3.2 경사하강법으로 이미지 복원하기

3.2.1 오염된 이미지 문제

- 이미지 처리를 위해 만들어 두었던 weird_function() 함수에 실수로 버그가 들어가 100 x 100 픽셀의 오염된 이미지 생성
- 오염된 이미지와 오염되기 전 원본 이미지를 동시에 파일로 저장하려 했으나 원본 이미지 파일은 삭제된 상황
- 오염된 이미지와 weird_function() 함수를 활용해 원본 이미지 복원

3.2.2 오염된 이미지를 복원하는 방법

(1) 오염된 이미지(broken_image)와 크기가 같은 랜덤 텐서(random_tensor)를 생성
(2) 랜덤 텐서를 weird_function() 함수에 입력해 똑같이 오염된 이미지를 가설(hypothesis)이라고 부름
  a. [사실]원본 이미지가 weird_function()함수에 입력되어 오염된 이미지를 출력
  b. [사실]인위적으로 생성한 무작위 이미지가 weird_function()함수에 입력되어 가설을 출력
(3) 가설과 오염된 이미지가 같다면, 무작위 이미지와 원본 이미지도 같을 것
(4) 그러므로 weird_function(random_tensor) = broken_image 관계가 성립하도록 만듬

- 원본 이미지가 weird_function() 함수를 통해 이미지가 오염
- 원본 이미지와 동일한 크기의 랜덤 텐서를 생성하여
- 랜덤 텐서가 weird_function() 함수를 통해 이미지가 동일하게 오염되면 원본 이미지와 랜덤 텐서가 같을 것이라는 가정
- 원본 이미지와 마찬가지로 똑같이 오염된 이미지(가설)과 원본 이미지(정답) 사이의 거리가 최솟값이 되도록 랜덤 텐서를 바꿔주는 것이 목표

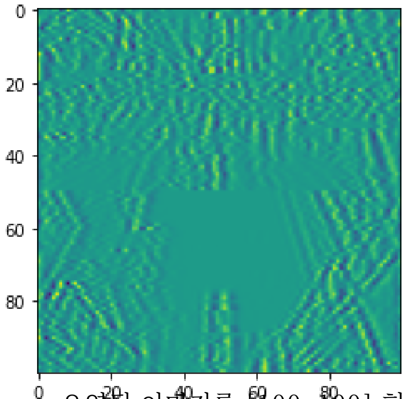
3.2.3 문제 해결과 코드 구현

```
import torch
import pickle  # 파이썬 객체를 파일 형태로 저장할 때 쓰는 라이브러리
import matplotlib.pyplot as plt

broken_image = torch.FloatTensor( pickle.load(open('/content/broken_image_t.p', 'rb'),encoding='latin1' ) )

print(broken_image)
print("Size:", broken_image.size())
print("랭크(차원):", broken_image.ndimension())
```

```
tensor([-0.0095, -0.0004,  0.0094,  ..., -0.0014, -0.0001, -0.0036])
Size: torch.Size([10000])
랭크(차원): 1
```

- 오염된 이미지(broken_image)를 텐서 형태로 불러오기
- encoding='latin1' 글자나 이미지가 깨지지 않게 하기 위한 인코딩 옵션
- 오염된 이미지(broken_image)는 랭크 1의 벡터로 표현한 텐서 데이터

```
plt.imshow(broken_image.view(100,100))
```



- 오염된 이미지를 [100, 100] 형태로 시각화하여 이미지 확인

```
def weird_function(x, n_iter=5):
    h = x
    filt = torch.tensor([-1./3, 1./3, -1./3])
    for i in range(n_iter):
        zero_tensor = torch.tensor([1.0*0])
        h_l = torch.cat( (zero_tensor, h[:-1]), 0)
        h_r = torch.cat((h[1:], zero_tensor), 0 )
        h = filt[0] * h + filt[2] * h_l + filt[1] * h_r
        if i % 2 == 0:
            h = torch.cat( (h[h.shape[0]//2:],h[:h.shape[0]//2]), 0 )
    return h
```

- 이미지를 오염시키는 weird_function() 함수의 코드
- 머신러닝을 이용하여 이미지를 복원하기 때문에 위의 코드는 이해할 필요 없음

```
def distance_loss(hypothesis, broken_image):
    return torch.dist(hypothesis, broken_image)
```

