Ch 04 신경망 학습

이번에는 훈련데이터로부터 가중치 매개변수의 값을 자동으로 획득하는 학습에 대해 배워봅니다. 신경망이 학습할 수 있도록 하는 데에는 지표인 손실 함수가 필요합니다. 이 손실 함수의 결과값을 가장 작게 만드는 가중치 매개변수를 찾는 것이 학습의 목표이고 이를 위한 경사법을 공부해볼 것입니다.

4.1 데이터에서 학습한다!

수천 개의 매개변수를 사람이 수동으로 설정하는 것은 불가능에 가까우므로, 드디어 매개변수를 자동으로 설정하는 방법에 대해서 알아볼 시간입니다.

4.1.1 데이터 주도 학습

문제에 대해 접근할 때, 어떤 규칙성(패턴)에 대해 생각하는 것은 일반적인 방법 중 하나입니다. MNIST 데이터를 예로 ‘5’라는 숫자를 어떻게 인식해 찾아낼 것인가에 대한 문제를 해결해봅시다. 사람이라면 쉽게 인식해 문제를 풀 수 있지만, ‘5’라는 숫자의 규칙성에 대해서 명확한 로직을 짜기 힘듭니다. 이럴 때 ‘5’를 인식하는 알고리즘을 설계하는 대신 데이터의 특징(feature)을 이용해 해결할 수 있습니다. 대부분의 특징은 벡터로 기술되고, 이미지를 벡터로 변환하여 변환된 데이터를 가지고 학습을 하는 기계 학습을 생각해 볼 수 있습니다. 다만, 이미지를 벡터로 변환할 때 사용하는 ‘특징’은 여전히 사람이 설계해야 합니다. 숫자 이미지와 얼굴 이미지에서 어떤 특징이 적절한지 사람이 잡아주어야 한다는 말입니다.

사람이 생각한 알고리즘

사람이 생각한 특징

기계학습 (SVM, KNN등)

신경망(딥러닝)

‘5’

결과

결과

결과

신경망은 두번째 방식과는 다릅니다. 신경망은 이미지에 포함된 중요한 특징까지 스스로 학습할 것입니다. 딥러닝은 입력을 그대로 ‘처음부터 끝까지’ 학습한다는 의미로 종단간 기계학습(end-to-end machine learning)이라고도 합니다.

4.1.2 훈련 데이터와 시험 데이터

신경망을 학습시킬 때 데이터는 굉장히 중요한 요소입니다. 일반적으로 훈련 데이터(training data)와 시험 데이터(test data)로 나누어 학습과 실험을 수행합니다. 이는 신경망 모델의 범용성(아직 보지 못한 데이터도 올바르게 풀어내는)을 평가하기 위한 것입니다. 만약 데이터셋 하나로 매개변수의 학습과 평가를 수행하면 그 데이터셋에만 매개변수가 지나치게 맞춰져 다른 데이터셋에는 결과가 엉망으로 나올 수 있습니다. 한 데이터셋에만 지나치게 최적화된 상태를 오버피팅(overfitting)이라고 합니다.

4.2 손실 함수

손실 함수(loss function, 또는 비용 함수(cost function))는 신경망 성능의 ‘나쁨’을 나타내는 지표입니다. 이 손실 함수를 최소화하는 가중치 매개변수를 찾는 일은 곧 신경망 성능을 좋게 만드는 것과 같습니다. 손실 함수는 일반적으로 오차제곱합과 교차 엔트로피 오차를 사용합니다.

4.2.1 오차제곱합

가장 많이 쓰이는 손실 함수는 오차제곱합(sum of squares for error, SEE)이고 수식은 다음과 같습니다.

[식 4.1]

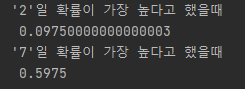
여기서 는 신경망의 출력(신경망이 추정한 값), 는 정답 레이블, k는 데이터의 차원 수를 나타냅니다.



y 데이터는 소프트맥스 함수의 출력이고 차례대로 손글씨 데이터 ‘0’, ‘1’, ‘2’, …일 때의 신경망이 추정한 값이고, t 데이터는 정답인 원소 1을 제외한 나머지를 0으로 표현한 원-핫 인코딩입니다.

파이썬 코드를 통해 오차제곱합을 구해보겠습니다.

import numpy as np  
  
  
def sum\_squares\_error(y, t):  
 return 0.5 \* np.sum((y-t)\*\*2)  
  
  
# '2'일 확률이 가장 높다고 추정함  
y1 = [0.1, 0.05, 0.6, 0.0, 0.05, 0.1, 0.0, 0.1, 0.0, 0.0]  
# '7'일 확률이 가장 높다고 추정함  
y2 = [0.1, 0.05, 0.1, 0.0, 0.05, 0.1, 0.0, 0.6, 0.0, 0.0]  
t = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]  
  
print("'2'일 확률이 가장 높다고 했을때\n", sum\_squares\_error(np.array(y1), np.array(t)))  
print("'7'일 확률이 가장 높다고 했을때\n", sum\_squares\_error(np.array(y2), np.array(t)))



