Ch 05 오차역전파법

앞 장에서 신경망 학습을 어떻게 설계하고 만드는지 알아보았습니다. 하지만 수치미분(numerical\_diff)를 사용했을 때 신경망이 훈련하는 속도가 매우 느렸었습니다. ‘오차역전파법(backpropagation)을 통해 가중치 매개변수의 기울기를 효율적으로 계산하는 법에 대해 알아봅시다. 이 책에서 오차역전파법을 이해하기 위해 수식과 계산 그래프를 사용합니다. 계산 그래프는 이해하기에 편하지만 모두 그리기에 시간이 많이 걸리기 때문에 수식으로 정리해보려 합니다.

5.1 계산그래프

계산 그래프(computational graph)는 여러 노드(node)와 에지(edge)로 표현되는 그래프 자료구조입니다. 간단한 예를 보여드리고 그래프는 넘어가겠습니다.

5.1.1 계산 그래프로 풀다

문제 1 : 현빈 군은 슈퍼에서 1개에 100원인 사과를 2개 샀습니다. 이때 지불 금액을 구하세요. 단, 소비세가 10% 부과됩니다.

이 문제를 계산 그래프로 표현하면 다음과 같이 그려집니다.

100

200

220

2개

1.1

그림처럼 계산을 왼쪽에서 오른쪽으로 진행하는 단계를 순전파(forward propagation)라고 합니다. 역전파(backward propagation)은 말 그대로 오른쪽에서 왼쪽으로 전파되는 것을 말하고, 이는 이후에 미분을 계산할 때 중요한 역할을 합니다.

5.1.2 국소적 계산

계산 그래프의 특징은 ‘국소적 계산’을 전파하여 최종 결과를 얻는 것입니다. 전체 계산이 아무리 복잡해도 각 단계에서 하는 일은 해당 노드의 ‘국소적 계산’이기 때문에 계산 자체는 단순하지만, 그 결과를 전달함으로써 전체의 복잡한 계산의 결과를 얻어낼 수 있습니다.

5.1.3 왜 계산 그래프로 푸는가?

계산 그래프는 ‘국소적 계산’을 통해 각 노드의 계산에 집중하여 문제를 단순화할 수 있는 이점이 있습니다. 그것 외에도 중간 계산 결과를 모두 저장할 수 있는 이점도 있습니다. 가중치 매개변수는 손실함수의 결과에 따라 계속해서 변해야 하는 변수이기 때문에 중간의 계산 결과들을 저장할 수 있다는 이점은 중요합니다. 그래도 계산 그래프를 사용하는 가장 큰 이유는 역전파를 통해 ‘미분’을 효율적으로 계산할 수 있는 점에 있습니다. 위의 예를 다시 꺼내 설명해보겠습니다. ‘사과 가격의 인상이 최종 금액에 끼치는 영향’을 알고 싶다고 했을 때, 이는 ‘사과 가격에 대한 지불 금액의 미분’을 구하는 문제입니다. 사과 값을 x, 지불 금액을 L이라 했을 때 을 구하는 것입니다.

5.2 연쇄법칙

역전파는 ‘국소적인 미분’을 연쇄법칙(chain rule)에 따라 순전파와 반대방향으로 전파합니다.

5.2.1 계산 그래프의 역전파

E

E

y

x

위 그림은 y = f(x)라는 계산의 역전파입니다. 계산 절차는 다음과 같습니다. 신호 E에 국소적 미분인 을 곱한 후 다음 노드로 전달하는 것입니다. 이런 방식으로 목표로 하는 미분 값을 효율적으로 구할 수 있다는 것이 역전파의 핵심입니다. 이를 가능하게 하는 연쇄법칙에 대해 알아봅시다.

5.2.2 연쇄법칙이란?

연쇄법칙을 얘기하기 위해서는 합성 함수부터 알아야 합니다. 합성 함수는 여러 함수로 구성된 함수로 예를 들어 z = (x + y)^2 라는 식은 z = t^2, t = x + y 라는 두개의 식으로 구성됩니다. 연쇄법칙은 이런 합성함수의 미분에 대한 성질이며 다음과 같이 정의됩니다.

*합성 함수의 미분은 합성 함수를 구성하는 각 함수의 미분의 곱으로 나타낼 수 있다.*

간단히 설명하자면 (z에 대한 x의 미분)은 (z에 대한 t의 미분)과 (x에 대한 t의 미분)의 곱으로 나타낼 수 있다는 것입니다. 수식으로는 다음과 같습니다.

아까 예를 들었던 식에 대한 를 구해보도록 합시다.

[식 5.1]

5.2.3 연쇄법칙과 계산 그래프

[식 5.1]의 연쇄법칙 계산을 계산 그래프로 나타내봅시다.

x

z

t

y

위 그림에서 역전파를 진행하면 오른쪽에서 왼쪽으로 들어오는 신호에 그 노드의 국소적 미분(편미분)을 곱해 다음 노드로 넘기는 것을 볼 수 있습니다. 마지막 전파를 봤을 때 이 계산은 결국 ‘x에 대한 z의 미분’이 됩니다. 즉, 역전파가 하는 일은 연쇄 법칙의 원리와 같다는 것을 알 수 있습니다.

