Basic Configuration Of Pytorch Package

Pytorch는 딥러닝을 통해 Classification을 만들어 보기 좋은 python library 입니다. Pytorch를이용해 다양한 기계학습 방법론을 Data에 적용시켜 볼 수 있습니다. 이 사이트는 Python으로 기계학습의 여러 방법론 중 Deep Learning을 실습하기 위해 만들어졌습니다. 이 교육과정은 Python 환경에서 pytorch를 이용해 CNN 모델을 만들고 Gradient Descent Algorithm을 적용시켜 모델에 Data(Cat and Dog)를 학습시키고, 이를 통해 새로운 Data가 들어왔을 때 Cat 인지 Dog인지 구분할 수 있게 하는 Cat and Dog Classification Model을 만드는 것을 최종 목표로 합니다.

**1. torch** 메인 네임스페이스입니다. 텐서 등의 다양한 수학 함수가 포함되어져 있으며 Numpy와 유사한 구조를 가집니다.

**2. torch.autograd** 자동 미분을 위한 함수들이 포함되어져 있습니다. 자체 미분 가능 함수를 정의할 때 사용하는 기반 클래스인 'Function' 등이 포함되어져 있습니다.

**3. torch.nn** 신경망을 구축하기 위한 다양한 데이터 구조나 레이어 등이 정의되어 있습니다. 예로는 ReLU와 같은 활성화 함수, MSELoss와 같은 손실 함수들이 있습니다.

**4. torch.optim** 확률적 경사 하강법(Stochastic Gradient Descent, SGD)를 중심으로 한 파라미터 최적화 알고리즘이 구현되어져 있습니다.

**5. torch.utils.data** SGD의 반복 연산을 실행할 때 사용하는 미니 배치용 유틸리티 함수가 포함되어져 있습니다.

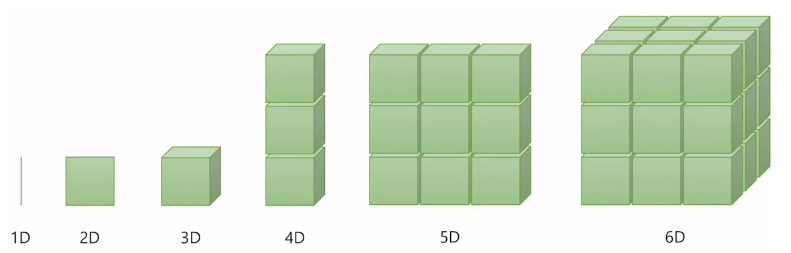
**6. torch.onnx** ONNX(Open Neural Network Exchange)의 포맷으로 모델을 익스포트(export)할 때 사용합니다. ONNX는 서로 다른 딥 러닝 프레임워크 간에 모델을 공유할 때 사용하는 포맷입니다.

Step 1 – Understanding of Tensor

Example은 단순 코드 적어놓으시고 사용자에게는 아무것도 안주시면 됩니다!

1 – 1 Tensor

Concept of Tensor



딥 러닝을 하게 되면 다루게 되는 가장 기본적인 단위는 벡터, 행렬, 텐서입니다. 차원이 없는 값을 스칼라(위의 그림에는 없음), 1차원으로 구성된 값을 우리는 벡터라고 합니다.

2차원으로 구성된 값을 행렬(Matrix)라고 합니다. 그리고 3차원이 되면 우리는 텐서(Tensor)라고 부릅니다. 사실 우리는 3차원의 세상에 살고 있으므로, 4차원 이상부터는 머리로 생각하기는 어렵습니다. 4차원은 3차원의 텐서를 위로 쌓아 올린 모습으로 상상해보겠습니다.

5차원은 그 4차원을 다시 옆으로 확장한 모습으로 생각해봅시다. 6차원은 5차원을 뒤로 확장한 모습으로 볼 수 있습니다.

* **옮긴이 주 : 데이터사이언스 분야 한정으로 3차원 이상의 텐서는 그냥 다차원 행렬 또는 배열로 간주할 수 있습니다. 또한 주로 3차원 이상을 텐서라고 하긴 하지만, 1차원 벡터나 2차원인 행렬도 텐서라고 표현하기도 합니다. 같은 표현입니다. 벡터 = 1차원 텐서, 2차원 행렬 = 2차원 텐서. 그리고 3차원 텐서, 4차원 텐서, 5차원 텐서 등...**

1 – 1 Example

아래와 같이 코드를 작성해보고 torch.FloatTensor()에 대해 dim(), size() shape을 사용해보세요

Input

import torch

#주석처리부분이 사용자가 구현해야 할 부분

def example1\_1():

    # t = torch.FloatTensor([[1., 2., 3.],

    #                    [4., 5., 6.],

    #                    [7., 8., 9.],

    #                    [10., 11., 12.]

    #                   ])

    # print(t)

    # print(t.dim()) # rank (차원)

    # print(t.size()) # shape

    # print(t.shape)

    return t

if \_\_name\_\_ == "\_\_main\_\_":

    #사용자가 봐야할 t의 결과

    t=example1\_1()

Output

tensor([[ 1., 2., 3.],

[ 4., 5., 6.],

[ 7., 8., 9.],

[10., 11., 12.]])

2

torch.Size([4, 3])

torch.Size([4, 3])

1 – 1 Practice

Example에서 배운 지식을 이용하여 FloatTensor의 shape에 대해서 생각해보고 Answer처럼 출력되는 t1, t2를 선언해보세요.

import torch

#주석처리부분이 사용자가 구현해야 할 부분

def practice1\_1():

    # t1 = torch.FloatTensor([[[1., 2., 3.],

    #                    [4., 5., 6.]],

    #                    [[7., 8., 9.],

    #                    [10., 11., 12.]]]

    #                   )

    # t2=torch.FloatTensor([[[[1]]]])

    return t1, t2

def judge\_1\_1(t1,t2):

    answer\_t1 = torch.FloatTensor([[[1., 2., 3.],

                       [4., 5., 6.]],

                       [[7., 8., 9.],

                       [10., 11., 12.]]]