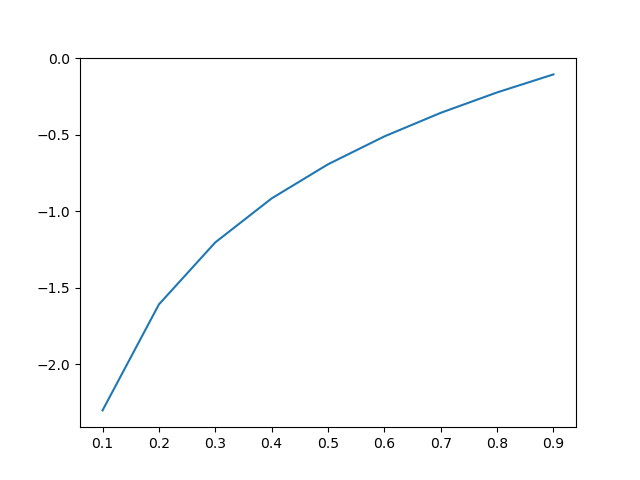
오차제곱합의 결과를 보면 ‘2’일 확률이 높다고 했을 때보다 ‘7’이라고 했을 때가 출력이 더 높습니다. 손실 함수는 출력이 작은 쪽이 좋은 것이고 첫 번째 예가 정답 레이블과의 오차가 작기 때문에 첫 번째 추정 결과가 정답에 더 가깝다고 판단할 수 있습니다.

4.2.2 교차 엔트로피 오차

교차 엔트로피 오차(cross entropy error, CEE)도 자주 이용되는 손실 함수입니다. 수식은 다음과 같습니다.

[식 4.2]

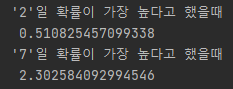
여기서 는 밑이 e인 자연 로그이고 는 신경망의 출력, 는 정답 레이블입니다. 는 정답에 해당하는 인덱스의 원소만 1이고 나머지는 0(원-핫 인코딩)이므로 실질적으로는 정답 가 1일 때의 의 자연로그를 계산하는 식이 됩니다. 다음은 자연로그 y=logx의 그래프입니다.



그래프에서 볼 수 있듯이 x가 1일 때 y는 0에 가까워지고, x가 0일 때 y의 값은 점점 작아집니다. 추정한 값이 1에 가까워질수록 오차가 적어진다는 의미입니다. 위 그래프에서 y 값이 점점 작아지다가 끊어진 것을 볼 수 있는데, np.log()에 들어갈 값이 0에 가까워질수록 -inf(마이너스 무한대)가 되어 더 이상 계산을 할 수 없기 때문입니다. 이를 위해 아주 작은 값을 더해서 CEE를 구현해야 합니다.

def cross\_entropy\_error(y, t):  
 delta = 1e-7  
 return -np.sum(t \* np.log(y + delta))

# CEE 실험  
# '2'일 확률이 가장 높다고 추정함  
y1 = [0.1, 0.05, 0.6, 0.0, 0.05, 0.1, 0.0, 0.1, 0.0, 0.0]  
# '7'일 확률이 가장 높다고 추정함  
y2 = [0.1, 0.05, 0.1, 0.0, 0.05, 0.1, 0.0, 0.6, 0.0, 0.0]  
t = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]  
  
print("'2'일 확률이 가장 높다고 했을때\n", cross\_entropy\_error(np.array(y1), np.array(t)))  
print("'7'일 확률이 가장 높다고 했을때\n", cross\_entropy\_error(np.array(y2), np.array(t)))



교차 엔트로피 오차의 코드와 결과는 위와 같습니다. 오차제곱합의 판단과 같이 첫 번째 추정이 정답일 가능성이 높다고 판단했습니다.

4.2.3 미니배치 학습

이때까지는 데이터 하나에 대한 손실 함수만 계산해봤습니다. 이제 배치를 이용해 훈련 데이터 전체에 대한 손실 함수의 합을 구하는 방법을 알아보겠습니다.

[식 4.3]

데이터가 N개라면 는 n번째 데이터의 k번째 값을 의미합니다. 이 식은 데이터 하나에 대한 손실 함수인 [식 4.2]를 N개의 데이터로 확장했을 뿐입니다. N으로 나누어 ‘평균 손실 함수’를 구하는 것입니다. 만약 MNIST 데이터셋처럼 훈련데이터가 60000개인데 모든 데이터를 대상으로 손실 함수의 합을 구하는 것은 시간이 꽤 걸립니다. 그래서 데이터의 일부를 추려 계산한 평균 손실 함수는 통계적 관점으로 전체의 ‘근사치’로 이용합니다. 예를 들어 60000개의 데이터 중에 100장을 무작위로 뽑아 100장 만을 사용하여 학습하는 것입니다. 이 데이터의 일부(앞의 예에서는 100장)를 미니배치(mini-batch)라고 하고 이러한 학습 방법을 미니배치 학습이라고 합니다. 미니배치를 구현해보도록 합시다.

import os, sys  
sys.path.append(os.pardir)  
import numpy as np  
from dataset.mnist import load\_mnist  
  
  
(x\_train, t\_train), (x\_test, t\_test) = \  
 load\_mnist(normalize=True, one\_hot\_label=True)  
  
train\_size = x\_train.size[0] # x\_train.shape = (60000, 784)  
batch\_size = 10 # 60000개의 데이터 중 10개만  
batch\_mask = np.random.choice(train\_size, batch\_size) # 0~59999의 수 중에서 무작위로 10개의 숫자를 뽑아냅니다.  
x\_batch = x\_train[batch\_mask] # 무작위로 뽑힌 숫자를 index로 하는 원소를 추출합니다.  
t\_batch = t\_train[batch\_mask]

4.2.4 (배치용) 교차 엔트로피 오차 구현하기

def cross\_entropy\_error(y, t):  
 if y.ndim == 1:  
 t = t.reshape(1, t.size)  
 y = y.reshape(1, y.size)  
  
 batch\_size = y.shape[0]