- 기존의 오염된 이미지(broken_image)와 가설(hypothesis : 랜덤 텐서가 함수를 통해 오염된 이미지)과 사이의 거리 정의
- torch.dist()는 두 텐서 사이의 거리를 구하는 함수

```
random_tensor = torch.randn(10000, dtype = torch.float)
# dtype = torch.float 실수 범위 내에서 랜덤 값을 뽑기 위한 옵션
```

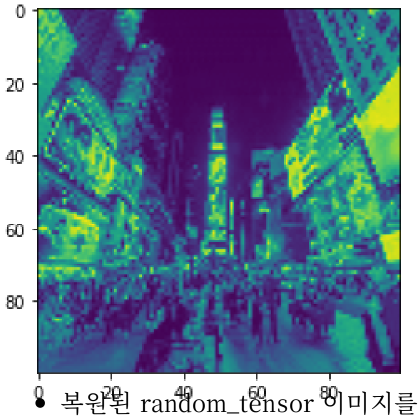
- 오염된 이미지 [100, 100] 형태와 동일한 랜덤 텐서 생성

```
# learning rate 설정
lr = 0.8
```

```
for i in range(0,20000):
# for문을 통해 경사하강법 20000번 수행
    random_tensor.requires_grad_(True)
# 오차함수를 random_tensor로 미분하기 위해 True 설정
    hypothesis = weird_function(random_tensor)
# random_tensor를 weird_function() 함수에 통과시켜 가설 설정
    loss = distance_loss(hypothesis, broken_image)
# 앞에서 정의한 distance_loss() 함수에 가설과 오염된 이미지를 입력하여 오차 계산
    loss.backward()
# 이 후 backward() 함수를 호출해 loss를 미분
    with torch.no_grad():
# 직접 경사하강법을 구현하기 때문에 torch.no_grad() 함수를 이용해 for문 실행시 파이토치의 자동 기울기 계산을 비활성화 설정
        random_tensor = random_tensor - lr*random_tensor.grad
# 최적화(optimizer)를 직접 구현
    if i % 1000 == 0:

# for문이 1000번 반복될 때마다 오차를 출력
        print('Loss at {} = {}'.format(i, loss.item()))
```

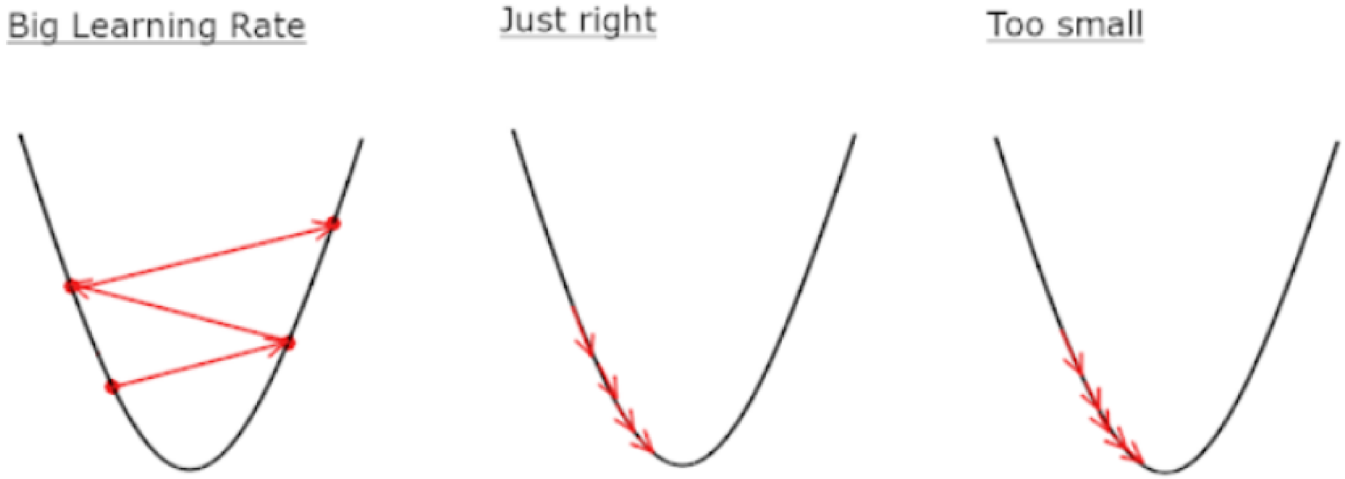
```
Loss at 0 = 12.377175331115723
Loss at 1000 = 1.083704948425293
Loss at 2000 = 0.5334588885307312
Loss at 3000 = 0.37594014406204224
Loss at 4000 = 0.2966930568218231
Loss at 5000 = 0.24751733243465424
Loss at 6000 = 0.21251222491264343
Loss at 7000 = 0.1849582940340042
Loss at 8000 = 0.16170020401477814
Loss at 9000 = 0.14112715423107147
Loss at 10000 = 0.12234684079885483
Loss at 11000 = 0.10482582449913025
Loss at 12000 = 0.0882231593132019
Loss at 13000 = 0.07231074571609497
Loss at 14000 = 0.056929461658000946
Loss at 15000 = 0.04196716099977493
Loss at 16000 = 0.027346517890691757
Loss at 17000 = 0.021157287061214447
Loss at 18000 = 0.021166130900382996
Loss at 19000 = 0.021167712286114693
```

```
plt.imshow(random_tensor.view(100,100).data)
```

