_set0 = datasets.MNIST(
    # 원본 데이터를 다운로드받을 디렉터리 지정
    root = data_root,
    # 훈련 데이터인지 또는 검증 데이터인지
    train = True,
    # 원본 데이터가 없는 경우, 다운로드를 실행하는지 여부
    download = True)

# 첫번째 요소 가져오기
image, label = train_set0[0]

# 정답 데이터와 함께 처음 20개 데이터를 이미지로 출력

plt.figure(figsize=(10, 3))
for i in range(20):
    ax = plt.subplot(2, 10, i + 1)

    # image와 label 취득
    image, label = train_set0[i]

    # 이미지 출력

    plt.imshow(image, cmap='gray_r')
    ax.set_title(f'{label}')
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

plt.show()
```

▼ plot한 결과



2. Transforms를 활용한 데이터 전처리

```
# 라이브러리 임포트
import torchvision.transforms as transforms

transform = transforms.Compose([
    # (1) 데이터를 텐서로 변환
    transforms.ToTensor(), # 0 ~ 255 → 0 ~ 1

    # (2) 데이터 정규화
    transforms.Normalize(0.5, 0.5), # -1 ~ 1

    # (3) 1d 텐서로 변환
    transforms.Lambda(lambda x: x.view(-1)), # 1d tensor
])

# 데이터 입수를 위한 Dataset 함수
```

```
# 훈련용 데이터셋 정의
train_set = datasets.MNIST(
    root = data_root,
    train = True,
    download = True,
    transform = transform)

# 검증용 데이터셋 정의
test_set = datasets.MNIST(
    root = data_root,
    train = False,
    download = True,
    transform = transform)
```

- 미니 배치 학습법

배치 학습법: 원래 학습 데이터 전체 건수로 한 번에 경사를 계산

미니 배치 학습법: 학습 데이터를 배치 수마다 가중치 업데이트

1. 데이터로더를 활용한 미니 배치 데이터 생성

```
# 라이브러리 임포트
from torch.utils.data import DataLoader

# 미니 배치 사이즈 지정
batch_size = 500

# 훈련용 데이터로더
# 훈련용이므로, 셔플을 적용함
train_loader = DataLoader(
    dataset = train_set,
    batch_size = batch_size,
    shuffle = True)
```

```
# 검증용 데이터로더
# 검증시에는 셔플을 필요로하지 않음
test_loader = DataLoader(
    dataset = test_set,
    batch_size = batch_size,
    shuffle = False)
```

2. 데이터 형태 확인

```
print(len(train_loader))

# 데이터로더로부터 가장 처음 한 세트를 가져옴
for images, labels in train_loader:
    break

print(images.shape)
print(labels.shape)
```

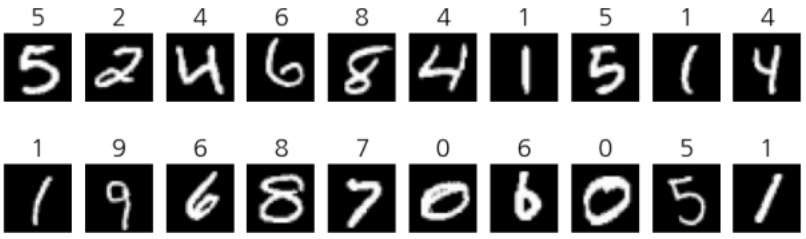
```
120
torch.Size([500, 784])
torch.Size([500])
```

- 여기서 나온 120은 total_data(60000) / batch_size(500)이다.

3. 추출한 데이터 셋 확인

```
# 이미지 출력
plt.figure(figsize = (10, 3))

for i in range(20):
  ax = plt.subplot(2, 10, i+1)
```

```
    image2 = images[i].view(28,28).data.numpy() # image reshape
    ax.imshow(image2, cmap = 'gray')
    ax.set_title(str(labels[i].item()))
    ax.axis('off')

plt.show()
```



- 모델 정의

1. 입력, 출력 차원 수 정의

```
# 입력 차원수
n_input = image.shape[0]

# 출력 차원수
# 분류 클래스 수는 10
n_output = len(set(list(labels.data.numpy())))

# 은닉층의 노드 수
n_hidden = 128

# 결과 확인
print(f'n_input: {n_input}  n_hidden: {n_hidden} n_output: {n_output}')
```

```
n_input: 784  n_hidden: 128 n_output: 10
```

2. 모델 클래스 정의

```
# 모델 정의
# 784입력 10출력 1은닉층의 신경망 모델

class Net(nn.Module):
    def __init__(self, n_input, n_output, n_hidden):
        super().__init__()

