# Pytorch Tutorial for Deep Learning Lovers

## Introduction

파이토치 기반 scientific computing 패키지

- 장점: 대화형으로 디버깅하여 편의 제공, 디버깅과 시각화 측면에서 사용하기 쉬움, 다양한 그래픽에 대한 지원, 페이스북의 조직적 지원, 다양한 API
- 단점: 다른 대안들에 비해 완전히 발달하지 않음, 참고자료 및 리소스가 제한됨

이 튜톨리얼에서는 파이토치를 사용하여 신경망 기본 개념을 설명함.

```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import matplotlib.pyplot as plt
# 입력 데이터 파일은 "../input/" 디렉토리에서 사용 가능
# 실행 버튼 또는 shift+Enter 키를 눌러 실행하면 입력 디렉토리 파일이 나열됨

import os
print(os.listdir("../input"))

# 현재 디렉토리에 기록한 결과는 모두 출력으로 저장됨
```

## Basics of Pytorch

### Matrices

- 파이토치에서 행렬은 tensors라고 불림
- 3*3 matrix는 3x3 tensor
- numpy를 통해 행렬 예시 살펴보기
  - np.numpy() 메소드로 numpy array 생성
  - Type(): array의 type(이 예시에서는 numpy)
  - np.shape(): array의 모양(row x column)

```
# import numpy library
import numpy as np

# numpy array
array = [[1,2,3],[4,5,6]]
first_array = np.array(array) # 2x3 array
print("Array Type: {}".format(type(first_array))) # type
print("Array Shape: {}".format(np.shape(first_array))) # shape
print(first_array)
```

```
Array Type: <class 'numpy.ndarray'>
Array Shape: (2, 3)
[[1 2 3]
 [4 5 6]]
```

```
# import pytorch library
import torch

# 위 코드에서는 numpy를 사용하여 2x3 행렬을 만들었다면, 이번에는 파이토치로 행렬을 생성
# 파이토치에서는 행렬을 tensor라고 하며, torch.Tensor() 메소드로 이를 생성함
# array의 타입은 tensor, shape은 numpy와 동일(row x column)

# pytorch array
tensor = torch.Tensor(array)
print("Array Type: {}".format(tensor.type)) # type
print("Array Shape: {}".format(tensor.shape)) # shape
print(tensor)
```

```
Array Type: <built-in method type of Tensor object at 0x780233326840>
Array Shape: torch.Size([2, 3])
tensor([[1., 2., 3.],
        [4., 5., 6.]])
```

- Allocation(선언)은 코딩 기술 중 가장 중요한 것 중 하나. 이를 파이토치로 만들어보자
- numpy와 tensor를 비교
  - np.ones()와 torch.one()
  - np.random.rand()와 torch.rand()

```
# numpy ones: 1로 초기화
print("Numpy {}₩n".format(np.ones((2,3))))
```

```
# pytorch ones: 1값을 가지는 tensor 생성
print(torch.ones((2,3)))

# np.ones()와 torch.one()는 같은 기능을 한다
```

```
Numpy [[1. 1. 1.]
 [1. 1. 1.]]

tensor([[1., 1., 1.],
        [1., 1., 1.]])
```

```
# numpy random: 0부터 1까지의 난수 반환
print("Numpy {}₩n".format(np.random.rand(2,3)))

# pytorch random: 0과 1 사이의 실수로 난수 생성, 텐서 생성
print(torch.rand(2,3))

# np.random.rand()와 torch.rand()는 같은 기능을 한
```

```
Numpy [[0.0390377  0.0885951  0.64588773]
 [0.98580024 0.9082953  0.58961536]]

tensor([[0.7315, 0.3779, 0.8039],
        [0.8809, 0.6728, 0.6371]])
```

- tensor와 numpy array 사이의 변환
  - torch.from_numpy(): numpy에서 tensor로
  - numpy(): tensor에서 numpy로

```
# 랜덤으로 numpy array 생성
array = np.random.rand(2,2)
print("{} {}₩n".format(type(array),array))

# numpy에서 tensor로
from_numpy_to_tensor = torch.from_numpy(array)
print("{}₩n".format(from_numpy_to_tensor))

# tensor에서 numpy로
tensor = from_numpy_to_tensor
from_tensor_to_numpy = tensor.numpy()
print("{} {}₩n".format(type(from_tensor_to_numpy),from_tensor_to_numpy))
```

```
<class 'numpy.ndarray'> [[0.29232851 0.68203044]
 [0.07457622 0.31602574]]

tensor([[0.2923, 0.6820],
        [0.0746, 0.3160]], dtype=torch.float64)

<class 'numpy.ndarray'> [[0.29232851 0.68203044]
 [0.07457622 0.31602574]]
```

## Basic Math with Pytorch

- Resize(차원 조작): view()
- a와 b가 tensor라고 했을 때
  - 덧셈: torch.add(a,b) = a+b
  - 뺄셈: a.sub(b) = a-b
  - 요소별 곱셈: torch.mul(a,b) = a*b
  - 요소별 나눗셈: torch.div(a,b) = a/b
- 평균: a.mean()
- 표준편차(std): a.std()

```
# tensor 생성
tensor = torch.ones(3,3)
print("₩n",tensor)

# 차원 조작
print("{}{}₩n".format(tensor.view(9).shape,tensor.view(9)))
# view()를 통해 모양을 바꾸면서 메모리를 공유하게 되므로,
# 원래의 텐서나 새로운 텐서 중 어느 하나를 변경하면 다른 하나도 영향을 받게 됨

# 덧셈
print("Addition: {}₩n".format(torch.add(tensor,tensor)))

# Subtraction
print("Subtraction: {}₩n".format(tensor.sub(tensor)))

# Element wise multiplication
print("Element wise multiplication: {}₩n".format(torch.mul(tensor,tensor)))

# Element wise division
print("Element wise division: {}₩n".format(torch.div(tensor,tensor)))

# Mean
```

```
tensor = torch.Tensor([1,2,3,4,5])
print("Mean: {}".format(tensor.mean()))

# Standart deviation (std)
print("std: {}".format(tensor.std()))
```

```
 tensor([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]])
torch.Size([9])tensor([1., 1., 1., 1., 1., 1., 1., 1., 1.])

Addition: tensor([[2., 2., 2.],
        [2., 2., 2.],
        [2., 2., 2.]])

Subtraction: tensor([[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.]])

Element wise multiplication: tensor([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]])

Element wise division: tensor([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]])

Mean: 3.0
std: 1.5811388492584229
```

## Variables

- 그래디언트(경사) 누적
- 신경망에서는 그래디언트가 계산되는 역전파가 있으므로, 우선 그래디언트를 처리해야 함.
- 변수와 텐서 사이의 차이는 변수 누적 그래디언트
- 변수로 수학 연산 가능
- 역전파를 하기 위해 변수 필요

```
# import variable from pytorch library
from torch.autograd import Variable

# define variable
# Variable()은 파이토치의 이전 버전에서 사용되었으며, 계산 그래프를 구성할 때 사용됨.
# 모델의 파라미터에 대해 자동으로 미분을 수행하기 위한 특별한 텐서로서, 역전파를 지원함.
# 최신 버전의 파이토치에서는 torch.Tensor 자체가 자동 미분을 지원하므로, 일반적으로 Variable을 사용할 필요는 없다.

# requires_grad=True 매개변수는 자동 미분을 사용하기 위해 필요한 것
# requires_grad를 True로 설정하면 파이토치는 이 텐서에서 이루어지는 모든 연산을 추적하고, 나중에 역전파를 통해 그래디언트를 계산할 수 있게 됨.
var = Variable(torch.ones(3), requires_grad = True)
var
```

```
tensor([1., 1., 1.], requires_grad=True)
```