                      )

    answer\_t2=torch.FloatTensor([[[[1]]]])

    t=torch.FloatTensor([0])

    if(type(t1)==type(answer\_t1))&(type(t2)==type(answer\_t2)):

        pass

    else:

        return False

    if(t1.shape==answer\_t1.shape)&(t2.shape==answer\_t2.shape):

        return True

    else:

        return False

if \_\_name\_\_=="\_\_main\_\_":

    t1,t2=practice1\_1()

    #사용자가 봐야할 t1,t2의 결과

    print(t1.shape)

    print(t2.shape)

    print(judge\_1\_1(t1,t2))

Answer

torch.Size([2,2,3])

torch.Size([1,1,1,1])

요구사항: print(“torch.Size([1, 1, 1, 1]”) 를 입력하진 않았는지 검사해 줘야한다.

1 – 2 Tensor Manipulation

Squeeze, Unsqueeze

스퀴즈(Squeeze)는 차원이 1인 경우에는 해당 차원을 제거합니다.

언스퀴즈(Unsqueeze)는 특정 위치에 1인 차원을 추가할 수 있습니다.

1 – 2 Example

torch.FloatTensor()에 대해 unsqueeze(), squeeze()를 사용해보세요!

input

import torch

#주석처리부분이 사용자가 구현해야 할 부분

def example1\_2():

    # t = torch.FloatTensor([[1., 2., 3.],

    #                    [4., 5., 6.],

    #                    [7., 8., 9.],

    #                    [10., 11., 12.]]

    #                   )

    # t1 = t.unsqueeze(0)

    # t2 = t.squeeze(0)

    return t1,t2

def judge\_1\_2ex(t1,t2):

    t=torch.FloatTensor([[1., 2., 3.],

                       [4., 5., 6.],

                       [7., 8., 9.],

                       [10., 11., 12.]]

                      )

    answer\_t1 = t.unsqueeze(0)

    answer\_t2 = t.squeeze(0)

    if(type(t1)==type(answer\_t1))&(type(t2)==type(answer\_t2)):

        pass

    else:

        return False

    if(t1.shape==answer\_t1.shape)&(t2.shape==answer\_t2.shape):

        return True

    else:

        return False

if \_\_name\_\_=="\_\_main\_\_":

    t1,t2=example1\_2()

    #사용자가 봐야할 t1,t2의 결과

    print(t1)

    print(t2)

    print(t1.shape)

    print(t2.shape)

    print(judge\_1\_2ex(t1,t2))

Output

tensor([[[ 1., 2., 3.],

[ 4., 5., 6.],

[ 7., 8., 9.],

[10., 11., 12.]]])

torch.Size([1, 4, 3])

torch.Size([4, 3])

1 – 2 Practice

아래의 Source code에 Squeeze와 Unsqueeze를 사용하여 Answer를 출력해보세요

import torch

#주석처리부분이 사용자가 구현해야 할 부분

def practice1\_2():

    t1 = torch.FloatTensor([[1., 2., 3.],

                       [4., 5., 6.],

                       [7., 8., 9.],

                       [10., 11., 12.]]

                      )

    t2=torch.FloatTensor([[1]])

    t1 = t1.unsqueeze(1).unsqueeze(0)

    t2 = t2.squeeze(0).squeeze(0)

    return t1, t2

#judge is\_answer => True or False

def judge\_1\_2(t1,t2):

    answer\_t1=torch.FloatTensor([[[[ 1.,  2.,  3.]],

         [[ 4.,  5.,  6.]],

         [[ 7.,  8.,  9.]],

         [[10., 11., 12.]]]])

    answer\_t2=torch.FloatTensor([1.]).squeeze(0)

    t=torch.FloatTensor([0])

    if(type(t1)==type(t)):

        pass

    else:

        return False

    if(t1.shape==answer\_t1.shape)&(t2.shape==answer\_t2.shape):

        return True

    else:

        return False

if \_\_name\_\_=="\_\_main\_\_":

    t1,t2=practice1\_2()

    #사용자가 봐야할 t1,t2의 결과

    print(t1)

    print(t2)

    print(judge\_1\_2(t1,t2))

Answer

tensor([[[[ 1., 2., 3.]],

[[ 4., 5., 6.]],

[[ 7., 8., 9.]],

[[10., 11., 12.]]]])

tensor(1.)

요구사항: print(“Answer”) 을 입력하진 않았는지 + unsqueeze와 squeeze를 사용했는지 검사해 줘야한다.

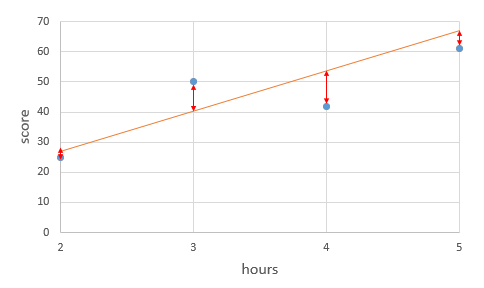
Step 2 – Gradient Descent Algorithm

2 - 1 Linear Regression

**선형 회귀(Linear Regression)**이란 학습 데이터와 가장 잘 맞는 하나의 직선을 찾는 일입니다.

선형 회귀의 가설(직선의 방정식)은 아래와 같은 형식을 가집니다.

y = Wx + b , x와 곱해지는 W를 가중치(Weight)라고 하며, b를 편향(bias)라고 합니다.

어떤 데이터를 이 가설에 대입하면 H(x) = Wx + b 라는 식으로 표현됩니다.

데이터와 직선이 얼마나 잘 맞는지에 대해 수학적인 방법으로 알 수 있습니다.