# return -np.sum(t \* np.log(y + 1e-7)) / batch\_size  
 return -np.sum(np.log(y[np.arange(batch\_size), t] + 1e-7)) / batch\_size

위 코드는 정답 레이블이 숫자 레이블로 주어졌을 때의 코드입니다. 정답 레이블이 ‘원-핫 인코딩’된 코드는 주석으로 적었습니다. 이 코드에서 y[np.arange(batch\_size), t]의 의미가 중요합니다. y.shape[0] 길이의 [0, 1, 2, …, batch\_size-1]의 배열(np.arange(batch\_size)은 y 데이터의 순서를 뜻하고 t는 정답 레이블에 대한 예측값을 뜻합니다. 즉, batch\_size가 5이고 t라는 정답 레이블이 [2, 7, 0, 9, 4]라면, y[np.arange(batch\_size), t]는 [y[0, 2], y[1,7], y[2, 0], y[3, 9], y[4, 4]], [(0번째 데이터에서 숫자 ‘2’에 관한 y의 예측값), (1번째 데이터에서 숫자 ‘7’에 관한 y의 예측값), …]라는 형식의 넘파이 배열을 추출해냅니다.

4.2.5 왜 손실 함수를 설정하는가?

신경망을 학습하면서 왜 ‘정확도’라는 지표는 사용하지 않고 ‘손실 함수’를 사용할까요? 이 의문에 답하려면 신경망 학습에서의 ‘미분’의 역할에 주목해야 합니다. 최적의 매개변수(가중치, 편향)을 탐색할 때 손실 함수의 값을 가능한 작게 만드는 매개변수를 찾습니다. 이때 매개변수의 미분을 계산하고, 그 미분 값을 단서로 매개변수의 값을 서서히 갱신하는 방법을 반복합니다.

가중치 매개변수의 손실 함수의 미분이란 ‘가중치 매개변수의 값을 아주 조금 변화시켰을 때, 손실 함수가 어떻게 변하나’의 의미입니다. 이때 미분 값이 0이면 매개변수를 어느 쪽으로 움직여도 손실 함수의 값은 줄어들지 않고 매개변수의 갱신이 멈추게 됩니다. 정확도를 지표로 삼으면 미분 값이 대부분의 장소에서 0이 된다고 하며 이러면 매개변수를 갱신할 수 없게 됩니다.

예를 들어, 정확도 32%(데이터 100개중 32개를 정확히 인식하는)를 가진 매개변수가 있다고 했을 때 매개변수를 약간씩 조정한다고 해도 정확도는 32.xxx%, 31.xxx%처럼 연속적으로 변하기 보다는 31%, 33%처럼 불연속적인 결과를 출력하기 때문에 손실 함수에 비해서 변화량에 대해 둔감합니다. 이는 ‘계단 함수’를 활성화 함수로 사용하지 않는 이유와 같습니다. 계단 함수는 매개변수의 작은 변화가 주는 파장을 무시해 손실 함수의 변화가 나타나지 않습니다. 이런 성질 때문에 어느 곳에서 미분을 해도 0이 되지 않는 시그모이드 함수 같은 활성화 함수를 사용합니다.

4.3 수치 미분

‘미분’에 대해 간단하게 복습하는 시간을 가지겠습니다.

4.3.1 미분

미분은 한순간의 변화량을 표시합니다. 수식은 다음과 같습니다.

[식 4.4]

이를 파이썬으로 구현하면 다음과 같습니다.

def numerical\_diff(f, x):  
 h = 10e-50  
 return (f(x+h) - f(x)) / h

위 코드는 두가지 문제점을 가지고 있습니다. h 값이 너무 작아 반올림 오차(rounding error)를 일으킬 수 있다는 것, f(x+h)와 f(x)의 차분은 결국 진정한 의미의 미분이 아니라는 것입니다. 첫 번째 문제점을 해결하기 위해 h = 1e-4 정도를 사용하면 좋은 결과를 얻는다고 합니다. 두 번째 문제점을 해결하는 것은 h가 0으로 무한히 가까워진다는 것이 불가능하기 때문에 애초에 구현의 여부에 문제가 있습니다. 이 때문에 수치 미분에는 오차가 포함될 수밖에 없기 때문에, 이 오차를 줄이기 위해 중심 차분 혹은 중앙 차분 (x+h)와 (x-h)의 차분)을 사용합니다. 개선한 코드는 다음과 같습니다. 오차 없이 수식을 전개해 미분을 구하는 방식을 해석 미분이라 합니다.

def numerical\_diff(f, x):  
 h = 1e-4  
 return (f(x+h) - f(x-h)) / (2 \* h)

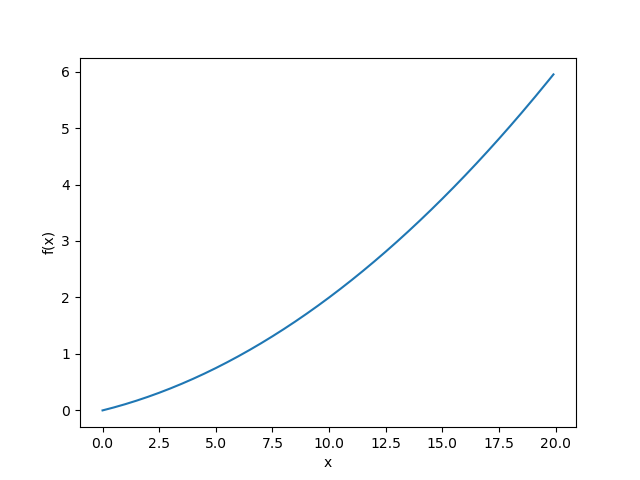
4.3.2 수치 미분의 예

파이썬 코드를 통해 [식 4.5]의 수치 미분을 구해보도록 합시다. x=5, 10 일 때 입니다.

import matplotlib.pyplot as plt  
import numpy as np  
from functions import numerical\_diff

def function\_1(x):  
 return 0.01 \* x\*\*2 + 0.1 \* x  
  
print(numerical\_diff(function\_1, 5))  
print(numerical\_diff(function\_1, 10))

x = np.arange(0.0, 20.0, 0.1)  
y = function\_1(x)  
plt.xlabel("x")  
plt.ylabel("f(x)")  
plt.plot(x, y)  
plt.show()





해석적 미분의 값인 0.2, 0.3과 비교해보았을 오차가 매우 작은 것을 알 수 있습니다.