        # 은닉층 정의(은닉층 노드 수 : n_hidden)
        self.l1 = nn.Linear(n_input, n_hidden)

        # 출력층 정의
        self.l2 = nn.Linear(n_hidden, n_output)

        # ReLU 함수 정의
        self.relu = nn.ReLU(inplace=True)

    def forward(self, x):
        x1 = self.l1(x)
        x2 = self.relu(x1)
        x3 = self.l2(x2)
        return x3
```

3. 모델 인스턴스 생성 및 summary

```
net = Net(n_input, n_outputs, n_hidden)
net = net.to(device) # cpu에서 처리하겠다는 의미

print(net)
summary(net, (784, ), device = 'cpu')
```

- 모델 학습

1. 하이퍼 파라미터 설정

```
# 학습률
lr = 0.01

# 모델 초기화
net = Net(n_input, n_output, n_hidden).to(device)

# 손실 함수 : 교차 엔트로피 함수
criterion = nn.CrossEntropyLoss()

# 최적화 함수: 경사 하강법
optimizer = optim.SGD(net.parameters(), lr=lr)

# 반복 횟수
# num_epochs = 100
num_epochs = 10

# 데이터로더에서 취득한 데이터를 GPU로 보냄 <- 이건 GPU 사용할 때 사용하면 됨
```

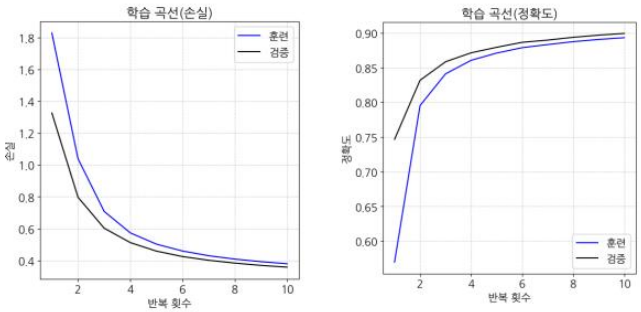
```
inputs = images.to(device)
labels = labels.to(device)
```

2. 경사 하강법 반복 처리

```
# 평가 결과 기록
history = np.zeros((0,5))

# tqdm 라이브러리 임포트
from tqdm.notebook import tqdm

# 반복 계산 메인 루프
for epoch in range(num_epochs):
  train_acc, train_loss = 0, 0
  val_acc, val_loss = 0, 0
  n_train, n_test = 0, 0

  # 훈련 페이즈
  for inputs, labels in tqdm(train_loader):
    n_train += len(labels)

    # GPU로 전송
    inputs = inputs.to(device)
    labels = labels.to(device)

    # 경사 초기화
    optimizer.zero_grad()

    # 예측 계산
    outputs = net(inputs)

    # 손실 계산
    loss = criterion(outputs, labels)
```

```
    # 경사 계산
    loss.backward()

    # 파라미터 수정
    optimizer.step()

    # 예측 라벨 산출
    predicted = torch.max(outputs, 1)[1]

    # 손실과 정확도 계산
    train_loss += loss.item()
    train_acc += (predicted == labels).sum().item()

# 예측 페이즈
for inputs_test, labels_test in test_loader:
    n_test += len(labels_test)

    inputs_test = inputs_test.to(device)
    labels_test = labels_test.to(device)