**비용 함수(cost function)** = **손실 함수(loss function)** = 오차 함수(error function) = 목적 함수(objective function)  
보통 비용 함수와 손실 함수라고 부릅니다.

n개의 ( x, y )의 학습데이터가 있는 상황을 가정했을 때 모든 데이터들은 임의의 직선과 차이인 오차를 가지고 있고, 음수와 양수 모두 존재하기 때문에 수학적인 방법을 통해서 Cost function을 표현해줘야 합니다. 이 값을 평균 제곱 오차(Mean Squared Error)로 정리해주면 아래와 같은 식으로 표현됩니다.

텍스트, 시계, 손목시계이(가) 표시된 사진

자동 생성된 설명

단순하게 생각해보면, cost function의 값이 가장 낮은 직선이 학습데이터와 가장 잘 맞는 직선이라 할 수 있을 것입니다.

2 – 1 Example

아래와 같이, Hypothesis와 Cost Function을 설정해 보고 출력해보세요

import torch

#주석처리부분이 사용자가 구현해야 할 부분

def example2\_1():

    # x\_train = torch.FloatTensor([[1], [2], [3]])

    # y\_train = torch.FloatTensor([[2], [4], [6]])

    # # 가중치 W를 0으로 초기화하고 학습을 통해 값이 변경되는 변수임을 명시합니다

    # W=torch.zeros(1, requires\_grad=True)

    # # 편향 b를 0으로 초기화하고 학습을 통해 값이 변경되는 변수임을 명시합니다

    # b=torch.zeros(1, requires\_grad=True)

    # # H(x)

    # hypothesis = x\_train \* W + b

    # # cost

    # cost = torch.mean((hypothesis - y\_train) \*\* 2 )

    # print(W, b)

    # print(hypothesis)

    # print(cost)

    # return hypothesis, cost

if \_\_name\_\_=="\_\_main\_\_":

    hypothesis,cost=example2\_1()

answer

tensor([0.], requires\_grad=True) tensor([0.], requires\_grad=True)

tensor([[0.],

[0.],

[0.]], grad\_fn=<AddBackward0>)

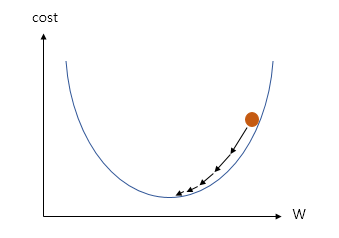
tensor(18.6667, grad\_fn=<MeanBackward0>)

2 – 1 Practice

2 – 2 Gradient Descent in Linear Regression

전 단계에서 배운 비용 함수의 값을 최소로 하는 W와 b를 찾는 방법들을 최적화(Optimizer) 알고리즘이라고 부릅니다. 이 옵티마이저 알고리즘을 통해 적절한 W와 b를 찾아내는 과정을 머신 러닝에서 학습(training)이라고 부릅니다. 경사하강법(Gradient Descent)은 가장 기본적인 옵티마이저 알고리즘중 하나입니다.

W와 cost의 관계를 그래프로 표현하면 다음과 같습니다.



Gradient Descent는 cost(W)를 최소로 하는 W를 cost(W)의 기울기를 통해 찾아내는 과정입니다.

위 그림에서 어떤 W값의 cost(W)의 기울기가 양수라면 W의 값이 감소 되어야 하고, 음수라면 W의 값이 증가 해야 합니다. 이를 수식으로 표현하면 다음과 같습니다.

텍스트이(가) 표시된 사진

자동 생성된 설명

여기서cost(W)의 기울기 값에 따라 얼마나 W의 값을 변경시켜줄지를 결정하는 α는 학습률(learning rate)이라 합니다.

텍스트, 안테나이(가) 표시된 사진

자동 생성된 설명

위 그림처럼 α 값에 따라 최적의 W값을 찾지 못하는 경우도 있기 때문에 적당한 α값을 설정하는 것이 중요합니다.

2 – 2 Example

위에서 배운 Hypothesis(직선의 방정식), Cost(비용 함수)를 이용해 경사 하강법을 구현해봅시다. lr은 학습률(learning rate)를 의미합니다. optimizer.zero\_grad()를 실행하게 되면 미분을 통해 얻은 기울기를 0으로 초기화합니다. 기울기를 초기화해야만 새로운 가중치 편향에 대해서 새로운 기울기를 구할 수 있습니다. 그 다음 cost.backward() 함수를 호출하면 가중치 W와 편향 b에 대한 기울기가 계산됩니다. 그 다음 경사 하강법 최적화 함수 optimizer의 .step() 함수를 호출하여 인수로 들어갔던 W와 b에서 리턴되는 변수들의 기울기에 학습률(learining rate) 0.01을 곱하여 빼줌으로써 업데이트합니다. 아래와 같이 작성해봅시다

import torch

#주석처리부분이 사용자가 구현해야 할 부분

def example2\_2():

    # 데이터

    x\_train = torch.FloatTensor([[1], [2], [3]])

    y\_train = torch.FloatTensor([[2], [4], [6]])

    # # 모델 초기화

    # W = torch.zeros(1, requires\_grad=True)

    # b = torch.zeros(1, requires\_grad=True)

    # # optimizer 설정

    # optimizer = torch.optim.SGD([W, b], lr=0.01)

    # nb\_epochs = 2000 # 원하는만큼 경사 하강법을 반복

    # for epoch in range(nb\_epochs + 1):

    #     # H(x) 계산

    #     hypothesis = x\_train \* W + b

    #     # cost 계산

    #     cost = torch.mean((hypothesis - y\_train) \*\* 2)

    #     # cost로 H(x) 개선

    #     optimizer.zero\_grad()

    #     cost.backward()

    #     optimizer.step()

    #     # 100번마다 로그 출력

    #     if epoch % 100 == 0:

    #         print('Epoch {:4d}/{} W: {:.3f}, b: {:.3f} Cost: {:.6f}'.format(

    #             epoch, nb\_epochs, W.item(), b.item(), cost.item()))

    return hypothesis, cost, W, b

def judge\_2\_2ex(cost,W,b):

    if(cost.item()<0.1):

        pass

    else:

        return False

    if(0<b<0.1)&(1.95<W<2.05):

        return True

    else:

        return False

if \_\_name\_\_=="\_\_main\_\_":

    hypothesis, cost, W, b = example2\_2()