- y=x^2이라는 방정식이 있다고 가정
- x = [2,4]라는 변수 정의
- 계산 후 우리는 y = [4,16] (y = x^2)를 찾을 수 있음
- 이 방정식을 비용함수 o로 재정의하면 o = (1/2)sum(y) = (1/2)sum(x^2)
- o를 x에 대해 미분한 결과, 이는 x와 같으므로 그라디언트(경사)는 [2,4]가 됨(x를 각 요소에 대해 미분한 결과)

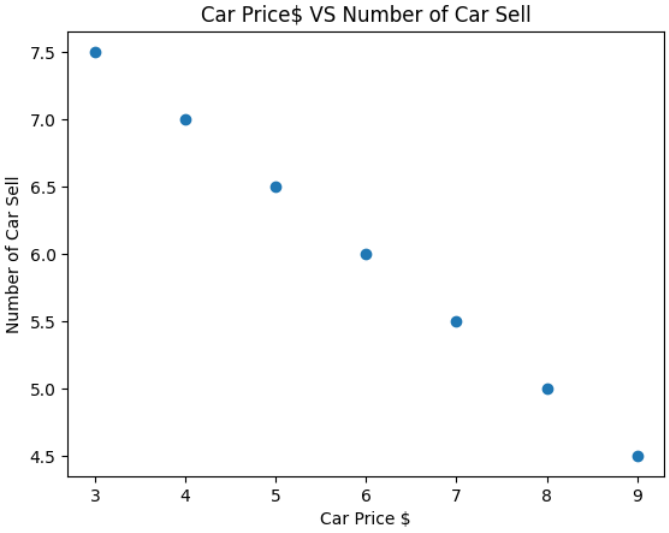
```
# 역전파를 생성해보자
# y = x^2 방정식이 있다고 가정
array = [2,4]
tensor = torch.Tensor(array)
x = Variable(tensor, requires_grad = True)
y = x**2
print(" y =  ",y)

# 비용함수 o를 재정의: o = 1/2*sum(y)
o = (1/2)*sum(y)
print(" o =  ",o)

# 역전파
o.backward() # 경사를 계산
# backword() 메소드는 각 매개변수에 대한 그래디언트를 계산함. 이는 손실함수를 각 매개변수로 미분한 값
# 해당 매개변수를 조절하여 손실을 최소화하는 방향으로 업데이트하는 데 사용됨.

# 변수들은 경사를 누적함. 이 경우 변수 x는 하나밖에 없음
# 변수 x에는 경사가 있어야 함
# x.grad로 경사 살펴보기
print("gradients: ",x.grad) # x가 2일 때 경사는 4
```

```
 y =   tensor([ 4., 16.], grad_fn=<PowBackward0>)
 o =   tensor(10., grad_fn=<MulBackward0>)
gradients:  tensor([2., 4.])
```

## Linear Regression

- y = Ax + b에서
  - A는 곡선의 기울기
  - B는 편향(y절편)
- 예를 들어 어떤 자동차 회사에서, 차값이 낮으면 차를 더 많이 팔고 차값이 높으면 차를 덜 판다고 하자. 우리는 이 사실에 대한 데이터셋을 가지고 있다.
- 알고 싶은 것: 자동차 가격이 100일 때 자동차 판매 대수

```
# 이전 판매 기록으로 자동차 회사에 대한 데이터셋을 확보
# 자동차 판매 가격 정의
car_prices_array = [3,4,5,6,7,8,9]
car_price_np = np.array(car_prices_array,dtype=np.float32) # np.array() 메소드
car_price_np = car_price_np.reshape(-1,1) # np.reshape() 메소드
car_price_tensor = Variable(torch.from_numpy(car_price_np)) # torch.from_numpy(), Variable() 메소드

# 자동차 판매 대수 정의(판매 가격 정의 코드의 반복)
number_of_car_sell_array = [ 7.5, 7, 6.5, 6.0, 5.5, 5.0, 4.5]
number_of_car_sell_np = np.array(number_of_car_sell_array,dtype=np.float32)
number_of_car_sell_np = number_of_car_sell_np.reshape(-1,1)
number_of_car_sell_tensor = Variable(torch.from_numpy(number_of_car_sell_np))

# 데이터 시각화
import matplotlib.pyplot as plt
plt.scatter(car_prices_array,number_of_car_sell_array)
plt.xlabel("Car Price $")
plt.ylabel("Number of Car Sell")
plt.title("Car Price$ VS Number of Car Sell")
plt.show() # 산점도 생성
```

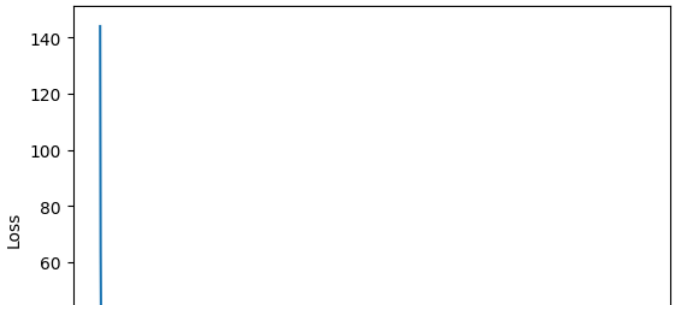



- 위 산점도는 우리가 모은 데이터를 시각화한 것
- 우리가 궁금한 것은 자동차 판매 가격이 100$일 때의 자동차 판매 대수
- 이 문제를 해결하기 위해 선형 회귀(linear regression)을 사용할 것
- 선형 회귀의 단계
  - 선형회귀 class 생성
  - LinearRegression class로부터 모델 정의
  - MSE(평균 제곱 오차): Mean squeared error
  - 최적화(확률적 경사 하강법)
  - 역전파 수행
  - 예측

```
# 파이토치를 이용한 선형 회귀

# 라이브러리 임포트
import torch
from torch.autograd import Variable
import torch.nn as nn
import warnings
warnings.filterwarnings("ignore")

# 클래스 생성
class LinearRegression(nn.Module): # nn.Module 클래스를 상속받는 LinearRegression을 새로 정의

    def __init__(self,input_size,output_size):
    # 입력 차원과 출력 차원을 인자로 받음
    # 선형 회귀 모델에서 입력과 가중치를 곱하고 편향을 더하는 연산을 수행
    # __init__ 메소드: 클래스의 생성자 메소드로, 클래스가 초기화될 때 호출
```