    # 예측 계산
    outputs_test = net(inputs_test)

    # 손실 계산
    loss_test = criterion(outputs_test, labels_test)

    # 예측 라벨 산출
    predicted_test = torch.max(outputs_test, 1)[1]

    # 손실과 정확도 계산
    val_loss +=  loss_test.item()
    val_acc +=  (predicted_test == labels_test).sum().item()

# 평가 결과 산출, 기록
train_acc = train_acc / n_train
val_acc = val_acc / n_test
train_loss = train_loss * batch_size / n_train
val_loss = val_loss * batch_size / n_test
print (f'Epoch [{epoch+1}/{num_epochs}], loss: {train_loss:.5f} acc: {train_acc:
item = np.array([epoch+1 , train_loss, train_acc, val_loss, val_acc])
history = np.vstack((history, item))
```

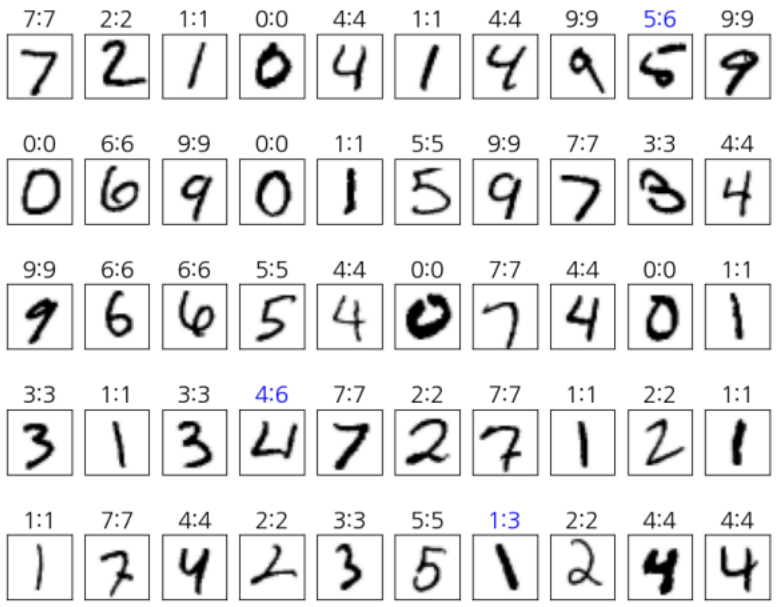
- 결과 확인(학습 곡선 출력)



	- 이미지 출력 확인

```
# 데이터로더에서 처음 한 세트 가져오기
for images, labels in test_loader:
    break

# 예측 결과 가져오기
inputs = images.to(device)
labels = labels.to(device)
outputs = net(inputs)
predicted = torch.max(outputs, 1)[1]
```

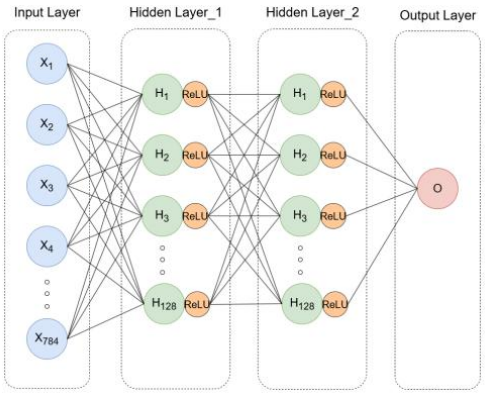
```
# 처음 50건의 이미지에 대해 "정답:예측"으로 출력

plt.figure(figsize=(10, 8))
for i in range(50):
  ax = plt.subplot(5, 10, i + 1)

  # 넘파이 배열로 변환
  image = images[i]
  label = labels[i]
  pred = predicted[i]
  if (pred == label):
    c = 'k'
  else:
    c = 'b'

  # 이미지의 범위를 [0, 1] 로 되돌림
  image2 = (image + 1)/ 2

  # 이미지 출력
  plt.imshow(image2.reshape(28, 28),cmap='gray_r')
  ax.set_title(f'{label}:{pred}', c=c)
  ax.get_xaxis().set_visible(False)
  ax.get_yaxis().set_visible(False)
plt.show()
```



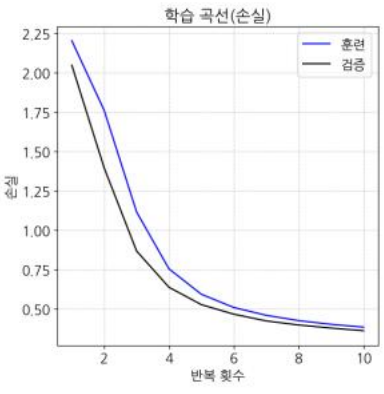
- 은닉층 추가하기

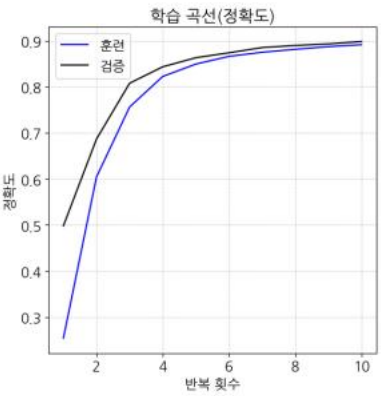
```
# 모델 정의
# 784입력 10출력을 갖는 2개의 은닉층을 포함한 신경망

class Net2(nn.Module):
    def __init__(self, n_input, n_output, n_hidden):
        super().__init__()