    #사용자가 봐야할 Regression의 결과

    print(hypothesis)

    print(cost)

    print(W)

    print(b)

    print(judge\_2\_2ex(cost,W,b))

answer

Epoch 0/2000 W: 0.187, b: 0.080 Cost: 18.666666

Epoch 100/2000 W: 1.746, b: 0.578 Cost: 0.048171

Epoch 200/2000 W: 1.800, b: 0.454 Cost: 0.029767

Epoch 300/2000 W: 1.843, b: 0.357 Cost: 0.018394

Epoch 400/2000 W: 1.876, b: 0.281 Cost: 0.011366

Epoch 500/2000 W: 1.903, b: 0.221 Cost: 0.007024

Epoch 600/2000 W: 1.924, b: 0.174 Cost: 0.004340

Epoch 700/2000 W: 1.940, b: 0.136 Cost: 0.002682

Epoch 800/2000 W: 1.953, b: 0.107 Cost: 0.001657

Epoch 900/2000 W: 1.963, b: 0.084 Cost: 0.001024

Epoch 1000/2000 W: 1.971, b: 0.066 Cost: 0.000633

Epoch 1100/2000 W: 1.977, b: 0.052 Cost: 0.000391

Epoch 1200/2000 W: 1.982, b: 0.041 Cost: 0.000242

Epoch 1300/2000 W: 1.986, b: 0.032 Cost: 0.000149

Epoch 1400/2000 W: 1.989, b: 0.025 Cost: 0.000092

Epoch 1500/2000 W: 1.991, b: 0.020 Cost: 0.000057

Epoch 1600/2000 W: 1.993, b: 0.016 Cost: 0.000035

Epoch 1700/2000 W: 1.995, b: 0.012 Cost: 0.000022

Epoch 1800/2000 W: 1.996, b: 0.010 Cost: 0.000013

Epoch 1900/2000 W: 1.997, b: 0.008 Cost: 0.000008

Epoch 2000/2000 W: 1.997, b: 0.006 Cost: 0.000005

2 – 2 Practice

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| x | 25 | 50 | 42 | 61 | 80 |
| y | 81 | 159 | 130 | 180 | 244 |

위 Data에 대해 Linear Regression방식으로 Hypothesis와 Cost 식을 만들고 Gradient Descent을 적용하여 적절한 Learning Rate를 설정해보고, Cost<30를 만족하게 하는 W와 b를 찾아보세요.

input

import torch

#주석처리부분이 사용자가 구현해야 할 부분

def practice\_2\_2():

    # 데이터

    x\_train = torch.FloatTensor([[25], [50], [42],[61],[80]])

    y\_train = torch.FloatTensor([[81], [159], [130],[180],[244]])

    # # 모델 초기화

    # W = torch.zeros(1, requires\_grad=True)

    # b = torch.zeros(1, requires\_grad=True)

    # # optimizer 설정

    # optimizer = torch.optim.SGD([W, b], lr=0.0001)

    # nb\_epochs = 100 # 원하는만큼 경사 하강법을 반복

    # for epoch in range(nb\_epochs + 1):

    #     # H(x) 계산

    #     hypothesis = x\_train \* W + b

    #     # cost 계산

    #     cost = torch.mean((hypothesis - y\_train) \*\* 2)

    #     # cost로 H(x) 개선

    #     optimizer.zero\_grad()

    #     cost.backward()

    #     optimizer.step()

    #     # 100번마다 로그 출력

    #     if epoch % 10 == 0:

    #         print('Epoch {:4d}/{} W: {:.3f}, b: {:.3f} Cost: {:.6f}'.format(

    #             epoch, nb\_epochs, W.item(), b.item(), cost.item()))

    return hypothesis, cost, W, b

def judge2\_2(cost,W,b):

    if(cost.item()<):

        pass

    else:

        return False

    if(0<b<0.1)&(2.9<W<3.1):

        return True

    else:

        return False

if \_\_name\_\_=="\_\_main\_\_":

    hypothesis, cost, W, b = practice\_2\_2()

    print(judge2\_2(cost,W,b))

    print(hypothesis)

    print(cost)

    print(W)

    print(b)

answer

tensor([[ 76.5475],

[153.0248],

[128.5520],

[186.6748],

[244.7975]], grad\_fn=<AddBackward0>)

tensor(20.5626, grad\_fn=<MeanBackward0>)

tensor([3.0591], requires\_grad=True)

tensor([0.0704], requires\_grad=True)tensor([3.0591], requires\_grad=True)

Step 3 – Data Preprocessing

3 – 1 Loading Data

어떤 방식으로 불러올지 정해야 함

3 – 2 Cleaning datasets

cv2 module을 이용해 data를 학습에 쓰일 형태로 변형 시킬 수 있다.

Append를 이용해 학습에 쓰일 Datasets을 list로 정리할 수 있다.