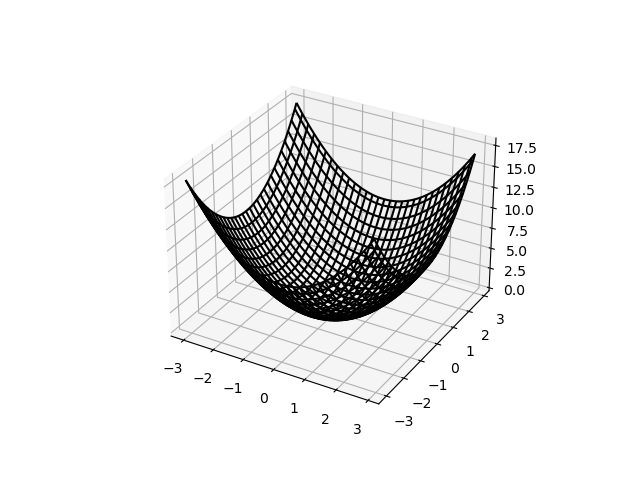
4.3.3 편미분

다음 식을 살펴봅시다.

[식 4.6]은 변수가 2개라는 점이 앞선 식과 다릅니다. 이 식도 파이썬으로 구현해보도록 합시다. 그래프는 3차원으로 그려집니다.

def function\_2(x1, x2):  
 return x1\*\*2 + x2\*\*2

x1 = np.arange(-3.0, 3.0, 0.1)  
x2 = np.arange(-3.0, 3.0, 0.1)  
x1, x2 = np.meshgrid(x1, x2)  
y = function\_2(x1, x2)  
  
fig = plt.figure()  
ax = fig.gca(projection='3d')  
surf = ax.plot\_wireframe(x1, x2, y, color='black')  
  
plt.show()





# x0 = 3, x1 = 4 일 때 x0에 대한 편미분  
def function\_tmp1(x0):  
 return x0 \* x0 + 4.0 \*\* 2  
  
  
# x0 = 3, x1 = 4 일 때 x1에 대한 편미분  
def function\_tmp2(x1):  
 return 3.0 \*\* 2 + x1 \* x1

print(numerical\_diff(function\_tmp1, 3.0))  
print(numerical\_diff(function\_tmp2, 4.0))

x0와 x1에 대한 각각의 수치 미분 값은 해석적 미분의 결과와 거의 같습니다.

4.4 기울기

앞서 편미분을 변수별로 따로 계산했습니다. 동시에 계산하고 싶다면 어떻게 할까요? (x0, x1) 양쪽을 묶어서 을 계산한다고 할 때 처럼 모든 변수의 편미분을 벡터로 정리한 것을 기울기(gradient)라 합니다.

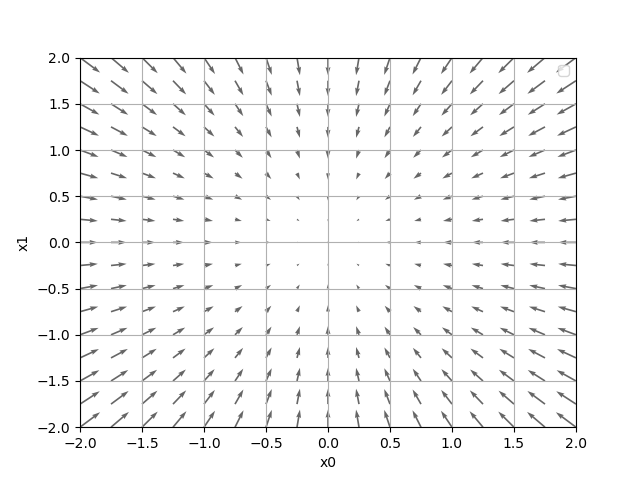
def numerical\_gradient(f, x):  
 h = 1e-4  
 grad = np.zeros\_like(x) # x와 형상이 같은 배열을 생성  
  
 for idx in range(x.size):  
 tmp\_val = x[idx]  
  
 # f(x+h)  
 x[idx] = tmp\_val + h  
 fxh1 = f(x)  
  
 # f(x-h)  
 x[idx] = tmp\_val - h  
 fxh2 = f(x)  
  
 grad[idx] = (fxh1 - fxh2) / (2 \* h)  
 x[idx] = tmp\_val  
  
 return grad

print(numerical\_gradient(function\_2, np.array([3.0, 4.0])))  
print(numerical\_gradient(function\_2, np.array([0.0, 2.0])))  
print(numerical\_gradient(function\_2, np.array([3.0, 0.0])))



결과가 뜻하는 것은 무엇일까요? 기울기의 결과에 마이너스를 붙인 벡터를 그려보면 알 수 있습니다. (교재에서 제공한 코드를 이용했습니다.)