        # 첫번째 은닉층 정의(은닉층 노드 수: n_hidden)
        self.l1 = nn.Linear(n_input, n_hidden)
```

```
        # 두번째 은닉층 정의(은닉층 노드 수: n_hidden)
        self.l2 = nn.Linear(n_hidden, n_hidden)

        # 출력층 정의
        self.l3 = nn.Linear(n_hidden, n_output)

        # ReLU 함수 정의
        self.relu = nn.ReLU(inplace=True)

    def forward(self, x):
        x1 = self.l1(x)
        x2 = self.relu(x1)
        x3 = self.l2(x2)
        x4 = self.relu(x3)
        x5 = self.l3(x4)
        return x5
```



- 학습 과정은 위와 동일한 과정이므로 생략
- 학습 곡선(Loss & Acc)





- 은닉층을 1개 사용했을때 최종 Acc : 0.89930
- 은닉층을 2개 사용했을때 최종 Acc : 0.89880 → 은닉층을 두 개로 늘린 효과는 미미하다.

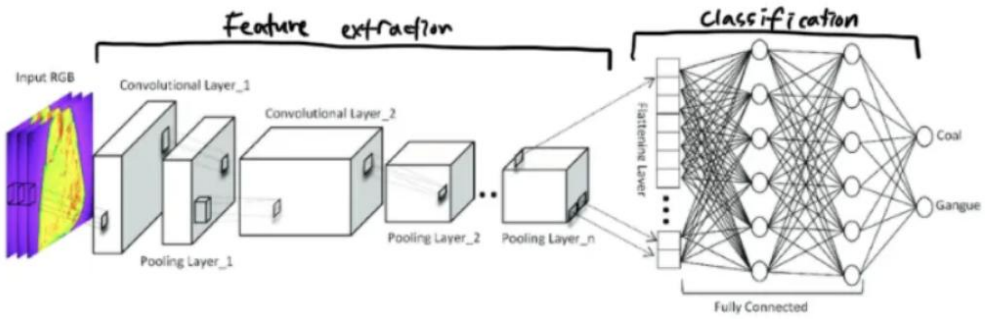
9단원 'CNN을 활용한 이미지 인식' (정찬원, 임소정)

- 개요

  - Convolution neural network의 약자이며 이미지 분석에 자주 사용되는 주변 정보에 영향을 받는 데이터에 적합한 딥러닝 기법.
  - 인간의 시신경 구조를 모방한 기술.
  - 특징맵을 생성하는 필터까지도 학습이 가능해 CV 분야에서 성능이 우수함.
  - 자율주행자동차, 얼굴인식과 같은 객체 인식 등 필요한 분야에 많이 사용되고 있음
  - 이미지의 공간 정보를 유지한 채 학습을 하게 하는 모델로써 1D로 변환하는 것이 아닌 2D 그대로 작업함.

- 구조

  - Feature extraction/learning(이미지 특징 추출)
    - Convolution layer: 입력 데이터에 필터를 적용 후 활성화 함수를 반영하는 필수 요소.
    - Pooling layer
  - Classification(클래스 분류)
    - 이미지 분류를 위해 Fully Connected Layer가 추가되며 이미지의 특징을 추출하는 부분과 이미지를 분류하는 부분 사이에 이미지 형태의 데이터를 배열 형태로 만드는 Flatten layer가 위치함.



- 장점

- 공간적 관계 유지

입력 이미지의 작은 영역에 대해서만 필터 적용. 즉, 인접 픽셀 간의 관계 유지하고 학습할 수 있으며, 이미지의 공간 패턴을 인식하는 데 유리함.

- 파라미터 수 감소

동일한 필터를 이미지 전체에 걸쳐 적용하기 때문에 파라미터 수가 크게 줄어듦. 이로 인해 모델이 더 효율적이고 메모리가 절약됨.

- 변환 불변성

필터가 이미지의 모든 위치에 동등하게 적용되기 때문에, 객체의 위치 변화에 대해 비교적 강인함. 이미지 내 객체의 위치가 약간 변해도 그 특징을 잘 인식할 수 있게 함.

- convolution

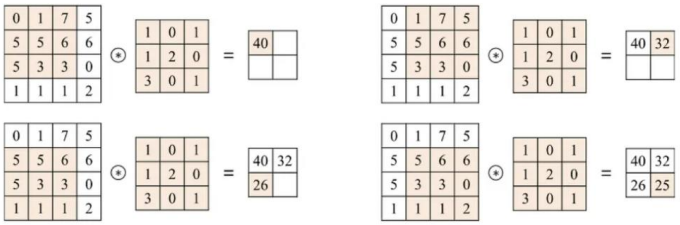