3 – 2 Example

input

import torch

import os

import os.path as p

import cv2

from torch.utils.data import TensorDataset

root\_dir='C:/cd\_classifier/Practice/Step3'

animal\_data=[]

data\_labels=[]

img\_rows = 256

img\_cols = 256

# Get animal data

# for animal in ('cats/', 'dogs/'):

#     animal\_dir=p.join(root\_dir,animal)

#     os.chdir(animal\_dir)

#     files = os.listdir(animal\_dir)

#     for file in files:

#         if '.jpg' in file:

#             f = cv2.imread(file, 0)

#             f = torch.FloatTensor(cv2.resize(f, (img\_rows, img\_cols)))

#         animal\_data.append(f)

#         if animal == "cats/":

#             data\_labels.append(1)

#         else:

#             data\_labels.append(0)

# print(animal\_data[0])

# for i in animal\_data:

#     print(i.shape)

# for j in data\_labels:

#     print(j, end=' ')

output

tensor([[ 42., 42., 43., ..., 203., 197., 195.],

[ 43., 43., 43., ..., 198., 194., 191.],

[ 43., 42., 42., ..., 193., 192., 193.],

...,

[ 30., 27., 23., ..., 28., 31., 46.],

[ 31., 28., 23., ..., 32., 34., 46.],

[ 31., 27., 22., ..., 44., 36., 33.]])

torch.Size([256, 256])

torch.Size([256, 256])

torch.Size([256, 256])

torch.Size([256, 256])

torch.Size([256, 256])

torch.Size([256, 256])

torch.Size([256, 256])

torch.Size([256, 256])

torch.Size([256, 256])

torch.Size([256, 256])

torch.Size([256, 256])

torch.Size([256, 256])

torch.Size([256, 256])

torch.Size([256, 256])

torch.Size([256, 256])

torch.Size([256, 256])

torch.Size([256, 256])

torch.Size([256, 256])

torch.Size([256, 256])

torch.Size([256, 256])

1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0

3 – 2 Practice

아래의 Source code를 이용하여 Size([100 ,100, 1])인 Tensor들로 Label을 포함한 Dataset을 만들고, answer를 출력하세요. (Hint : unsqueeze를 이용하세요)

import torch

import os

import os.path as p

import cv2

from torch.utils.data import TensorDataset

root\_dir='C:/cd\_classifier/Practice/Step3'

animal\_data=[]

data\_labels=[]

img\_rows = 256

img\_cols = 256

#주석처리부분이 사용자가 구현해야 할 부분

# # Get animal data

# print("Get data...")

# for animal in ('cats/', 'dogs/'):

#     animal\_dir=p.join(root\_dir,animal)

#     os.chdir(animal\_dir)

#     files = os.listdir(animal\_dir)

#     for file in files:

#         if '.jpg' in file:

#             f = cv2.imread(file, 0)

#             f = torch.FloatTensor(cv2.resize(f, (img\_rows, img\_cols)))

#         animal\_data.append(f)

#         if animal == "cats/":

#             data\_labels.append(1)

#         else:

#             data\_labels.append(0)

# animal\_data=torch.stack(animal\_data)

# animal\_data=animal\_data.unsqueeze(1)

# data\_labels=torch.LongTensor(data\_labels)

# dataset=TensorDataset(animal\_data, data\_labels)

# print(dataset[0])

# print(dataset[10])

# for j in data\_labels:

#     print(j, end=' ')

Answer

(tensor([[[ 42., 42., 43., ..., 203., 197., 195.],

[ 43., 43., 43., ..., 198., 194., 191.],

[ 43., 42., 42., ..., 193., 192., 193.],

...,

[ 30., 27., 23., ..., 28., 31., 46.],

[ 31., 28., 23., ..., 32., 34., 46.],

[ 31., 27., 22., ..., 44., 36., 33.]]]), tensor(1))

(tensor([[[221., 140., 122., ..., 109., 98., 95.],

[222., 134., 114., ..., 61., 56., 57.],

[227., 128., 105., ..., 80., 78., 74.],

...,

[252., 245., 239., ..., 233., 230., 231.],

[255., 250., 245., ..., 237., 236., 236.],

[255., 253., 251., ..., 247., 246., 245.]]]), tensor(0))

tensor(1) tensor(1) tensor(1) tensor(1) tensor(1) tensor(1) tensor(1) tensor(1) tensor(1) tensor(1) tensor(0) tensor(0) tensor(0) tensor(0) tensor(0) tensor(0) tensor(0) tensor(0) tensor(0) tensor(0)

요구사항: print(“Answer”)을 입력하진 않았는지

Step 4 – CNN ( Convolutional Neural Network )

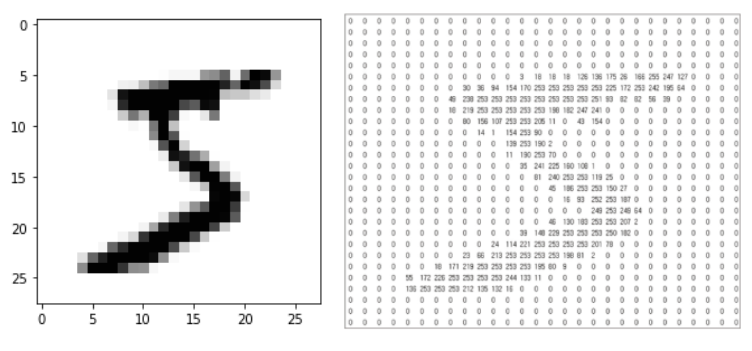
CNN이란?

Convolution Neural Network의 줄임말로 이미지 처리에 탁월한 성능을 보이는 딥러닝 모델 중 하나입니다.

왜 이미지 처리에 탁월한가?

이것에 대해선 모델은 Convolution (합성곱) 연산을 통해 이미지의 공간적인 구조 정보를 보존하면서 학습하기 때문입니다. 이때 공간적인 구조 정보란 이미지의 인접한 픽셀들끼리 어떤 관계가 있는지 파악할 수 있는 정보를 말합니다.

이미지 데이터를 예시로 설명을 해보겠습니다.



우선 이미지 데이터를 봅시다. 이미지 데이터는 MNIST 데이터를 예를 든 것입니다.