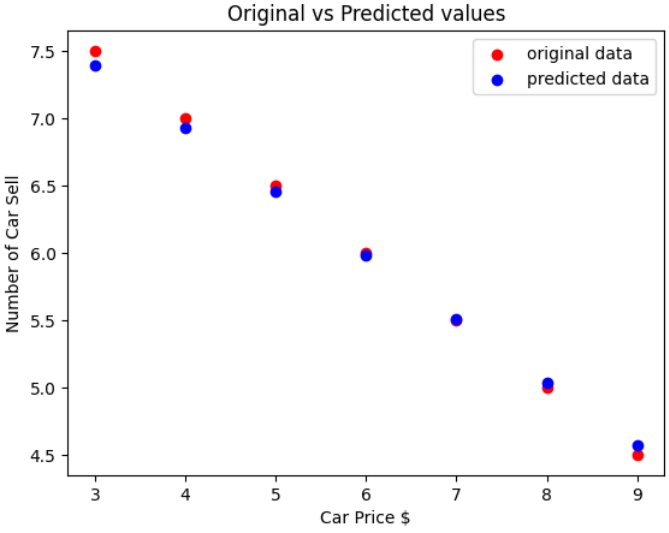
```
        # super function: nn.Module 상속받음, nn.Module의 모든 think에 접근 가능
        super(LinearRegression,self).__init__()
        # 부모 클래스인 nn.Module 호출
        # 부모 클래스인 nn.Module의 생성자를 호출하여 초기화

        # Linear function.
        self.linear = nn.Linear(input_dim,output_dim)
        # nn.Linear: 선형 변환 수행 레이어, 입력 차원 input size와 출력 차원 output_size를 인자로 받음
        # 선형 회귀 모델에서 입력과 가중치(w)를 곱하고 편향(b)을 더하는 연산 수행

    def forward(self,x): # nn.Module 클래스에서 상속받은 메소드로, 모델의 순전파 연산을 저장
        return self.linear(x)
        # self.linear(x): 앞서 정의한 선형 레이어 nn.Linear에 입력 x를 전달하여 선형 변환을 수행 및 결과 반환
        # 선형 레이어는 가중치와 편향이 내제된 연산을 수행하므로 이를 통해 입력 데이터ㅔ 대한 모델의 예측값이 생성됨

# def __init__ 메소드는 모델의 구조를 정의하고 초기화할 때 호출
# def forward 메소드는 주어진 입력 데이터를 사용하여 실제로 모델의 순전파를 수행

# 모델 정의
input_dim = 1 # 입력 차원
output_dim = 1 # 출력 차원
model = LinearRegression(input_dim,output_dim) # 입력 차원과 출력 차원 모두 1

# 평균제곱오차 MSE
mse = nn.MSELoss()
# nn.MSELoss(): 파이토치에서 제공하는 평균 제곱 오차 손실함수
# 평균 제곱 오차 손실함수는 회귀 문제에서 모델의 예측과 실제 타겟 값 간의 평균 오차 제곱을 계산함으로써 오차를 측정

# 최적화 (오차를 최소화하는 매개변수 찾기)
learning_rate = 0.02  # 학습률 α(각 업데이트에서 모델의 매개변수를 얼마나 변화시킬 것인지), 최적의 매개변수에 다다르는 데 얼마나 빨리 걸리는지 결정
optimizer = torch.optim.SGD(model.parameters(),lr = learning_rate)
# torch.optim.SGD 인스턴스 생성: SGD(확률적 경사 하강법) 최적화 알고리즘을 구현한 파이토치의 클래스
# model.parameters(): 최적화할 모델의 매개변수 전달, 모델 내의 가주치와 편향 등을 포함
# lr은 학습률이라 불리는 하이퍼파라미터
# 옵티마이저: 손실 함수에서 계산된 그래디언트를 사용하여 모델의 매개변수 업데이트
# SGD는 각 매개변수를 현재 그래디언트와 학습률을 사용하여 업데이트 함.

# 이렇게 설정된 SGD 옵티마이저는 이후에 모델을 학습할 때 사용되며, 역전파 및 가중치 업데이트를 수행
# 최적화 알고리즘과 학습률의 선택은 모델의 학습 성능에 영향을 미칠 중요한 하이퍼파라미터

# 모델 훈련
loss_list = [] # 훈련 중에 발생한 손실값을 저장하기 위한 리스트 초기화
iteration_number = 1001 # 훈련 1001번 반복
for iteration in range(iteration_number):

    # optimization, 옵티마이저 초기화
    # 매 반복마다 새로운 그래디언트를 계산하기 전에 기존의 그래디언트를 0으로 초기화
    # 파이토치의 zero_grad() 메소드 사용
    optimizer.zero_grad()


    # Forward to get output, 순전파 수행
    # 모델의 입력 데이터를 전달하여 순전파를 수행하고, 모델은 입력에 대한 예측값을 반환
    results = model(car_price_tensor)

    # Calculate Loss, 손실 계산
    # 예측값과 실제 타겟값을 사용하여 평균 제곱 오차를 계산
    loss = mse(results, number_of_car_sell_tensor)

    # backward propagation, 역전파 수행
    # 손실에 대한 역전파를 수행하여 그래디언트를 계산
    loss.backward()

    # Updating parameters, 옵티마이저를 사용하여 매개변수 업데이트
    # 경사하강법을 통해 손실을 최소화하기 위해 수행
    optimizer.step()

    # store loss, 매 반복마다 계산된 손실값 저장
    loss_list.append(loss.data)

    # print loss, 일정한 간격으로 현재 epoch와 손실 출력
    # epoch: 전체 훈련 데이터셋이 모델에 대해 한 번 순전파와 역전파를 거치는 주기
    # 각 에포크는 훈련 데이터셋을 한 번 사용하는 것을 의미하며, epoch가 증가할수록 모델은 더 많은 훈련 데이터를 보고 학습하게 됨
    if(iteration % 50 == 0):
        print('epoch {}, loss {}'.format(iteration, loss.data))

plt.plot(range(iteration_number),loss_list)
plt.xlabel("Number of Iterations")
plt.ylabel("Loss")
plt.show()
```

```
epoch 0, loss 143.95468139648438
epoch 50, loss 6.129683971405029
epoch 100, loss 4.14208459854126
epoch 150, loss 2.7989799976348877
epoch 200, loss 1.891387701034546
epoch 250, loss 1.2780905961990356
epoch 300, loss 0.863659679889679
epoch 350, loss 0.5836107134819031
epoch 400, loss 0.3943707048892975
epoch 450, loss 0.266492635011673
epoch 500, loss 0.18008029460906982
epoch 550, loss 0.12168773263692856
epoch 600, loss 0.08222945779561996
epoch 650, loss 0.055565740913152695
epoch 700, loss 0.03754819557070732
epoch 750, loss 0.025372754782438278
epoch 800, loss 0.0171455517411232
epoch 850, loss 0.011585956439375877
epoch 900, loss 0.007829082198441029
epoch 950, loss 0.005290493369102478
epoch 1000, loss 0.0035749587696045637
```




- 1001회 반복
- 그래프를 통해 손실은 거의 0이라는 점을 확인할 수 있음
- 훈련은 1000 epoch 동안 진행되었으며 마지막 에포크에서의 손실값은 거의 0에 가까움
- 이렇게 훈련된 모델을 이용하여 자동차 가격 예측을 수행

```
# 자동차 가격 예측

# 훈련된 선형 회귀 모델인 model을 사용, 입력 데이터인 car_price_tensor를 모델에 전달하여 예측값 획득
# .data: 텐서의 데이터 부분을 나타내며, 여기서는 모델의 예측값을 나타냄
# 모델을 통해 얻은 예측값을 Numpy 배열로 변환하여 predicted에 할당
predicted = model(car_price_tensor).data.numpy()

plt.scatter(car_prices_array, number_of_car_sell_array,label = "original data",color ="red")
plt.scatter(car_prices_array,predicted,label = "predicted data",color ="blue")

# 자동차 가격이 $10일 때 자동차 판매 대수 예측(특정한 값 예)
# predicted_10 = model(torch.from_numpy(np.array([10]))).data.numpy()
# plt.scatter(10,predicted_10.data,label = "car price 10$",color ="green")
plt.legend()
plt.xlabel("Car Price $")
plt.ylabel("Number of Car Sell")
plt.title("Original vs Predicted values")
plt.show()
```




- 빨간 점: 실제 데이터의 타겟 값
- 파란 점: 모델을 통해 예측한 값
- 빨간 점과 파란 점의 위치가 거의 유사함, 모델을 통한 예측이 잘 되었음을 알 수 있음