이미지 데이터는 높이x너비x채널의 3차원 텐서로 표현될 수 있으며 이미지 색상이 RGB 코드로 표현되었다면, 채널의 크기는 3이 되고 각각의 채널은 R,G,B값이 저장됨.

- **filter 적용**

하나의 합성곱 계층에는 입력되는 이미지 채널 개수만큼 필터가 존재하며, 각 채널에 할당된 필터를 적용함으로써 합성곱 계층의 출력 이미지가 생성됨.

보통 3x3 혹은 5x5와 같은 정방향 배열이 filter가 되며 이를 합성곱 처리에서는 '커널'이라고 부름.

example 1) 4x4x1 텐서 형태의 입력 이미지에 대해 3x3 필터(각 filter 안에 weight)를 적용하여 2x2x1 텐서 형태 이미지 생성.



하나의 채널에 대한 Convolution(합성곱) 계층의 동작

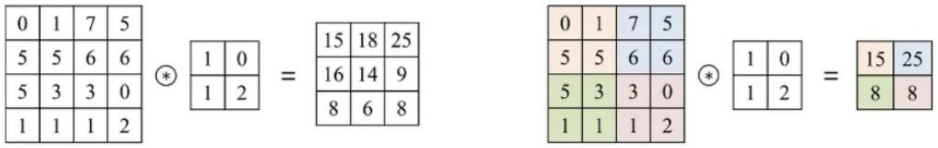
- **stride**

이미지에 대해 필터를 적용할 때 필터의 이동량을 의미.

stride 값이 1이면 filter가 한번에 1 fixel씩 이동하며 stride 값이 2이면 2 fixel씩 이동하며 feature map을 만들 때 stride 값이 커질수록 feature map이 작아짐.

보통 CNN 구현할 때 합성곱 계층의 stride는 주로 1로 설정됨.

(왼쪽) stride = 1인 경우, (오른쪽)stride = 2인 경우

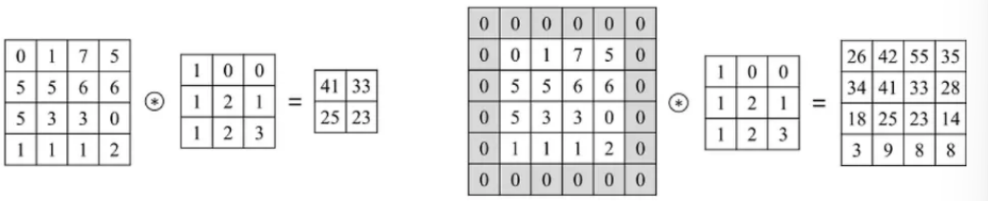


- **padding**

입력 이미지에 대해 합성곱을 수행하면, 출력 이미지의 크기는 입력 이미지의 크기보다 작아지게 되는데 합성곱 계층을 거치면서 이미지의 크기는 점점 작아지게 되고, 이미지의 가장자리에 위치한 픽셀들의 정보는 사라지게 되는 문제점 발생.

이를 해결하고자 사용되는 것이 padding이며, 입력 이미지의 가장 자리에 특정 값으로 설정된 픽셀들을 추가하여 입력 이미지와 출력 이미지의 크기를 같거나 비슷하게 만드는 역할을 수행.

CNN에서는 주로 이미지 가장자리에 0의 값을 갖는 zero-padding 이용됨.



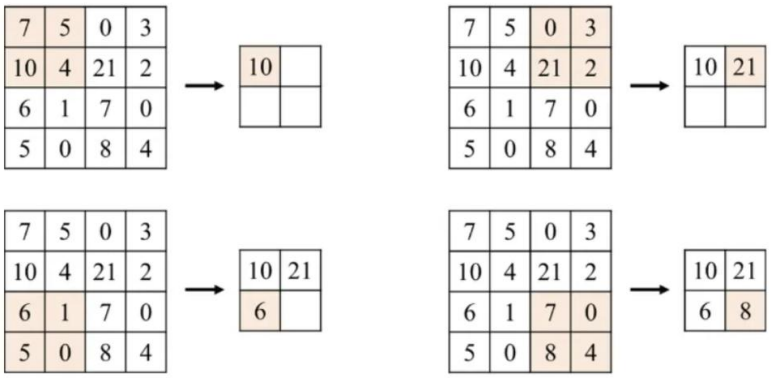
(왼쪽) Padding을 안한 경우, (오른쪽) Zero-padding인 경우

- pooling layer

  - max-pooling

    - 이미지의 크기를 계속 유지한 채 fully connected layer로 가게 되면 연산량이 기하급수적으로 늘기 때문에 적당히 크기도 줄이며 특정 feature를 강조할 때 pooling layer에서 그 역할을 하게 됨.

    - pooling의 종류는 max pooling, average pooling, min pooling이 있으며 CNN에서는 주로 **max pooling**이 사용되며 노이즈가 감소하고 속도가 빨라지며 영상의 분별력의 좋아짐.



Max-pooling이 적용된 Pooling layer

- fully connected layer

  - 종류

    - Flatten Layer: 데이터 타입을 Fully Connected 네트워크 형태로 변경하여 입력 데이터의 shape 변경만 수행.

    - Softmax Layer: Classification 수행.