def \_numerical\_gradient\_no\_batch(f, x):  
 h = 1e-4  
 grad = np.zeros\_like(x) # x와 형상이 같은 배열을 생성  
  
 for idx in range(x.size):  
 tmp\_val = x[idx]  
  
 # f(x+h)  
 x[idx] = tmp\_val + h  
 fxh1 = f(x)  
  
 # f(x-h)  
 x[idx] = tmp\_val - h  
 fxh2 = f(x)  
  
 grad[idx] = (fxh1 - fxh2) / (2 \* h)  
 x[idx] = tmp\_val  
  
 return grad  
  
  
def numerical\_gradient(f, X):  
 if X.ndim == 1:  
 return \_numerical\_gradient\_no\_batch(f, X)  
 else:  
 grad = np.zeros\_like(X)  
  
 for idx, x in enumerate(X):  
 grad[idx] = \_numerical\_gradient\_no\_batch(f, x)  
  
 return grad  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 x0 = np.arange(-2, 2.5, 0.25)  
 x1 = np.arange(-2, 2.5, 0.25)  
 X, Y = np.meshgrid(x0, x1)  
  
 X = X.flatten()  
 Y = Y.flatten()  
  
 grad = numerical\_gradient(function\_2, np.array([X, Y]))  
  
 plt.figure()  
 plt.quiver(X, Y, -grad[0], -grad[1], angles="xy", color="#666666")  
 plt.xlim([-2, 2])  
 plt.ylim([-2, 2])  
 plt.xlabel('x0')  
 plt.ylabel('x1')  
 plt.grid()  
 plt.legend()  
 plt.draw()  
 plt.show()



결과는 위 그림처럼 함수의 ‘가장 낮은 장소(최솟값)’을 가리킵니다. ‘가장 낮은 곳’에서 멀어질수록 화살표의 크기가 커지는 것도 알 수 있습니다. 중요한 점은 “기울기가 가리키는 쪽은 각 장소에서 함수의 출력 값을 가장 크게 줄이는 방향”입니다.

4.4.1 경사법(경사 하강법)

신경망은 학습 단계에서 최적(손실 함수가 최솟값일 때)의 매개변수를 찾아내야 합니다. 일반적인 문제의 손실 함수는 매우 복잡하고, 매개변수 공간이 광대하여 어디가 최솟값이 되는 곳인지 짐작하기 힘들기 때문에 기울기를 이용해 함수의 최솟값을 찾으려는 것이 경사법입니다. 여기서 주의해야할 것은 기울기가 각 지점에서 함수의 값을 낮추는 방안을 제시하는 것이기 때문에 그 방향이 정말 맞는 방향인지 보장할 수 없습니다. (기울기가 0이 된다고 함수의 최솟값이 아닐 수 있기 때문입니다.)

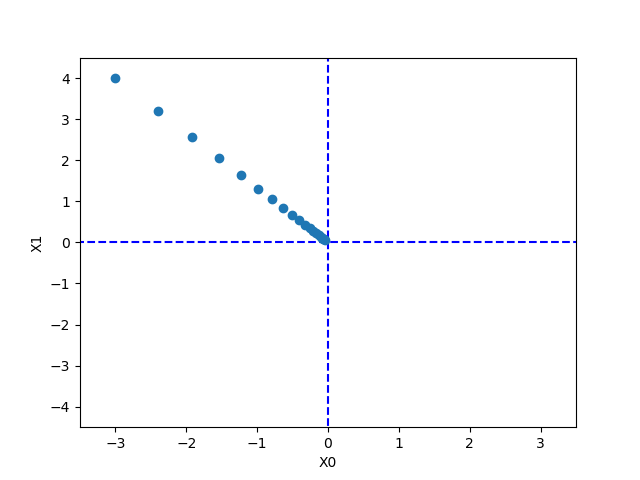
경사법은 현 위치에서 기울어진 방향으로 일정 거리만큼 이동하고, 다시 기울기를 구하고, 또 기울어진 방향으로 나아가는 것을 반복합니다. 이런 식으로 함수의 값을 점차 줄이는 것이 경사법(gradient method)입니다. 최솟값을 찾느냐, 최댓값을 찾느냐에 따라 전자는 경사 하강법(gradient descent method), 후자는 경사 상승법(gradient ascent method)라고 합니다. 일반적으로 딥러닝 분야는 ‘경사 하강법’을 사용할 때가 많습니다. 경사법을 수식으로 나타내면 다음과 같습니다.

[식 4.7]

기호(eta, 에타)는 갱신하는 양을 나타내고 학습률(learning rate)라고 합니다. 한 번의 학습으로 매개변수 값을 얼마나 갱신하는가를 정합니다. [식 4.7]은 1회에 해당하는 갱신이고, 이 단계를 반복합니다. 변수는 2개인 경우를 보여줬지만, 변수의 수가 늘어도 각 변수의 편미분 값(기울기)으로 갱신하게 됩니다. 학습률 값은 0.01이나 0.001 등 미리 특정 값으로 정해야 하고, 너무 크거나 작으면 ‘좋은 장소’를 찾아갈 수 없습니다. 경사 하강법을 간단하게 구현해보도록 하겠습니다.

def gradient\_descent(f, init\_x, lr=0.01, step\_num=100):  
 x = init\_x  
  
 for i in range(step\_num):  
 grad = numerical\_gradient(f, x)  
 x -= lr \* grad  
  
 return x

f는 최적화하려는 함수, init\_x는 초깃값(가중치), lr은 학습률, step\_num은 경사법에 따른 반복 횟수를 뜻합니다. 그래프 출력은 교재에서 제공한 gradient\_method.py 코드를 사용했습니다.