## Logistic Regression

- 선형 회귀는 분류에 효과적이지는 않음
- 분류를 할 때는 로지스틱 회귀를 사용
- 선형 회귀 + 로지스틱 함수(softmax) = 로지스틱 회귀
- 로지스틱 회귀의 단계
  - 라이브러리 임포트
  - 데이터셋 준비
    - MNIST 데이터셋 사용
    - 28*28 크기의 이미지와 0부터 9까지 10개의 label(분류 타겟) 존재
    - 데이터를 나누기 위해 사이킷런의 train_test_split 사용
    - 80%가 train 데이터셋, 20%가 test 데ㅣ터셋
    - feature와 target 텐서 만들기. 다음 부분부터 이 텐서들로부터 변수를 생성(경사의 누적에 대한 변수 정의)
    - batch_size: 전체 데이터셋을 몇 개의 그룹으로 나눌 때, 그룹의 크기를 의미.예를 들어 데이터셋에 1000개의 샘플이 있고 이를 100개의 샘플을 포함하는 10개의 그룹으로 나눈다고 할 때 batch_size = 100.
    - epoch: 1 epoch은 모든 샘플들을 한 번씩 훈련하는 것을 의미
    - 우리의 예시에서 33600개의 샘플을 훈련시키고 batch_size를 100으로 설정. 또한 epoch은 29로 설정함(정확도는 epoch가 29일 때 거의 가장 높은 값을 달성). 데이터는 29번 훈련되며, 총 반복 횟수는 9744회(100사이즈의 그룹 336개, epoch가 29이므로 336*29=9744).
    - TensorDataset(): tensor를 wrapping하는 데이터셋. 각 샘플은 첫 번째 차원을 따라 tensor를 인덱싱하여 검색함.
    - DataLoader(): 데이터셋과 샘플을 결합하고, 데이터셋에 대한 멀티프로세스 반복을 제공
    - 데이터셋의 이미지 중 하나를 시각화
  - 로지스틱 회귀 모델 생성
    - 선형 회귀와 동일
    - 모델 안에 로지스틱 함수 포함
    - 파이토치에서 로지스틱 함수는 손실 함수 안에 존재
  - 모델 예시
    - 입력 차원: 28*28 (이미지 사이즈)
    - 출력 차원: 10 (0부터 9까지의 label)
  - 손실 예시
    - 교차 엔트로피 오차
  - 모델 훈련 및 예측
- 결과적으로 그래프를 통해 확인할 수 있듯이, 손실은 감소하는 반면 정확도(85%)는 증가하고 있으며 모델은 훈련 중임.

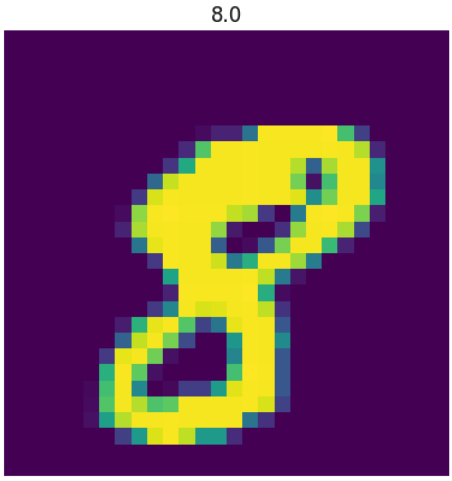
```
# Import Libraries
import torch
import torch.nn as nn
from torch.autograd import Variable
from torch.utils.data import DataLoader
import pandas as pd
from sklearn.model_selection import train_test_split
```