5.3 역전파

5.3.1 덧셈 노드의 역전파

[식 5.2]

[식 5.2]와 같이 해석적으로 덧셈에서의 미분을 구할 수 있습니다. 이를 계산 그래프로 그려보겠습니다.

계산 그래프를 그려보았을 때 덧셈 노드의 역전파는 상류의 흐름을 그대로 넘겨줍니다.

5.3.2 곱셈 노드의 역전파

[식 5.3]

곱셈 식과 계산그래프는 위와 같습니다. 계산 그래프를 보면 역전파 과정에서 상류 값에 순전파 때의 입력 신호들을 서로 바꿔 곱해서 전파합니다. 계산 그래프 상의 역전파를 통해 ‘국소적 미분’을 쉽게 구하는 방법을 알아보았습니다. 5.3.3은 앞에서 잠시 예를 들었던 사과 쇼핑에 대한 문제를 다시금 역전파로 풀어보는 것으로 생략하겠습니다.

5.4 단순한 계층 구현하기

앞에서 예를 들었던 ‘사과 쇼핑’예를 파이썬으로 구현해봅시다. 곱셈 노드를 ‘MulLayer’, 덧셈 노드를 ‘AddLayer’로 구현하겠습니다.

5.4.1 곱셈 계층

모든 계층은 순전파 forward(), 역전파 backward()라는 공통의 메서드(인터페이스)를 갖도록 구현하겠습니다.

class MulLayer:  
 def \_\_init\_\_(self):  
 self.x = None  
 self.y = None  
  
 def forward(self, x, y):  
 self.x = x  
 self.y = y  
 out = x \* y  
 return out  
  
 def backward(self, dout):  
 # dout은 상류의 흐름을 말합니다.  
 dx = dout \* self.y  
 dy = dout \* self.x # 서로의 순전파 때의 값을 바꿔 곱합니다.  
  
 return dx, dy

MulLayer를 이용하여 ‘사과 쇼핑’의 순전파를 구현할 수 있습니다.

from layer\_naive import MulLayer  
  
  
apple = 100  
apple\_num = 2  
tax = 1.1  
  
mul\_apple\_layer = MulLayer()  
mul\_tax\_layer = MulLayer()  
  
apple\_price = mul\_apple\_layer.forward(apple, apple\_num)  
price = mul\_tax\_layer.forward(apple\_price, tax)  
  
print("Apple total price : ", int(price))



5.4.2 덧셈 계층

이어서 덧셈 계층을 구현해보도록 하겠습니다.

class AddLayer:  
 # 곱셈 계층처럼 이전의 순전파 값을 기억할 필요가 없습니다.  
 def \_\_init\_\_(self):  
 pass  
  
 def forward(self, x, y):  
 out = x + y  
 return out  
  
 def backward(self, dout):  
 dx = dout \* 1  
 dy = dout \* 1  
 return dx, dy

이제 덧셈 계층과 곱셈 계층을 이용하여 ‘귤과 사과 쇼핑’예를 구현하겠습니다. 사과를 2개 귤을 3개 삽니다. 귤의 가격은 150입니다.

from layer\_naive import \*  
  
  
apple = 100  
orange = 150  
apple\_num = 2  
orange\_num = 3  
tax = 1.1  
  
mul\_apple\_layer = MulLayer()  
mul\_orange\_layer = MulLayer()  
add\_apple\_orange\_layer = AddLayer()  
mul\_tax\_layer = MulLayer()  
  
# 순전파  
apple\_price = mul\_apple\_layer.forward(apple, apple\_num)  
orange\_price = mul\_orange\_layer.forward(orange, orange\_num)  
apple\_orange\_price = add\_apple\_orange\_layer.forward(apple\_price, orange\_price)  
total\_price = mul\_tax\_layer.forward(apple\_orange\_price, tax)  
  
#역전파  
dprice = 1  
dtotal\_price, dtax = mul\_tax\_layer.backward(dprice)  
dapple\_price, dorange\_price = add\_apple\_orange\_layer.backward(dtotal\_price)  
dorange, dorange\_num = mul\_orange\_layer.backward(dorange\_price)  
dapple, dapple\_num = mul\_apple\_layer.backward(dapple\_price)  
  
print("Total price : ", int(total\_price))  
print(dapple\_num, dapple, dorange, dorange\_num, dtax)



계산 그래프 계층을 구현함으로써 미분을 쉽게 구하는 방법에 대해 알아보았습니다.

5.5 활성화 함수 계층 구현하기

이제 계산 그래프의 계층을 신경망에 적용할 때가 왔습니다. 신경망의 계층을 각각의 클래스 하나로 구현해보도록 하겠습니다. 우선은

5.5.1 ReLU 계층

활성화 함수 ReLU의 수식과 x에 대한 y의 미분은 다음과 같습니다.