나쁜 학습률의 예입니다.

init\_x = np.array([-3.0, 4.0])  
print(gradient\_descent(function\_tmp2, init\_x, lr=10.0, step\_num=20))  
  
init\_x = np.array([-3.0, 4.0])  
print(gradient\_descent(function\_tmp2, init\_x, lr=1e-4, step\_num=20))



학습률이 너무 크면 (lr=10.0) 큰 값으로 발산하고, 학습률이 너무 작으면 (lr=1e-4) 초기값에서 큰 차이가 없습니다. 이런 학습률 같은 매개변수를 하이퍼파라미터(hyper parameter, 초매개변수)라고 합니다. 가중치와 편향같이 자동으로 학습되는 매개변수와 달리 여러 후보 중에서 시험을 통해 가장 잘 학습하는 값을 사람이 직접 설정해주어야 합니다.

4.4.2 신경망에서의 기울기

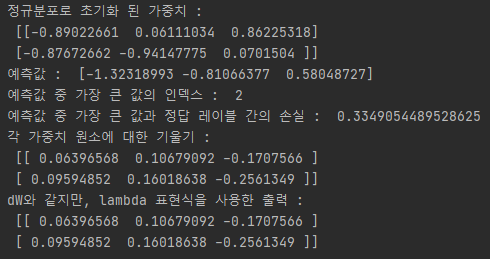
신경망에서의 기울기는 가중치 매개변수에 대한 손실 함수의 기울기를 말합니다. 예를 들어 2x3, 가중치가 W, 손실 함수가 L인 경우 경사는 로 나타내고 수식은 다음과 같습니다.

[식 4.8]

의 각 원소는 각각의 원소에 관한 편미분입니다. 예를 들어 1행 1번째 원소인 은 을 변경했을 때 손실 함수L의 변화량을 나타냅니다. 짚고 넘어갈 것은 W의 형태와 의 형태가 2x3으로 같다는 것입니다. 코드를 통해 살펴봅시다.

이때까지 배웠던 것을 바탕으로 간단한 신경망 구성을 했습니다. 더미 데이터이긴 하지만 x = [0.6, 0.9] 1x2행렬과 정규분포로 초기화된 2x3 형상의 W의 행렬 곱의 결과는 1x3 z로 나옵니다. z는 다시 softmax() 함수를 통해 0~1.0 사이의 실수(확률)의 형태로 y에 저장됩니다. 예측값 y와 정답 레이블 t 사이의 손실을 구하기 위해 CEE 손실 함수를 사용했습니다. 출력은 다음과 같습니다.

import sys, os  
sys.path.append(os.pardir)  
from common.functions import softmax, cross\_entropy\_error  
from common.gradient import numerical\_gradient  
import numpy as np  
  
  
class SimpleNet:  
 def \_\_init\_\_(self):  
 self.W = np.random.randn(2, 3) # 정규분포로 초기화  
  
 def predict(self, x):  
 return np.dot(x, self.W)  
  
 def loss(self, x, t):  
 z = self.predict(x)  
 y = softmax(z)  
 loss = cross\_entropy\_error(y, t)  
  
 return loss  
  
  
def f(W):  
 return net.loss(x, t)  
  
  
net = SimpleNet()  
print("정규분포로 초기화 된 가중치 : \n", net.W)  
  
x = np.array([0.6, 0.9])  
p = net.predict(x)  
print("예측값 : ", p)  
print("예측값 중 가장 큰 값의 인덱스 : ", np.argmax(p))  
  
t = np.array([0, 0, 1]) # 정답 레이블  
print("예측값 중 가장 큰 값과 정답 레이블 간의 손실 : ", net.loss(x, t))  
  
dW = numerical\_gradient(f, net.W)  
print("각 가중치 원소에 대한 기울기 : \n", dW)  
  
f1 = lambda w: net.loss(x, t)  
dW2 = numerical\_gradient(f1, net.W)  
print("dW와 같지만, lambda 표현식을 사용한 출력 : \n", dW2)



이제 가중치 W를 기울기를 통해 갱신하기만 하면 학습 과정을 구현하게 됩니다.

4.5 학습 알고리즘 구하기

구현할 신경망 학습의 절차는 다음과 같습니다.

전제

신경망에는 적응 가능한 가중치와 편향이 있고, 이 가중치와 편향을 훈련 데이터에 적응하도록 조정하는 과정을 ‘학습’이라 합니다. 신경망 학습은 다음과 같이 4단계로 수행합니다.

1단계 – 미니배치

훈련 데이터 중 일부를 무작위로 가져옵니다. 이렇게 선별한 데이터를 미니배치라 하며, 그 미니 배치의 손실 함수 값을 줄이는 것이 목표입니다.

2단계 – 기울기 산출

미니배치의 손실 함수 값을 줄이기 위해 각 가중치 매개변수의 기울기를 구합니다. 기울기는 손실 함수의 값을 가장 작게 하는 방향을 제시합니다.

3단계 – 매개변수 갱신

가중치 매개변수를 기울기 방향으로 아주 조금 갱신합니다.

4단계 – 반복

1~3단계를 반복합니다.