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

```
# 데이터셋 준비
# 데이터 로드
train = pd.read_csv(r"/content/drive/MyDrive/data/digit_train.csv",dtype = np.float32)

# features(pixels) 와 labels(numbers from 0 to 9)로 데이터 분리
targets_numpy = train.label.values
features_numpy = train.loc[:,train.columns != "label"].values/255 # 정규화

# 훈련 세트, 테스트 세트 분리 (80:20)
features_train, features_test, targets_train, targets_test = train_test_split(features_numpy,
                                                                               targets_numpy,
                                                                               test_size = 0.2,
                                                                               random_state = 42)

# 훈련 세트에 대한 feature, target tensor 생성
# 경사 누적을 위한 변수(variable) 필요 >> tensor 생성 후 변수 생성
targetsTrain = torch.from_numpy(targets_train).type(torch.LongTensor) # data type is long

# 테스트 세트에 대한 feature, target tensor 생성
featuresTest = torch.from_numpy(features_test)
targetsTest = torch.from_numpy(targets_test).type(torch.LongTensor) # data type is long

# batch_size, epoch and iteration
batch_size = 100
n_iters = 10000
num_epochs = n_iters / (len(features_train) / batch_size)
```

```
num_epochs = int(num_epochs)

# Pytorch train and test sets
# torch.utils.data: PyTorch에서 데이터셋과 데이터로더를 관리하기 위한 유틸리티 함수와 클래스를 제공하는 서브 모듈
# featuresTrain과 targetsTrain은 훈련 데이터의 입력 특성과 타겟(레이블)을 나타내는 PyTorch 텐서
# TensorDataset 클래스: 입력된 텐서들을 묶어서 하나의 데이터셋으로 만들어줌
# 각각의 텐서는 동일한 인덱스에 있는 데이터들끼리 연결
train = torch.utils.data.TensorDataset(featuresTrain,targetsTrain)
test = torch.utils.data.TensorDataset(featuresTest,targetsTest)

# data loader: 반복문에서 간편하게 미니배치 단위로 데이터를 추출

# PyTorch의 DataLoader를 사용하여 훈련 데이터셋과 테스트 데이터셋을 미니배치로 나누고 셔플링함
# train은 훈련 데이터셋을 나타내는 TensorDataset 객체, 'test는 테스트 데이터셋을 나타내는 TensorDataset 객체
# batch_size는 미니배치의 크기를 지정하는 매개변수로, 훈련 데이터를 미니배치로 나눔
# shuffle은 데이터를 에포크마다 섞을지 여부를 나타내는 매개변수로 여기서는 False로 설정하여 순서대로 데이터를 로드함
train_loader = DataLoader(train, batch_size = batch_size, shuffle = False)
test_loader = DataLoader(test, batch_size = batch_size, shuffle = False)

# 데이터셋에 있는 이미지를 시각화

# features_numpy는 이미지 데이터를 담고 있는 NumPy 배열이라고 가정, 여기서는 10번째 이미지를 시각화
# reshape(28,28)를 사용하여 이미지를 28x28 크기로 변형하여 시각화
#.imshow(): 이미지를 표시하는 함수
plt.imshow(features_numpy[10].reshape(28,28))

plt.axis("off") # 이미지 주변에 있는 축을 비활성화하여 이미지 주변에 불필요한 눈금 및 축을 제거
plt.title(str(targets_numpy[10])) # 이미지의 레이블을 타이틀로 추가 (targets_numpy는 이미지에 대한 레이블이 담긴 NumPy 배열이라고 가정)
plt.savefig('graph.png') # 시각화한 이미지를 'graph.png'라는 파일로 저장
plt.show()
```




```
# 로지스틱 회귀 모델 생성
class LogisticRegressionModel(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(LogisticRegressionModel, self).__init__()
        # 선형 부분
        self.linear = nn.Linear(input_dim, output_dim)
        # 로지스틱 회귀이므로 로지스틱 함수가 있어야 함
        # 그러나 Pytorch의 로지스틱 함수는 손실 함수에 있음
        # 함수 넣는 것을 잊은 것이 아니라, 단지 다음 단계에 있을 뿐

    def forward(self, x): # 선형 회귀와 동일
        out = self.linear(x)
        return out

# 모델 클래스 초기화
input_dim = 28*28 # size of image px*px
output_dim = 10  # labels 0,1,2,3,4,5,6,7,8,9

# 로지스틱 회귀 모델 생성
model = LogisticRegressionModel(input_dim, output_dim)

# 교차 엔트로피 오차
error = nn.CrossEntropyLoss()

# 확률적 경사 하강법의 최적화 알고리즘
learning_rate = 0.001 # 학습률

# PyTorch에서 SGD 최적화 알고리즘을 사용하여 모델의 매개변수를 업데이트하는데 사용되는 옵티마이저를 설정
# torch.optim.SGD: 확률적 경사 하강법(SGD)을 구현한 PyTorch의 SGD 옵티마이저
# model.parameters(): 모델의 학습 가능한 모든 매개변수를 반환하는 함수
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)


# 모델 훈련
count = 0
loss_list = []
iteration_list = []
```

```
# 전체 데이터셋에 대해 여러 번(epoch 수 만큼) 반복하며 미니배치를 가져오기 위한 루프
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):

        # 변수 정의
        # 불러온 이미지 데이터를 펼쳐서 1차원 벡터로 만들고, Variable을 사용하여 PyTorch에서 처리할 수 있는 형태로 변환
        train = Variable(images.view(-1, 28*28))
        labels = Variable(labels)

        # 그래디언트 초기화
        # PyTorch에서는 기본적으로 이전 반복의 그래디언트를 저장하기 때문에, 매 반복마다 그래디언트를 초기화해야 함
        optimizer.zero_grad()

        # 순전파
        # 모델에 입력 데이터를 전달하여 예측값을 얻음
        outputs = model(train)

        # 손실 계산
        # 모델의 예측값과 실제 레이블 간의 손실을 계산
        loss = error(outputs, labels)

        # 역전파
        # 손실을 사용하여 역전파를 수행하고, 각 매개변수에 대한 그래디언트를 계산
        loss.backward()

        # 계산된 그래디언트를 사용하여 매개변수 업데이트
        optimizer.step()

        count += 1 # 반복 횟수 증가

        # 예측
        # 매 일정한 간격(예: 50번의 반복마다) 정확도를 계산하고, 일정한 간격(예: 500번의 반복마다) 손실과 정확도를 출력
        if count % 50 == 0:
            # 정확도 계산
            correct = 0
            total = 0
            # Predict test dataset

            # test_loader를 통해 테스트 데이터셋에 대한 미니배치를 가져옴
            # 각 미니배치를 모델에 전달하고, 모델의 출력에서 예측값을 얻음
            # 정확한 예측 수를 총 정답 수에 더함
            for images, labels in test_loader:
                test = Variable(images.view(-1, 28*28))

                # 순전파
                outputs = model(test)

                # 모델의 출력에서 최댓값을 찾아 예측 클래스를 가져옴
                # 최댓값을 찾아서 예측 클래스를 가져오는 것은 주로 분류 문제에서 모델의 출력 중에서 가장 높은 확률을 가지는 클래스를 선택하는 방법
                # 모델의 출력은 보통 소프트맥스(softmax) 함수를 통과하게 되면 각 클래스에 속할 확률값으로 나타남
                predicted = torch.max(outputs.data, 1)[1]

                # 전체 레이블 수를 더하고, 정확한 예측 수를 계산하여 correct에 더함

                total += len(labels)

                # 전체 correct predictions
                correct += (predicted == labels).sum()

            # 정확도 계산 및 저장
            # 정확도를 전체 데이터셋에 대한 정답률로 계산
            # 계산된 정확도를 리스트에 추가
            # 손실과 반복 횟수도 리스트에 추가
            accuracy = 100 * correct / float(total)

            # 손실과 반복 저장
            loss_list.append(loss.data)
            iteration_list.append(count)

        if count % 500 == 0:
            # Print Loss
            print('Iteration: {}  Loss: {}  Accuracy: {}%'.format(count, loss.data, accuracy))
```

```
Iteration: 500  Loss: 0.5645173788070679  Accuracy: 85.96428680419922%
Iteration: 1000  Loss: 0.6604678630828857  Accuracy: 86.11904907226562%
Iteration: 1500  Loss: 0.4804125130176544  Accuracy: 86.1547622680664%
Iteration: 2000  Loss: 0.5825194120407104  Accuracy: 86.25%
Iteration: 2500  Loss: 0.5339609384536743  Accuracy: 86.35713958740234%
Iteration: 3000  Loss: 0.3981034755706787  Accuracy: 86.55952453613281%
Iteration: 3500  Loss: 0.5301312208175659  Accuracy: 86.63095092773438%
Iteration: 4000  Loss: 0.38779616355895996  Accuracy: 86.80952453613281%
Iteration: 4500  Loss: 0.7063114643096924  Accuracy: 86.89286041259766%
Iteration: 5000  Loss: 0.5221177935600281  Accuracy: 86.94047546386719%
Iteration: 5500  Loss: 0.5085040926933289  Accuracy: 87.0%
Iteration: 6000  Loss: 0.6531243920326233  Accuracy: 87.11904907226562%
Iteration: 6500  Loss: 0.44170719385147095  Accuracy: 87.16666412353516%
Iteration: 7000  Loss: 0.518631100654602  Accuracy: 87.21428680419922%
Iteration: 7500  Loss: 0.45419037342071533  Accuracy: 87.22618865966797%
Iteration: 8000  Loss: 0.5872360467910767  Accuracy: 87.39286041259766%
Iteration: 8500  Loss: 0.4137895703315735  Accuracy: 87.45237731933594%
```

```
Iteration: 9000  Loss: 0.5009725093841553  Accuracy: 87.52381134033203%
Iteration: 9500  Loss: 0.41010287404060364  Accuracy: 87.54762268066406%
```

```
# 시각화
plt.plot(iteration_list,loss_list)
plt.xlabel("Number of iteration")
plt.ylabel("Loss")
plt.title("Logistic Regression: Loss vs Number of iteration")
plt.show()
```


## 🌀 Artificial Neural Network(ANN)

- 로지스틱 회귀는 분류에 효과적이지만, 복잡도가 증가하면 모델의 정확도가 떨어진다
- 따라서 모델의 복잡도를 증가시켜야 한다
- 모델의 복잡도를 높이기 위해, 은닉층으로 비선형 함수를 더 추가해야 한다
- 우리가 ANN에서 기대하는 것은 복잡성이 증가할 때, 모델이 잘 적응할 수 있도록 은닉층을 더 넣어주는 것이며, 그 결과 정확도가 증가할 것이다.

- ANN의 단계
  - 라이브러리 임포트
  - 데이터셋 준비
    - 데이터셋, batch size, epoch, iteration number 이전과 동일
  - ANN 모델 생성
    - 3개의 은닉층 추가
    - ReLU, Tanh, ELU 활성화 함수 사용
  - 모델 클래스의 인스턴스화
    - 입력 차원: 28*28
    - 출력 차원: 10
    - 은닉층의 차원: 150(정해진 것이 아님, 다른 차원을 이용하고 결과를 비교해보는 것도 좋음)
    - 모델 생성
  - 손실의 인스턴스화
    - 교차 엔트로피 오차
    - softmax(logistic function) 보유
  - 최적화 알고리즘 인스턴스화
    - SGD Optimizer
  - 모델 훈련 및 예측

- 그래프를 통해 알 수 있듯, 손실이 감소하는 동안 정확도 증가
- 은닉 계층 덕분에 모형이 더 잘 학습되었고 정확도(거의 95%)가 로지스틱 회귀 모형의 정확도보다 더 향상되었음

```
# Import Libraries
import torch
import torch.nn as nn
from torch.autograd import Variable
```

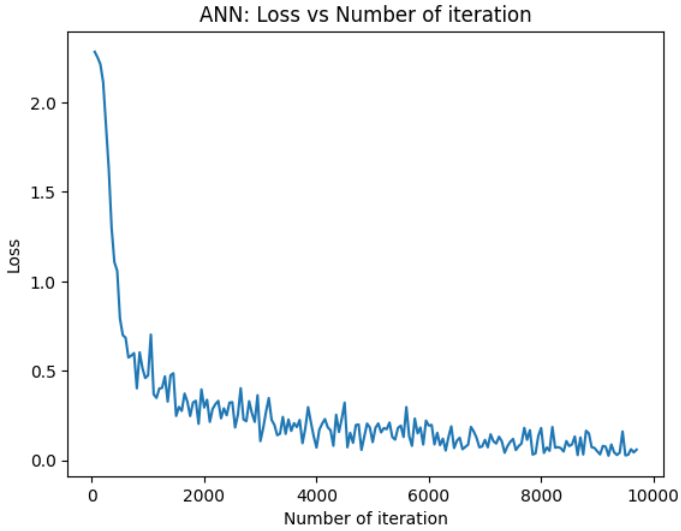
```
# Create ANN Model
class ANNModel(nn.Module): # nn.Module 클래스를 상속하는 ANNModel 클래스를 정의

    def __init__(self, input_dim, hidden_dim, output_dim): # __init__ 메서드에서 모델의 초기 설정을 수행
```