(28,28,1) 즉 위 이미지는 28x28=784개의 픽셀로 이루어져 있습니다, 만약 이미지가 컬러라면, RGB로 빨간색, 초록색, 파란색으로 (28,28,3) 총 3개의 **channel**로 이루어진다고 말합니다.

우리는 모델에 이대로 데이터를 넣는 것이 아니라, **이 픽셀들을 오른쪽 사진처럼 명암에 따라 숫자로 바꾼 뒤 이를 1차원으로 바꾸어 모델에 넣어줍니다.** (여기서 말하는 모델은 기존 다층 퍼셉트론을 생각하면됩니다. 이것을 모든 층들이 유기적으로 연결되어 Fully connected layer에 연결한다고 말합니다.)

(**\*Fully connected Layer(Fc-Layer)**는 말 그대로 모든 노드가 연결되어 있는 것을 의미하는데, 입력층을 거쳐 은닉층의 노드들이 모두 연결되어 가중치를 도출하는 기본적인 인공 신경망입니다.)

문제는 여기서 발생합니다. 3차원 이미지들을 1차원으로 바꾸어 Fully connected layer에 적용하면서, 각 픽셀들의 연관관계에 대한 정보 손실이 일어나고 같이 분류해야할 이미지를 올바르게 분류하지 못하게 됩니다.

텍스트, 낱말맞추기게임, 기타이(가) 표시된 사진

자동 생성된 설명

직관적으로 이해하자면, 둘의 이미지를 같은 y라고 사람들은 인식합니다. 이미지는 이렇게 한 덩어리로 봐야하지만 모델에는 1차원으로 바꿔서 입력하기 때문에 **각 픽셀간의 연관성의 정보 손실**이 일어나고, 심지어는 왼쪽의 y의 1차원 배열값과 오른쪽 y의 1차원 배열값이 한개씩 밀려있고 약간의 균열이 있어 똑같은 y라고 인식하지 못하게 되는겁니다.

이것을 해결하는 방법이 Convolutional layer를 추가하는 방식입니다.

CNN이라고 완전히 다른 신경망이 아닌 기존의 신경망에서 입력층 이전에 Convolutional layer(합성곱 층)을 추가해주는 겁니다. 합성곱 층에서 이미지 벡터들의 연관관계에 대한 정보들, 이미지 특유의 공간에 대한 정보들을 지닌 Feature Map을 입력층에 input해줌으로써 더 좋은 결과를 얻어내는 CNN 모델이 되는 것이죠.

합성곱의 과정으로 공간에 대한 정보를 가진 Feature Map을 만드는 과정을 봅시다. Stride = 1로 설정하여 Filter를 한 칸씩만 이동하여 Feature Map을 만들어냈습니다.

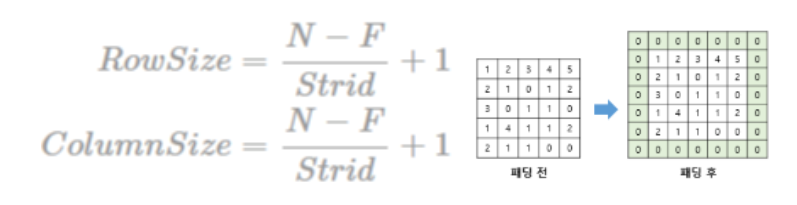
(\*Stride는 이미지 데이터에 적용할 Filter를 얼마나 움직여 적용할지를 정하는 단위)

(\*Feature Map은 이미지 데이터에 필터를 적용해 공간에 대한 정보를 지닌 변환 데이터)

두개의 함수를 이용해 적분한다는 합성곱의 개념이 여기에 적용됩니다. 필터에 적용된 가중치로 이미지 데이터의 부분 부분을 넓이로 표현한다는 개념이죠.

테이블이(가) 표시된 사진

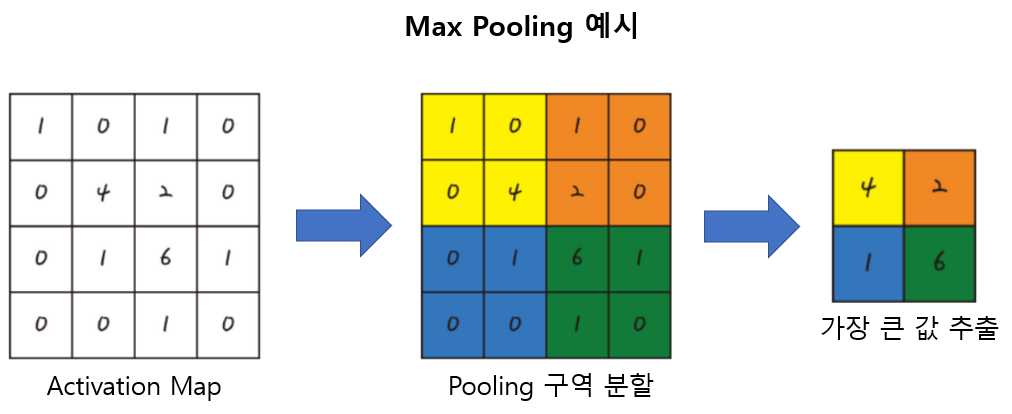
자동 생성된 설명



합성곱을 하면 위의 식처럼 피쳐맵이 작아지는 것을 알 수 있어요. (N은 이미지 데이터의 높이와 너비, F는 필터 사이즈)

**convolution 레이어의 출력 데이터가 작아지는 것을 방지하기 위해서는 위와 같이 어떤 수로 외곽을 둘러싸는padding 기법이 필요합니다.** **보통은 0으로 제로패딩을 이용합니다. 출력 데이터의 사이즈를 보호하면서도 인공 신경망에 외곽임을 인식시키는 효과도 있습니다.**