이 순서로 학습이 이루어질 것입니다. 경사 하강법으로 매개변수를 갱신하며, 이때 데이터를 미니배치로 무작위로 선정하기 때문에 확률적 경사 하강법(stochastic gradient descent, SGD)이라 부릅니다. MNIST 데이터셋을 사용하여 학습하는 신경망을 구현해보도록 하겠습니다! 우선 신경망을 하나의 클래스로 구현하는 것부터 시작하겠습니다. 필요한 함수와 메서드(활성화 함수, 손실 함수, 경사 하강 등은 책에서 제공하는 예제 코드를 사용했습니다.) numerical\_diff() 함수는 추후에 개선된 gradient() 함수를 작성해 대체한다고 합니다.

import sys, os  
sys.path.append(os.pardir)  
from common.functions import \*  
from common.gradient import numerical\_gradient  
  
  
class TwoLayerNet:  
 def \_\_init\_\_(self, input\_size, hidden\_size, output\_size, weight\_init\_std=0.01):  
 *"""* ***:param*** *input\_size: 입력층의 크기입니다. MNIST 데이터셋을 사용할 것이기 때문에 784가 될것입니다.* ***:param*** *hidden\_size: 은닉층의 크기입니다.* ***:param*** *output\_size: 출력층의 크기입니다.* ***:param*** *weight\_init\_std: 정규분포로 초기화 되는 가중치에 곱해줄 값입니다. 기본 0.01으로 설정합니다.  
 """* # 가중치를 저장해줄 dictionary 입니다.  
 self.params = {}  
 self.params['W1'] = weight\_init\_std \* \  
 np.random.randn(input\_size, hidden\_size) # key : W1(가중치), value : 정규분포로 초기화  
 self.params['b1'] = np.zeros(hidden\_size) # key : b1(편향), value : 0으로 초기화  
 self.params['W2'] = weight\_init\_std \* \  
 np.random.randn(hidden\_size, output\_size)  
 self.params['b2'] = np.zeros(output\_size)  
  
 # 추론 과정. 입력 데이터를 통해 신경망의 결과값을 출력한다.  
 def predict(self, x):  
 W1, W2 = self.params['W1'], self.params['W2']  
 b1, b2 = self.params['b1'], self.params['b2']  
  
 a1 = np.dot(x, W1) + b1  
 z1 = sigmoid(a1)  
 a2 = np.dot(z1, W2) + b2  
 y = softmax(a2)  
  
 return y  
  
 # x : 입력 데이터, t : 정답 레이블  
 def loss(self, x, t):  
 y = self.predict(x)  
 return cross\_entropy\_error(y, t)

def accuracy(self, x, t):  
 y = self.predict(x)  
 y = np.argmax(y, axis=1) # 예측값중 가장 큰값의 인덱스 추출  
 t = np.argmax(t, axis=1) # 원-핫 인코딩 된 정답레이블에서 '1'의 인덱스 추출  
  
 # 입력 데이터 중에 예측값의 인덱스와 정답 레이블의 인덱스가 같은 횟수를 정확도로 계산합니다.  
 accuracy = np.sum(y == t) / float(x.shape[0])  
 return accuracy  
  
 # x : 입력 데이터, t : 정답 레이블. 가중치 매개변수의 기울기를 구합니다.  
 def numerical\_gradient(self, x, t):  
 loss\_W = lambda W: self.loss(x, t)  
  
 # 기울기를 저장할 dictionary  
 grads = {}  
 grads['W1'] = numerical\_gradient(loss\_W, self.params['W1'])  
 grads['b1'] = numerical\_gradient(loss\_W, self.params['b1'])  
 grads['W2'] = numerical\_gradient(loss\_W, self.params['W2'])  
 grads['b2'] = numerical\_gradient(loss\_W, self.params['b2'])  
  
 return grads

4.5.2 미니배치 구현하기

TwolayerNet 클래스는 작성했지만, 아직 미니배치 학습과 매개변수 갱신을 구현하지 않았습니다. 이어서 구현해보도록 하겠습니다.

import numpy as np  
from dataset.mnist import load\_mnist  
from two\_layer\_net import TwoLayerNet  
  
  
(x\_train, t\_train), (x\_test, t\_test) = load\_mnist(normalize=True, one\_hot\_label=True)  
  
train\_loss\_list = []  
  
# 하이퍼파라미터  
iters\_name = 10000 # 반복 횟수  
train\_size = x\_train.shape[0] # 학습 데이터 크기  
batch\_size = 100 # 미니배치 크기  
learning\_rate = 0.1 # 학습률  
  
network = TwoLayerNet(input\_size=784, hidden\_size=50, output\_size=10)  
  
for i in range(iters\_name):  
 # 미니배치 획득  
 batch\_mask = np.random.choice(train\_size, batch\_size)  
 x\_batch = x\_train[batch\_mask]  
 t\_batch = t\_train[batch\_mask]  
  
 # 기울기 계산  
 grad = network.numerical\_gradient(x\_batch, t\_batch)  
 # 개선판!  
 # grad = network.gradient(x\_batch, t\_batch)  
  
 # 매개변수 갱신  
 for key in ('W1', 'b1', 'W2', 'b2'):  
 network.params[key] -= learning\_rate \* grad[key]  
  
 loss = network.loss(x\_batch, t\_batch)  
 train\_loss\_list.append(loss)

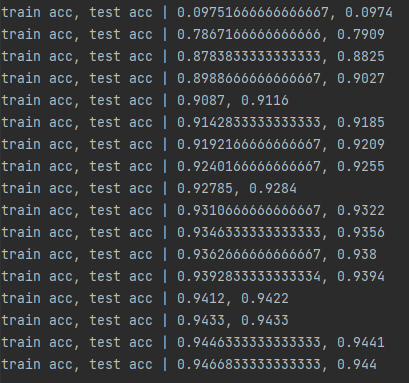
미니배치 크기는 100입니다. 매번 60000개의 데이터에서 100개를 무작위로 추려내 그 100개의 미니배치를 대상으로 SGD를 수행해 매개변수를 갱신합니다. 갱신 횟수(iters\_num)은 10000번입니다. 이제 훈련 과정은 끝이고 오버피팅이 일어나지는 않았는지 시험 데이터로 검증을 할 차례입니다. 이를 위해서는 학습 도중 정기적으로 훈련 데이터와 시험 데이터를 대상으로 정확도를 기록합니다. 여기서 사용할 에폭(epoch)은 하나의 단위로 1회의 학습에서 훈련 데이터를 모두 소진했을 때의 횟수입니다. 예를 들어 10000개의 데이터를 100개의 미니배치로 훈련한다 했을 때, 1에폭은 100개의 미니배치로 100회 학습할 때까지 입니다.