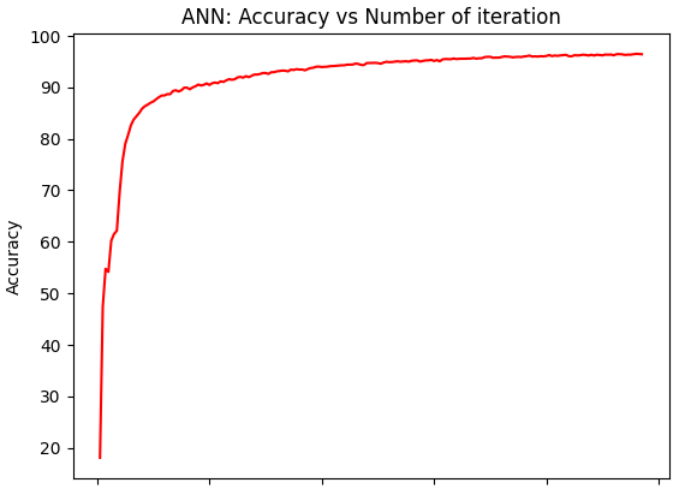
```
        # super(ANNModel, self).__init__()은 부모 클래스인 nn.Module의 초기화 메서드를 호출
        super(ANNModel, self).__init__()

        # 신경망의 각 층 정의
        # 선형 변환 층 (nn.Linear)과 활성화 함수 (nn.ReLU, nn.Tanh, nn.ELU)을 갖는 각 층을 정의
        # 입력 차원(input_dim), 은닉층 차원(hidden_dim), 출력 차원(output_dim)을 인자로 받아 초기화

        # 은닉층1
        # Linear function 1: 784 --> 150
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        # Non-linearity 1
        self.relu1 = nn.ReLU()

        # 은닉층2
        # Linear function 2: 150 --> 150
        self.fc2 = nn.Linear(hidden_dim, hidden_dim)
        # Non-linearity 2
        self.tanh2 = nn.Tanh()

        # 은닉층3
        # Linear function 3: 150 --> 150
        self.fc3 = nn.Linear(hidden_dim, hidden_dim)
        # Non-linearity 3
        self.elu3 = nn.ELU()

        # 출력층
        # Linear function 4 (readout): 150 --> 10
        self.fc4 = nn.Linear(hidden_dim, output_dim)

    # 순전파 매소드 정의, 각 층의 연산을 순서대로 진행하고 최종적으로 출력을 반환
    def forward(self, x):

        # 첫 번째 은닉층에 입력 데이터 x를 전달하여 선형 변환을 수행하고, 그 결과를 활성화 함수 ReLU에 적용
        # Linear function 1
        out = self.fc1(x)
        # Non-linearity 1
        out = self.relu1(out)

        # 두 번째 은닉층에 이전 층의 출력 out을 전달하여 선형 변환을 수행하고, 그 결과를 활성화 함수 Tanh에 적용
        # Linear function 2
        out = self.fc2(out)
        # Non-linearity 2
        out = self.tanh2(out)

        # 세 번째 은닉층에 이전 층의 출력 out을 전달하여 선형 변환을 수행하고, 그 결과를 활성화 함수 ELU에 적용
        # Linear function 2
        out = self.fc3(out)
        # Non-linearity 2
        out = self.elu3(out)

        # 출력층에 이전 층의 출력 out을 전달하여 선형 변환을 수행
        # Linear function 4 (readout)
        out = self.fc4(out)

        return out # 출력 반환

# ANN을 인스턴스화
input_dim = 28*28 # 입력 차원은 28x28 이미지를 1차원으로 펼친 크기인 input_dim
hidden_dim = 150 # 직접 조정해야 하며, 150으로 정한 특별한 이유가 없음
output_dim = 10

# ANN 생성
model = ANNModel(input_dim, hidden_dim, output_dim)

# 손실 함수로 Cross Entropy Loss 설정
# nn.CrossEntropyLoss()는 Cross Entropy Loss를 나타내는 PyTorch의 손실 함수
# 모델의 출력과 정답 레이블 사이의 손실을 계산
error = nn.CrossEntropyLoss()

# 최적화 알고리즘으로 SGD Optimizer 설정
# torch.optim.SGD는 확률적 경사 하강법(Stochastic Gradient Descent) 옵티마이저를 나타냄
# model.parameters()를 통해 모델의 학습 가능한 매개변수들을 전달하고, lr=learning_rate로 학습률을 설정
# 이 옵티마이저는 모델을 훈련할 때 사용될 역전파 알고리즘을 구현
learning_rate = 0.02
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)


# ANN model training
count = 0
loss_list = []
iteration_list = []
accuracy_list = []

for epoch in range(num_epochs): # 전체 데이터셋에 대해 몇 번 반복할지를 결정하는 에폭 루프
    # 훈련 데이터셋에서 미니배치를 가져오는 루프, enumerate 함수를 사용하여 미니배치의 인덱스(i)와 데이터(images, labels)를 순회
    for i, (images, labels) in enumerate(train_loader):

        # 데이터 전처리
        # 미니배치의 이미지 데이터를 1차원으로 평탄화하고 변수에 저장
        train = Variable(images.view(-1, 28*28))
        labels = Variable(labels)
```

```
        # 역전파 전에 저장된 그래디언트를 초기화
        optimizer.zero_grad()

        # 순전파, 모델에 입력 데이터를 전달하여 순전파를 수행하고 출력을 얻음
        outputs = model(train)

        # 모델의 출력과 정답 레이블을 사용하여 Cross Entropy Loss(손실)를 계산
        loss = error(outputs, labels)

        # 경사 계산, 역전파 수행
        loss.backward()

        # 옵티마이저를 사용하여 모델의 매개변수를 업데이트
        optimizer.step()

        # 반복 횟수 업데이트
        count += 1

        # 일정한 간격으로 정확도를 계산하고 기록
        if count % 50 == 0:
            # 정확도 계산
            correct = 0
            total = 0
            # 테스트 데이터에 예측, 정확도 계산
            for images, labels in test_loader:

                test = Variable(images.view(-1, 28*28))

                # 순전파
                outputs = model(test)

                # 모델의 출력에서 최댓값을 찾아 예측 클래스를 가져옴
                # 최댓값을 찾아서 예측 클래스를 가져오는 것은 주로 분류 문제에서 모델의 출력 중에서 가장 높은 확률을 가지는 클래스를 선택하는 방법
                # 모델의 출력은 보통 소프트맥스(softmax) 함수를 통과하게 되면 각 클래스에 속할 확률값으로 나타남
                predicted = torch.max(outputs.data, 1)[1]

                # 전체 레이블 수를 더하고, 정확한 예측 수를 계산하여 correct에 더함

                total += len(labels)

                # 전체 correct predictions
                correct += (predicted == labels).sum()

            accuracy = 100 * correct / float(total)

            # 손실, 반복 저장
            # 정확도 및 손실 기록
            loss_list.append(loss.data)
            iteration_list.append(count)
            accuracy_list.append(accuracy)
        if count % 500 == 0: # 특정 반복 횟수에 도달할 때마다 현재 손실과 정확도를 출력하여 훈련 진행 상황을 모니터링
            # Print Loss
            print('Iteration: {}  Loss: {}  Accuracy: {} %'.format(count, loss.data, accuracy))
```

```
Iteration: 500  Loss: 0.061433807015419006  Accuracy: 96.54762268066406 %
Iteration: 1000  Loss: 0.03397037088871002  Accuracy: 96.5952377319336 %
Iteration: 1500  Loss: 0.022273726761341095  Accuracy: 96.71428680419922 %
Iteration: 2000  Loss: 0.03900328278541565  Accuracy: 96.60713958740234 %
Iteration: 2500  Loss: 0.06380531191825867  Accuracy: 96.66666412353516 %
Iteration: 3000  Loss: 0.017173264175653458  Accuracy: 96.78571319580078 %
Iteration: 3500  Loss: 0.04039861634373665  Accuracy: 96.9047622680664 %
Iteration: 4000  Loss: 0.008767378516495228  Accuracy: 96.91666412353516 %
Iteration: 4500  Loss: 0.060781460255384445  Accuracy: 96.8452377319336 %
Iteration: 5000  Loss: 0.020362703129649162  Accuracy: 96.92857360839844 %
Iteration: 5500  Loss: 0.05642027035355568  Accuracy: 96.83333587646484 %
Iteration: 6000  Loss: 0.04280589893460274  Accuracy: 96.72618865966797 %
Iteration: 6500  Loss: 0.04761621356010437  Accuracy: 96.94047546386719 %
Iteration: 7000  Loss: 0.016610467806458473  Accuracy: 96.83333587646484 %
Iteration: 7500  Loss: 0.02146853506565094  Accuracy: 97.04762268066406 %
Iteration: 8000  Loss: 0.09096607565879822  Accuracy: 96.71428680419922 %
Iteration: 8500  Loss: 0.01594165712594986  Accuracy: 97.01190185546875 %
Iteration: 9000  Loss: 0.01295202225446701  Accuracy: 97.03571319580078 %
Iteration: 9500  Loss: 0.005850407294929027 6  Accuracy: 96.83333587646484 %
```

```
# 손실(오차) 시각화
plt.plot(iteration_list,loss_list)
plt.xlabel("Number of iteration")
plt.ylabel("Loss")
plt.title("ANN: Loss vs Number of iteration")
plt.show()

# 정확도 시각화
plt.plot(iteration_list,accuracy_list,color = "red")
plt.xlabel("Number of iteration")
plt.ylabel("Accuracy")
plt.title("ANN: Accuracy vs Number of iteration")
plt.show()
```