지금까지 컨볼루션 층으로부터 이미지의 특징들을 도출하였습니다. 하지만 한 번만으로는 데이터가 여전히 크고 복잡해 다시 한번 더 축소해야 합니다.**이 과정을 Pooling또는 sub sampling이라고 합니다.**



그림은 풀링 중에서도 가장 보편적인 **MaxPooling** 예시입니다. **컨볼루션으로 도출한 Activation Map(=Feature Map)의 구역을 분할하여 그 중** **가장 큰 수를 구역의 대표값으로 정하는 것입니다.**

다음은 기본적인 CNN 모델의 예시 코드입니다.

class CNN(nn.Module):

    def \_\_init\_\_(self):

        super().\_\_init\_\_()

        self.conv1 = nn.Conv2d(1, 4, kernel\_size=3, stride=1, padding=1)

        self.conv2 = nn.Conv2d(4, 8, kernel\_size=3, stride=1, padding=1)

        self.conv3 = nn.Conv2d(8, 16, kernel\_size=3, stride=1, padding=1)

        self.fc1 = nn.Linear(16 \* 32 \* 32, 256)

        self.fc2 = nn.Linear(256, 1)

    def forward(self, x):

        x = x.view(-1, 1, 256, 256)

        x = self.conv1(x)

        x = F.relu(x)

        x = F.avg\_pool2d(x, 2)

        x = self.conv2(x)

        x = F.relu(x)

        x = F.avg\_pool2d(x, 2)

        x = self.conv3(x)

        x = F.relu(x)

        x = F.avg\_pool2d(x, 2)

        x = x.view(x.size(0), -1)   # Flatten them for FC

        x = self.fc1(x)

        x = F.relu(x)

        logits = self.fc2(x)

        return logits

Quiz – Convolution 계산 중 max pooling을 통한 결과에 해당하는 값은 무엇인가?

텍스트이(가) 표시된 사진

자동 생성된 설명

정답 :

= 4

= 8

= 2

= 5**Step 5 – Training Process**

이제 모델을 정말로 학습시킬 차례입니다.

먼저 앞서 설계한 모델을 정의합니다.

model = CNN().to(device)

이어서 learning rate와 몇 번을 학습할지를 나타내는 epoch,

Loss function, optimizer와 같은 학습에 사용되는 hyperparameter를 정의해 줍니다.

import torch

import torch.nn as nn

# Hyperparameter

BATCH\_SIZE = 32

learning\_rate = 0.00005

epochs = 10

criterion = nn.BCEWithLogitsLoss()

optimizer = torch.optim.AdamW(model.parameters(), lr=learning\_rate)

### 손실 함수(loss function)

학습용 데이터를 제공하면, 학습되지 않은 신경망은 정답을 제공하지 않을 확률이 높습니다. **손실 함수(loss function)**는 획득한 결과와 실제 값 사이의 틀린 정도(degree of dissimilarity)를 측정하며, 학습 중에 이 값을 최소화하려고 합니다. 주어진 데이터 샘플을 입력으로 계산한 예측과 정답(label)을 비교하여 손실(loss)을 계산합니다.

일반적인 손실함수에는 회귀 문제(regression task)에 사용하는 [nn.MSELoss](https://pytorch.org/docs/stable/generated/torch.nn.MSELoss.html" \l "torch.nn.MSELoss)(평균 제곱 오차(MSE; Mean Square Error))나 분류(classification)에 사용하는 [nn.NLLLoss](https://pytorch.org/docs/stable/generated/torch.nn.NLLLoss.html" \l "torch.nn.NLLLoss) (음의 로그 우도(Negative Log Likelihood)), 그리고 nn.LogSoftmax와 nn.NLLLoss를 합친 [nn.CrossEntropyLoss](https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html" \l "torch.nn.CrossEntropyLoss) 등이 있습니다.

Dataloader를 통해 dataset으로부터 모델에 들어갈 형태로 loader를 설정해주면 학습에 필요한 준비는 끝났습니다.

from torch.utils.data import TensorDataset, DataLoader

# Make Dataloader

print("Get Dataloader")

train\_loader = DataLoader(

    dataset=train\_ds,

    batch\_size=BATCH\_SIZE,

    shuffle=True

)

valid\_loader = DataLoader(

    dataset=valid\_ds,

    batch\_size=BATCH\_SIZE,

    shuffle=False

)

test\_loader = DataLoader(

    dataset=test\_ds,

    batch\_size=BATCH\_SIZE,

    shuffle=True

)

다음은 학습이 실제로 진행되는 코드입니다.

# Train

print("Start Training...")

os.makedirs(ckpt\_path, exist\_ok=True)

best\_epoch = 0

for epoch in range(epochs):

    model.train()

    epoch\_loss = 0.0

    for step, (inputs, labels) in enumerate(tqdm(train\_loader)):

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

        optimizer.zero\_grad()

        logits = model(inputs)

        loss = criterion(logits.squeeze(1), labels.float())

        loss.backward()

        optimizer.step()

        epoch\_loss += loss.item()

    print(f"[Epoch {epoch + 1}],  loss: {epoch\_loss/(step + 1): .3f}")

    train\_loss.append(epoch\_loss / len(train\_loader))

model.train()을 통해 모델이 loss를 학습할 수 있는 상태임을 지정하고

Dataloader에서 하나씩 mini-batch를 꺼내어 학습을 진행합니다.

Logits가 모델이 내놓는 output이라고 생각을 하시면 됩니다.

미리 정의해둔 criterion (=loss function)을 통해 모델이 내놓는 logits와 실제 정답인 label의 차이를 통해 loss를 계산합니다.

loss.backward()를 통해 모델이 학습을 할 수 있도록 하고

optimizer.step()을 호출하여 역전파 단계에서 수집된 변화도(loss)로 매개변수를 조정합니다.

Quiz\_1 (객관식) Classification 과정에서 다음 중 아래와 같은 텐서값을 만드는 함수로 올바른 것은?