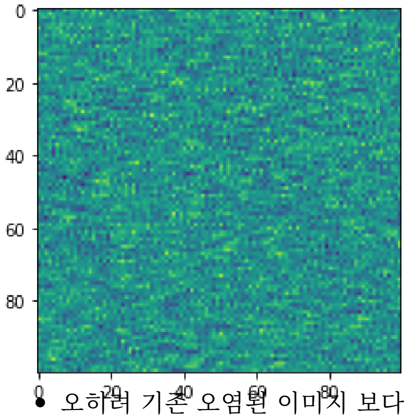


- 복원된 random_tensor 이미지를 출력
- 원본은 뉴욕 타임스퀘어의 풍경임을 알 수 있음

- **lr(learning rate, 학습률)**
  - 한 반복에서 최솟점으로 얼마나 많이 이동하는지, 학습을 얼마나 급하게 진행하는가를 정하는 매개변수
  - 경사하강법 알고리즘은 기울기에 학습률을 곱해 다음 지점을 결정
  - 학습률이 큰 경우 데이터가 무질서하게 이탈하며, 최솟점을 찾지 못함(최솟점을 지나침)
  - 학습률이 작은 경우 학습시간이 매우 오래 걸리며, 최솟점에 도달하지 못함(학습 속도가 느려짐)

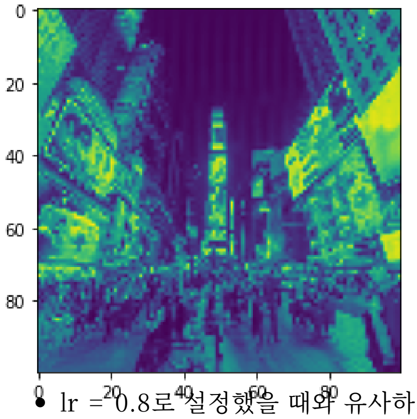


```
# 학습률 낮게 설정
lr = 0.01
```



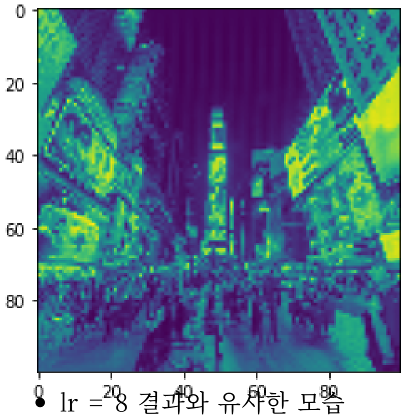
- 오히려 기존 오염된 이미지 보다 더 흐릿한 이미지를 생성

```
# 학습률 높게 설정
lr = 8
```



- lr = 0.8로 설정했을 때와 유사하게 이미지를 복원

```
# 학습률 매우 높게 설정
lr = 16
```



- lr = 8 결과와 유사한 모습

파일

clipboard-202203301404-zkiny.png	77.7 KB	2022-03-30	신재영
clipboard-202203301449-e8vfz.png	49.3 KB	2022-03-30	신재영
clipboard-202203301508-p5yf7.png	86 KB	2022-03-30	신재영
clipboard-202204050953-zpjoh.png	97.7 KB	2022-04-05	신재영
clipboard-202204050956-ekp0s.png	84.9 KB	2022-04-05	신재영
clipboard-202204050957-znong.png	85 KB	2022-04-05	신재영