- 위 그래프에서 반복이 진행될수록 손실은 점점 감소하는 반면, 정확도는 점점 증가하고 있다

## 🌀 Convolutional Neural Network(CNN)

- CNN은 이미지를 분류하는 데 유용함
- CNN의 단계
  - 라이브러리 임포트
  - 데이터셋 준비
    - 이전과 동일
  - Convolutional layer(합성곱 레이어): 입력 데이터에 대해 필터(커널)를 이용한 컨볼루션 연산을 수행하여 특징을 추출
    - 필터(kernels)를 사용하여 형상 지도 제작
    - 패딩: 필터를 적용한 후 원본 이미지의 차수 감소. 그러나 원본 이미지에 대한 정보를 보존하기 위해, 패딩을 이용하여 합성곱 레이어 이후에 feature map의 차원을 증가
    - 2개의 합성곱 레이어 사용
    - feature map 개수: out_channels = 16
    - feature map: 입력 데이터로부터 추출된 특징을 나타내는 3D 배열, 높이/너비/채널에 대한 정보 담고있음
    - 필터(kernel) 크기: 5*5
  - Pooling layer(풀링 레이어)
    - 이미지 처리와 같은 신경망에서 공간 차원을 줄이고 계산량을 감소시키기 위해 사용되는 레이어
    - 합성곱 레이어(feature map)의 출력에서 축약된 feature map을 준비
    - max pooling: 주어진 영역에서 최대값을 추출하는 방식, 풀링 영역(윈도우)이 입력을 스캔하면서 각 영역에서 가장 큰 값을 선택하여 특징 맵을 생성
    - max pooling을 위한 2 pooling layer
    - pooling size: 풀링 레이어에서 사용되는 풀링 윈도우의 크기, 여기서는 2*2
    - pooling window: 풀링 연산을 수행할 때, 입력 데이터를 격자로 나누어 각 격자에 대해 적용되는 영역
  - Flattening(평탄화): feature map(3차원)을 평탄화
  - Fully Connected Layer(전결합층, 완전 연결층)
    - 각 뉴런이 이전 층의 모든 뉴런과 연결되어 있는 층
    - 이전에 학습했던 인공신경망(ANN)
    - 또는 로지스틱 회귀 분석처럼 선형일 수 있지만, 마지막에는 항상 softmax 함수가 존재
    - 활성화 기능 사용 X