```
# 라이브러리 임포트

%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
from IPython.display import display

# 폰트 관련 용도
import matplotlib.font_manager as fm

# 나눔 고딕 폰트의 경로 명시
path = '/usr/share/fonts/truetype/nanum/NanumGothic.ttf'
font_name = fm.FontProperties(fname=path, size=10).get_name()
```

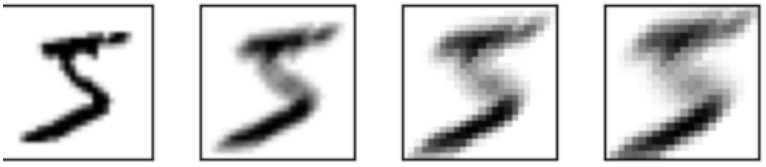
```
# 파이토치 관련 라이브러리

import torch
import torch.nn as nn
import torch.optim as optim
from torchinfo import summary
from torchviz import make_dot
import torchvision.datasets as datasets
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
```

```
data_root = './data'

# 샘플 손글씨 숫자 데이터 가져오기
transform = transforms.Compose([
    transforms.ToTensor(),
])

train_set = datasets.MNIST(
    root = data_root,  train = True,
    download = True, transform = transform)

image, label = train_set[0]
image = image.view(1,1,28,28)
```

```
# 대각선상에만 가중치를 갖는 특수한 합성곱 함수를 만듦
conv1 = nn.Conv2d(1, 1, 3)

# bias를 0으로
nn.init.constant_(conv1.bias, 0.0)

# weight를 특수한 값으로
w1_np = np.array([[0,0,1],[0,1,0],[1,0,0]])
w1 = torch.tensor(w1_np).float()
w1 = w1.view(1,1,3,3)
conv1.weight.data = w1
```

```
# 손글씨 숫자에 3번 합성곱 처리를 함
image, label = train_set[0]
image = image.view(1,1,28,28)
w1 = conv1(image)
w2 = conv1(w1)
w3 = conv1(w2)
images = [image, w1, w2, w3]
```

```
# 결과 화면 출력

plt.figure(figsize=(5, 1))
for i in range(4):
    size = 28 - i*2
    ax = plt.subplot(1, 4, i+1)
    img = images[i].data.numpy()
    plt.imshow(img.reshape(size, size),cmap='gray_r')
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```