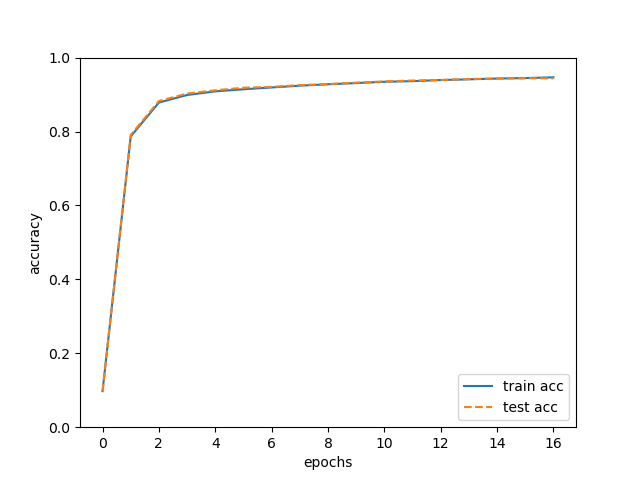
4.5.3 시험 데이터로 평가하기

import numpy as np  
from dataset.mnist import load\_mnist  
from two\_layer\_net import TwoLayerNet  
import matplotlib.pyplot as plt  
  
(x\_train, t\_train), (x\_test, t\_test) = load\_mnist(normalize=True, one\_hot\_label=True)  
  
network = TwoLayerNet(input\_size=784, hidden\_size=50, output\_size=10)  
  
# 하이퍼파라미터  
iters\_name = 10000 # 반복 횟수  
train\_size = x\_train.shape[0] # 학습 데이터 크기  
batch\_size = 100 # 미니배치 크기  
learning\_rate = 0.1 # 학습률  
  
train\_loss\_list = []  
train\_acc\_list = []  
test\_acc\_list = []  
  
iter\_per\_epoch = max(train\_size / batch\_size, 1)  
  
for i in range(iters\_name):  
 # 미니배치 획득  
 batch\_mask = np.random.choice(train\_size, batch\_size)  
 x\_batch = x\_train[batch\_mask]  
 t\_batch = t\_train[batch\_mask]  
  
 # 기울기 계산  
 grad = network.numerical\_gradient(x\_batch, t\_batch)  
 # 개선판!  
 # grad = network.gradient(x\_batch, t\_batch)  
  
 # 매개변수 갱신  
 for key in ('W1', 'b1', 'W2', 'b2'):  
 network.params[key] -= learning\_rate \* grad[key]  
  
 loss = network.loss(x\_batch, t\_batch)  
 train\_loss\_list.append(loss)  
  
 # 1epoch당 정확도 계산  
 if i % iter\_per\_epoch == 0:  
 train\_acc = network.accuracy(x\_train, t\_train)  
 test\_acc = network.accuracy(x\_test, t\_test)  
 train\_acc\_list.append(train\_acc)  
 test\_acc\_list.append(test\_acc)  
 print("train acc, test acc | " + str(train\_acc) + ", " + str(test\_acc))  
  
# 그래프 그리기  
markers = {'train': 'o', 'test': 's'}  
x = np.arange(len(train\_acc\_list))  
plt.plot(x, train\_acc\_list, label='train acc')  
plt.plot(x, test\_acc\_list, label='test acc', linestyle='--')  
plt.xlabel("epochs")  
plt.ylabel("accuracy")  
plt.ylim(0, 1.0)  
plt.legend(loc='lower right')  
plt.show()

원래의 코드에서 정확도 계산 부분과 그래프 출력 부분을 추가했습니다. 출력은 다음과 같습니다. 기울기 계산 과정이 너무 느려 출력 결과는 기울기 계산 개선판 코드를 사용했습니다.

def gradient(self, x, t):  
 W1, W2 = self.params['W1'], self.params['W2']  
 b1, b2 = self.params['b1'], self.params['b2']  
 grads = {}  
   
 batch\_num = x.shape[0]  
   
 # forward  
 a1 = np.dot(x, W1) + b1  
 z1 = sigmoid(a1)  
 a2 = np.dot(z1, W2) + b2  
 y = softmax(a2)  
   
 # backward  
 dy = (y - t) / batch\_num  
 grads['W2'] = np.dot(z1.T, dy)  
 grads['b2'] = np.sum(dy, axis=0)  
   
 da1 = np.dot(dy, W2.T)  
 dz1 = sigmoid\_grad(a1) \* da1  
 grads['W1'] = np.dot(x.T, dz1)  
 grads['b1'] = np.sum(dz1, axis=0)  
  
 return grads





실선은 훈련 데이터에 대한 정확도, 실선은 시험 데이터에 대한 정확도를 나타냅니다. 두 경우 모두 정확도가 높게 나오기 때문에 이 학습은 오버피팅이 일어나지 않았다고 볼 수 있습니다.