        - 로지스틱 회귀라고 생각할 수 있음
        - 합성곱 부분과 로지스틱 회귀 부분을 결합하여 CNN 모델 생성
    - 모델 클래스 인스턴스화
        - 모델 생성
    - 손실 인스턴스화
        - 교차 엔트로피 손실
        - 로지스틱 함수의 softmax도 보유
    - 최적화 함수 인스턴스화
        - SGD Optimizer
    - 모델 훈련 및 예측
- 시각화 그림을 보면 결과적으로 손실은 감소하는 반면, 정확도는 증가함
- 합성곱 계층 덕본에 모델이 더 잘 학습되었으며 정확도가 ANN 모델보다 우수함. 실제로 하이퍼파라미터를 조정하는 동안 반복이 증가하고 합성곱 신경망이 확장되면 정확도가 높아질 수 있지만, 실행시간이 많이 소요된다는 단점이 있음

```
# Import Libraries
import torch
import torch.nn as nn
from torch.autograd import Variable


# CNN 모델 생성
class CNNModel(nn.Module):
    def __init__(self):
        super(CNNModel, self).__init__() # __init__ 메서드에서 모델의 초기 설정을 수행
        # super(CNNModel, self).__init__()은 부모 클래스인 nn.Module의 초기화 메서드를 호출

        # 합성곱 레이어: 이미지나 음성과 같은 다차원 데이터에서 지역적인 패턴을 인식, 입력 데이터에 대해 커널(필터)을 이동시켜가며 각 위치에서의 특징을 추출
        # 지역적인 패턴을 학습하여 다양한 특징을 추출

        # 맥스 풀링: 공간 차원을 감소시키면서 주요한 특징을 강조, 각 영역에서 최댓값을 추출하여 다음 층으로 전달
        # 중요한 특징을 강조하고, 계산량을 줄여 속도를 향상

        # 합성곱 레이어 1
        # nn.Conv2d: 2차원 합성곱 레이어를 정의하는 클래스
        # 입력 채널 수: 1, 출력 채널 수: 16, 커널 크기: 5x5, 스트라이드: 1, 패딩: 0
        self.cnn1 = nn.Conv2d(in_channels=1, out_channels=16, kernel_size=5, stride=1, padding=0)
        self.relu1 = nn.ReLU() # 신경망의 비선형성을 도입하기 위해 사용

        # Max pool 1
        # nn.MaxPool2d: 2차원 최대 풀링 레이어, 주어진 영역에서 최댓값을 선택하여 다운샘플링(데이터 크기 작게)을 수행하는 연산
        # 2x2 크기의 맥스 풀링
        self.maxpool1 = nn.MaxPool2d(kernel_size=2)

        # 합성곱 레이어 2
        # 입력 채널 수: 16, 출력 채널 수: 32, 커널 크기: 5x5, 스트라이드: 1, 패딩: 0
        self.cnn2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=5, stride=1, padding=0)
        self.relu2 = nn.ReLU() # 신경망의 비선형성을 도입하기 위해 사용

        # Max pool 2
        # 2x2 크기의 맥스 풀링
        self.maxpool2 = nn.MaxPool2d(kernel_size=2)

        # 완전 연결 층 1
        # 모든 입력 뉴런과 출력 뉴런이 서로 연결되어 있는 층
        # 입력 크기: 32 * 4 * 4, 출력 크기: 10
        self.fc1 = nn.Linear(32 * 4 * 4, 10)

    def forward(self, x): # 순전
        # Convolution 1

        # 입력 데이터 x에 대해 첫 번째 합성곱 레이어인 cnn1을 적용
        # 입력 이미지에 커널을 적용하여 특징 맵을 생성
        out = self.cnn1(x)

        # 첫 번째 합성곱 레이어의 출력에 ReLU(활성화 함수)를 적용. 비선형성 추가
        out = self.relu1(out)

        # Max pool 1
        # 첫 번째 최대 풀링 레이어를 적용
        # 특징 맵을 다운샘플링하여 공간 차원을 줄
        out = self.maxpool1(out)

        # Convolution 2
        out = self.cnn2(out)
        out = self.relu2(out)

        # Max pool 2
        out = self.maxpool2(out)

        # flatten
        # 3차원 텐서를 1차원으로 평탄화
        # fully connected layer에 입력하기 위해 특징 맵의 모양을 변경하는 과정
        out = out.view(out.size(0), -1)
```

```
        # Linear function (readout)
        # 평탄화된 특징을 fully connected layer fc1에 적용하여 최종 출력을 얻음
        # 입력된 특징을 10개의 클래스로 매핑
        out = self.fc1(out)

        return out


# batch_size, epoch and iteration
batch_size = 100
n_iters = 2500
num_epochs = n_iters / (len(features_train) / batch_size)
num_epochs = int(num_epochs)

# Pytorch train and test sets
# torch.utils.data: PyTorch에서 데이터셋과 데이터로더를 관리하기 위한 유틸리티 함수와 클래스를 제공하는 서브 모듈
# featuresTrain과 targetsTrain은 훈련 데이터의 입력 특성과 타겟(레이블)을 나타내는 PyTorch 텐서
# TensorDataset 클래스: 입력된 텐서들을 묶어서 하나의 데이터셋으로 만들어줌
# 각각의 텐서는 동일한 인덱스에 있는 데이터들끼리 연결
train = torch.utils.data.TensorDataset(featuresTrain,targetsTrain)
test = torch.utils.data.TensorDataset(featuresTest,targetsTest)

# data loader 반복문에서 간편하게 미니배치 단위로 데이터를 추출

# PyTorch의 DataLoader를 사용하여 훈련 데이터셋과 테스트 데이터셋을 미니배치로 나누고 셔플링함
# train은 훈련 데이터셋을 나타내는 TensorDataset 객체, 'test는 테스트 데이터셋을 나타내는 TensorDataset 객체
# batch_size는 미니배치의 크기를 지정하는 매개변수로, 훈련 데이터를 미니배치로 나눔
# shuffle은 데이터를 에포크마다 섞을지 여부를 나타내는 매개변수로 여기서는 False로 설정하여 순서대로 데이터를 로드함
train_loader = torch.utils.data.DataLoader(train, batch_size = batch_size, shuffle = False)
test_loader = torch.utils.data.DataLoader(test, batch_size = batch_size, shuffle = False)

# CNN 모델 생성
model = CNNModel()

# 손실 함수로 Cross Entropy Loss 설정
# nn.CrossEntropyLoss()는 Cross Entropy Loss를 나타내는 PyTorch의 손실 함수
# 모델의 출력과 정답 레이블 사이의 손실을 계산
error = nn.CrossEntropyLoss()

# 최적화 알고리즘으로 SGD Optimizer 설정
# torch.optim.SGD는 확률적 경사 하강법(Stochastic Gradient Descent) 옵티마이저를 나타냄
# model.parameters()를 통해 모델의 학습 가능한 매개변수들을 전달하고, lr=learning_rate로 학습률을 설정
# 이 옵티마이저는 모델을 훈련할 때 사용될 역전파 알고리즘을 구현
learning_rate = 0.1
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)


# CNN model training
count = 0
loss_list = []
iteration_list = []
accuracy_list = []
for epoch in range(num_epochs): # 전체 데이터셋에 대해 몇 번 반복할지를 결정하는 에폭 루프
    # 훈련 데이터셋에서 미니배치를 가져오는 루프, enumerate 함수를 사용하여 미니배치의 인덱스(i)와 데이터(images, labels)를 순회
    for i, (images, labels) in enumerate(train_loader):

        # 데이터 전처리
        # 100개의 이미지가 한 미니배치에 있고 각 이미지는 1채널(흑백 이미지)이며, 크기가 28x28 픽셀임
        train = Variable(images.view(100,1,28,28))
        labels = Variable(labels)

        # 역전파 전에 저장된 그래디언트를 초기화
        optimizer.zero_grad()

        # 순전파, 모델에 입력 데이터를 전달하여 순전파를 수행하고 출력을 얻음
        outputs = model(train)

        # 모델의 출력과 정답 레이블을 사용하여 Cross Entropy Loss(손실)를 계산
        loss = error(outputs, labels)

        # 경사 계산, 역전파 수행
        loss.backward()

        # 옵티마이저(알고리즘 최적화)를 사용하여 모델의 매개변수를 업데이트
        optimizer.step()

        # 반복 횟수 업데이트
        count += 1

        # 일정한 간격으로 정확도를 계산하고 기록
        if count % 50 == 0:
            # 정확도 계산
            correct = 0
            total = 0
            # 테스트 데이터에 예측, 정확도 계산
            for images, labels in test_loader:

                test = Variable(images.view(100,1,28,28))

                # 순전파
                outputs = model(test)
```

```
                # 모델의 출력에서 최댓값을 찾아 예측 클래스를 가져옴
                # 최댓값을 찾아서 예측 클래스를 가져오는 것은 주로 분류 문제에서 모델의 출력 중에서 가장 높은 확률을 가지는 클래스를 선택하는 방법
                # 모델의 출력은 보통 소프트맥스(softmax) 함수를 통과하게 되면 각 클래스에 속할 확률값으로 나타남
                predicted = torch.max(outputs.data, 1)[1]

                # 전체 레이블 수를 더하고, 정확한 예측 수를 계산하여 correct에 더함

                total += len(labels)

                # 전체 correct predictions
                correct += (predicted == labels).sum()

            accuracy = 100 * correct / float(total)

            # 손실, 반복 저장
            # 정확도 및 손실 기록
            loss_list.append(loss.data)
            iteration_list.append(count)
            accuracy_list.append(accuracy)
        if count % 500 == 0: # 특정 반복 횟수에 도달할 때마다 현재 손실과 정확도를 출력하여 훈련 진행 상황을 모니터링
            # Print Loss
            print('Iteration: {}  Loss: {}  Accuracy: {} %'.format(count, loss.data, accuracy))
```

```
Iteration: 500  Loss: 0.12923818826675415  Accuracy: 96.52381134033203 %
Iteration: 1000  Loss: 0.044352903962135315  Accuracy: 97.55952453613281 %
Iteration: 1500  Loss: 0.05668875202536583  Accuracy: 97.82142639160156 %
Iteration: 2000  Loss: 0.01846305839717388  Accuracy: 98.08333587646484 %
```

```
# 손실 시각화
plt.plot(iteration_list,loss_list)
plt.xlabel("Number of iteration")
plt.ylabel("Loss")
plt.title("CNN: Loss vs Number of iteration")
plt.show()

# 정확도 시각화
plt.plot(iteration_list,accuracy_list,color = "red")
plt.xlabel("Number of iteration")
plt.ylabel("Accuracy")
plt.title("CNN: Accuracy vs Number of iteration")
plt.show()
```