- nn.Conv2d와 nn.MaxPool2d

```
# CNN 모델 전반 부분, 레이어 함수 정의

conv1 = nn.Conv2d(3, 32, 3)      #(입력
relu = nn.ReLU(inplace=True)
conv2 = nn.Conv2d(32, 32, 3)
maxpool = nn.MaxPool2d((2,2))       #(풀
```

```
# conv1 확인
print(conv1)

# conv1 내부 변수의 shape 확인
print(conv1.weight.shape)
print(conv1.bias.shape)

# conv2 내부 변수의 shape 확인
print(conv2.weight.shape)
print(conv2.bias.shape)
```

```
# conv1의 weight[0]는 0번째 출력 채널의 가중치
w = conv1.weight[0]

# weight[0]의 shape과 값 확인
print(w.shape)
print(w.data.numpy())
```

```
# 더미로 입력과 같은 사이즈를 갖는 텐서를 생성
inputs = torch.randn(100, 3, 32, 32)
print(inputs.shape)
```

```
# CNN 전반부 처리 시뮬레이션

x1 = conv1(inputs)
x2 = relu(x1)
x3 = conv2(x2)
x4 = relu(x3)
x5 = maxpool(x4)        #작은 사각형 2x2
```

- nn.Sequential: pytorch에서 'container'라고 불리우는 클래스 중 하나.

```
# 함수로 정의
features = nn.Sequential(
    conv1,
    relu,
    conv2,
    relu,
    maxpool
)
```

```
# 동작 테스트
outputs = features(inputs)

# 결과 확인
print(outputs.shape)
```

```
torch.Size([100, 32, 14, 14])
```

- nn.Flatten: 3계 텐서를 선형 함수에서 사용할 수 있도록 1계 텐서로 형태 변환

```
# 함수 정의
flatten = nn.Flatten()

# 동작 테스트
outputs2 = flatten(outputs)

# 결과 확인: 4계->2계(한 건의 데이터에 대해서는 3계->1계)
print(outputs.shape)
print(outputs2.shape)
```

```
torch.Size([100, 32, 14, 14])
torch.Size([100, 6272])
```

- 공통 함수 (eval_loss, fit, eval_history, show_images_labels)

eval_loss: 손실 계산 함수.

fit(학습): 반복 계산 부분을 함수로 정의하여 한 번에 처리하며 함수 호출 시 8개의 인수(net, optimizer, criterion, num_epoch, train_loader, test_loader, device, history) 필요함.

eval_history: history 앞부분과 마지막 부분을 print해서 학습 결과의 개요를 표시하며 학습 곡선을 손실, 정확도 두 가지로 출력.

show_images_labels(예측 결과 표시): 사전에 학습이 끝난 모델이 올바르게 예측하고 있는지, 원본 데이터의 이미지를 출력함과 동시에 수행.

(함수 및 데이터셋 코드는 이전 코드와 동일하기에 생략)

- CNN 모델 정의

```
class CNN(nn.Module):
  def __init__(self, n_output, n_hidden):
    super().__init__()
    self.conv1 = nn.Conv2d(3, 32, 3)
    self.conv2 = nn.Conv2d(32, 32, 3)
    self.relu = nn.ReLU(inplace=True)
    self.maxpool = nn.MaxPool2d((2,2))
    self.flatten = nn.Flatten()
    self.l1 = nn.Linear(6272, n_hidden)
    self.l2 = nn.Linear(n_hidden, n_output)

    self.features = nn.Sequential(
        self.conv1,
        self.relu,
        self.conv2,
        self.relu,
        self.maxpool)

    self.classifier = nn.Sequential(
       self.l1,
       self.relu,
       self.l2)

  def forward(self, x):
    x1 = self.features(x)
    x2 = self.flatten(x1)
    x3 = self.classifier(x2)
    return x3
```

- 모델 인스턴스 생성

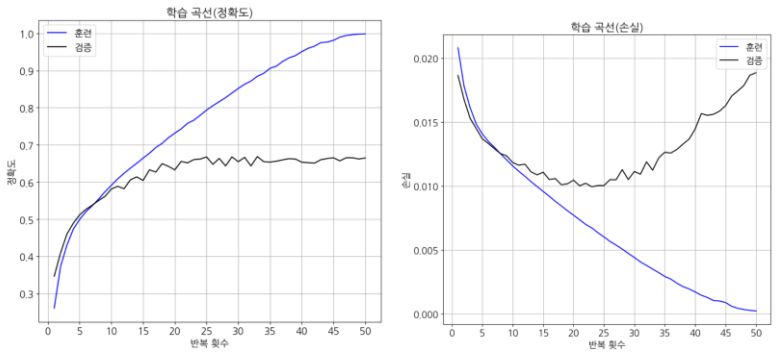
```
# 모델 인스턴스 생성
net = CNN(n_output, n_hidden).to(device)

# 손실 함수 : 교차 엔트로피 함수
criterion = nn.CrossEntropyLoss()

# 학습률
lr = 0.01

# 최적화 함수: 경사 하강법
optimizer = torch.optim.SGD(net.parameters(), lr=lr)
```