(label의 개수 = 3)

tensor[0.2, 0.1, 0.7]

정답 : softmax

오답 : sigmoid, linear, binary\_cross\_entropy, mean\_squared\_error

Quiz\_2 (객관식) 다음 train 과정에서 QUIZ 사이에 들어갈 코드로 알맞은 것은?

print("Start Training...")

best\_epoch = 0

for epoch in range(epochs):

model.train()

epoch\_loss = 0.0

for step, (inputs, labels) in enumerate(tqdm(train\_loader)):

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

optimizer.zero\_grad()

logits = model(inputs)

loss = criterion(logits.squeeze(1), labels.float())

\*\*\* QUIZ \*\*\*

\*\*\* QUIZ \*\*\*

optimizer.step()

epoch\_loss += loss.item()

print(f"[Epoch {epoch + 1}], loss: {epoch\_loss/(step + 1): .3f}")

train\_loss.append(epoch\_loss / len(train\_loader))

정답 :

loss.backward()

오답 :

model.train()

loss.calculate()

print(loss)

**Step 6 – Validation Process**

이젠 학습 시 over-fitting을 방지하기 위한 validation 과정을 작성할 차례입니다.

Validation 과정은 train과 같이 진행됩니다.

# Train

print("Start Training...")

os.makedirs(ckpt\_path, exist\_ok=True)

best\_epoch = 0

for epoch in range(epochs):

    model.train()

    epoch\_loss = 0.0

    for step, (inputs, labels) in enumerate(tqdm(train\_loader)):

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

        optimizer.zero\_grad()

        logits = model(inputs)

        loss = criterion(logits.squeeze(1), labels.float())

        loss.backward()

        optimizer.step()

        epoch\_loss += loss.item()

    print(f"[Epoch {epoch + 1}],  loss: {epoch\_loss/(step + 1): .3f}")

    train\_loss.append(epoch\_loss / len(train\_loader))

    with torch.no\_grad():

        current\_loss = 0.0

        for step, (inputs, labels) in enumerate(valid\_loader):

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

            logits = model(inputs)

            loss = criterion(logits.squeeze(1), labels.float())

            current\_loss += loss.item()

        valid\_loss = current\_loss / len(valid\_loader)

    if valid\_loss < best\_loss:

            best\_loss = valid\_loss

            save\_ckpt(

                ckpt\_path=p.join(ckpt\_path, f"checkpoint\_epoch\_{epoch + 1}.pt"),

                model=model, optimizer=optimizer,

                epoch=epoch+1,

                train\_loss=train\_loss, best\_loss=valid\_loss

            )

            best\_epoch = epoch + 1

            print(f"Success to save checkpoint. Best loss so far: {best\_loss: .3f}")

    else:

        print("No improvement detected. Skipping save")

with torch.no\_grad()를 통해 validation 과정의 모든 연산들이 gradient를 계산하지 않도록 합니다.

나머지 부분은 train과 동일하게 진행됩니다.

그 중 validation loss가 가장 낮은 상태를 저장하여 사용합니다.

Quiz\_1 (주관식) 다음과 같은 설명이 의미하는 코드는 무엇인가?

gradient 계산 contxt를 비활성화 해주는 역할을 하며, inference, validation을 할 때 사용됨.

따라서 필요한 메모리를 줄여주고 연산속도를 증가시키는 역할을 함.

정답 : with torch.no\_grad():

Exercise (객관식) 다음 validation 과정에서 QUIZ 사이에 들어갈 코드로 알맞은 것은?

\*\*\* QUIZ \*\*\*

\*\*\* QUIZ \*\*\*

current\_loss = 0.0

for step, (inputs, labels) in enumerate(valid\_loader):

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

logits = model(inputs)

loss = criterion(logits.squeeze(1), labels.float())

current\_loss += loss.item()

valid\_loss = current\_loss / len(valid\_loader)

정답 :

with torch.no\_grad():

오답 :

model.train()

loss.backward()

for step, (inputs, labels) in enumerate(tqdm(train\_loader)):

**Step 7 – Test Process**

이제 마지막으로 만든 model이 test data에 대해 얼마나 예측 성능을 보이는지 알아볼 차례입니다.

# Test

print("Start Test...")

test\_correct = 0

test\_total = 0

test\_model = CNN()

test\_model, optimizer, ckpt\_epoch, train\_loss, best\_loss = load\_ckpt(

        ckpt\_path=p.join(ckpt\_path, f"checkpoint\_epoch\_{best\_epoch}.pt"),

        model=model,

        optimizer=optimizer

    )

test\_model = test\_model.to(device)

with torch.no\_grad():

    for inputs, labels in test\_loader:

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

        logits = test\_model(inputs)

        loss = criterion(logits.squeeze(1), labels.float())

        pred = torch.sigmoid(logits.squeeze(1))

        pred = torch.round(pred)

        test\_total += labels.size(0)

        test\_correct += (pred == labels).sum().item()

test\_acc = test\_correct / test\_total

test\_acc = test\_acc \* 100

print('TestAcc: %.2f' % (test\_acc))

다른 부분들은 validation과 동일하지만 Test 과정에서는 sigmoid 함수를 통해 0~1 사이의 값으로 변환시켜 줍니다.

그리고 이를 반올림하여 모델이 0 또는 1을 예측하게 하는 것이죠.

예측한 결과와 실제 label을 비교하여 얼마나 일치했는지를 통해 모델의 성능을 평가합니다.

Quiz\_1 (주관식) 다음과 같은 결과에서 accuracy를 계산하여라.

prediction : 0, 1, 1, 0, 0

label : 0, 1, 1, 1, 0

정답 : 0.8

References: 위키독스 : wicidocs.net/32471

파이토치 공식 튜토리얼

모두를 위한 딥 러닝 시즌2 Pytorch