- 학습

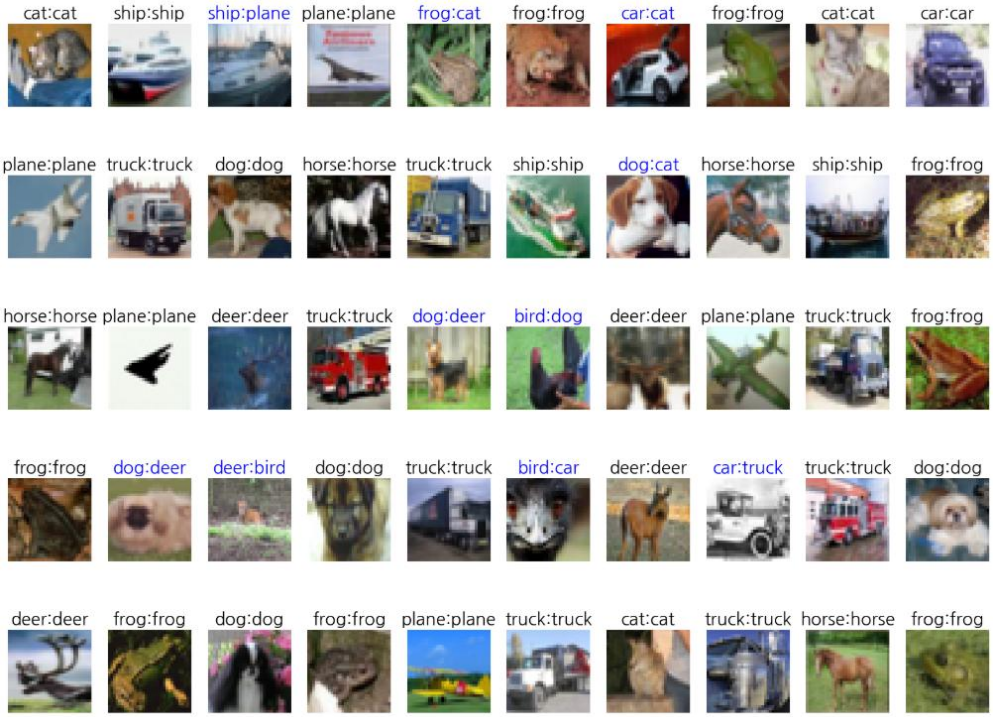
```
# 난수 초기화
torch_seed()

# 모델 인스턴스 생성
net = CNN(n_output, n_hidden).to(device)

# 손실 함수 : 교차 엔트로피 함수
criterion = nn.CrossEntropyLoss()

# 학습률
lr = 0.01

# 최적화 함수: 경사 하강법
optimizer = optim.SGD(net.parameters(), lr=lr)

# 반복 횟수
num_epochs = 50

# 평가 결과 기록
history2 = np.zeros((0,5))

# 학습
history2 = fit(net, optimizer, criterion, num_epochs, train_loader2, test_loader2, device, histor
```

```
# 평가

evaluate_history(history2)
```

초기상태 : 손실 : 0.01866 정확도 : 0.34680
최종상태 : 손실 : 0.01887 정확도 : 0.66490



```
# 처음 50개 데이터 표시

show_images_labels(test_loader2, classes, net, device)
```



기존 강의자료 내 Net 모델로 선형으로 학습을 진행하는 것에 비해 CNN 모델로 이미지 classification을 진행하는 것이 정확도가 더 높다는 것을 확인할 수 있다.

활동평가	7단원~9단원까지 교재 내용 요약 및 발표를 진행하면서, 정규 수업 때 놓쳤던 내용들이나 중요한 내용들을 다시 한번 되돌아보는 시간을 가질 수 있었다. 다중분류와 MNIST, CNN을 활용해 딥러닝에 대해 한발 더 가까워질 수 있는 시간이 되었다.
과제	스터디 자료 업로드 및 단원별 복기, CNN 개념 복기하기

향후 계획	CNN 관련 논문 선정 후 리뷰하기
첨부 자료	**활동 사진**