이제 구현해보도록 합시다. 모든 클래스(계층)의 forward()와 backward()는 넘파이 배열을 인수로 받는다고 가정합시다.

class Relu:  
 def \_\_init\_\_(self):  
 # x <= 0인 원소의 index에 맞춰 True False 값을 저장하는 numpy 배열입니다.  
 # 작으면 True, 크면 False를 저장합니다.  
 self.mask = None  
  
 def forward(self, x):  
 self.mask = (x <= 0)  
 out = x.copy()  
 # 0보다 작은 원소들을 0으로 맞춰줍니다.  
 out[self.mask] = 0  
  
 return out  
  
 def backward(self, dout):  
 # 순전파 때와 마찬가지로 0으로 맞춰줍니다.  
 dout[self.mask] = 0  
 dx = dout  
  
 return dx

5.5.2 sigmoid 계층

시그모이드 함수의 순전파와 역전파는 다음과 같이 이루어집니다.

간소화시키면 다음과 같이 표현할 수 있습니다.

그리고 역전파의 결과인 는 다음처럼 정리할 수 있습니다.

정리한 결과를 보면 Sigmoid 계층의 역전파는 순전파의 출력(y)만으로 계산할 수 있습니다. 이를 파이썬으로 구현해보겠습니다.

class Sigmoid:  
 def \_\_init\_\_(self):  
 self.out = None # 역전파를 구하기 위해 순전파의 출력을 저장할 변수  
  
 def forward(self, x):  
 out = 1 / (1 + np.exp(-x))  
 self.out = out  
  
 return out  
  
 def backward(self, dout):  
 dx = dout \* (1.0 - self.out) \* self.out  
  
 return dx

순전파의 출력을 인스턴스 변수 out에 저장했다가, 역전파에 그 값을 사용하여 쉽게 구현할 수 있었습니다.

5.6 Affine/Softmax 계층 구현하기

5.6.1 Affine 계층

신경망에서 순전파 때 수행하는 행렬의 곱을 기하학에서 어파인 변환(Affine transformation)이라 합니다. 구현할 때 주의해야할 점은 행렬의 곱이기 때문에 각 변수의 형상에 주의해야 합니다. 계산 그래프로 표현하면 다음과 같이 그려집니다.

(3,)

(3,)

Y

(3,)

X ∙ W

(3,)

(3,)

B

(3,)

1. :
2. :

(2,)

X

(2,3)

W

①

②

5.6.2 배치용 Affine 계층

지금까지 얘기한 Affine 계층은 데이터 하나만을 생각한 것입니다. 데이터 N개를 묶어 순전파하는 배치용 Affine 계층을 생각해보겠습니다. 위의 예에서 입력데이터 X가 (N, 2)로 바뀐다 생각하여 계산 그래프의 순서를 따라 계산을 하면, Y가 (N,3)가 된다는 것을 알 수 있습니다.

(N, 2) ∙ (2, 3) + (3, ) = (N, 3)

편향에 대해선 주의해야 합니다. 데이터의 개수가 2개인 경우를 예를 들어 편향은 두 데이터 각각에 더해지기 때문에 역전파 때는 각 데이터의 역전파 값이 편향의 원소에 모여야 합니다. Affine 계층의 구현은 다음과 같습니다. 책에서 입력데이터가 텐서(4차원 데이터)인 경우도 고려했기 때문에 코드에 조금 차이가 있습니다.

class Affine:  
 def \_\_init\_\_(self, W, b):  
 self.W = W  
 self.b = b  
  
 self.x = None  
 self.original\_x\_shape = None  
 self.dW = None  
 self.db = None  
  
 def forward(self, x):  
 # 텐서 대응  
 self.original\_x\_shape = x.shape  
 x = x.reshape(x.shape[0], -1)  
 self.x = x  
  
 out = np.dot(x, self.W) + self.b  
  
 return out  
  
 def backward(self, dout):  
 dx = np.dot(dout, self.W.T)  
 self.dW = np.dot(self.x.T, dout)  
 self.db = np.sum(dout, axis=0)  
  
 dx = dx.reshape(\*self.original\_x\_shape) # 입력 데이터의 모양 변경(텐서 대응)  
 return dx

5.6.3 Softmax-with-Loss 계층

출력층에서 사용하는 소프트맥스(Softmax) 계층에 대해 알아보겠습니다. 소프트맥스 계층은 입력 값을 정규화(출력의 합이 1이 되도록)하여 출력합니다. 소프트맥스와 교차 엔트로피 오차(CEE)를 같이 사용하여 Softmax-with-Loss 계층을 만들어보도록 합시다. Softmax 계층은 입력 (a1, a2, a3)를 정규화하여 (y1, y2, y3)를 출력하고, Cross Entropy Error 계층은 Softmax 계층의 출력 (y1, y2, y3)와 정답 레이블 (t1, t2, t3)를 받아 손실L을 출력합니다.

이 계층의 역전파의 결과에 주목해야합니다. Softmax 계층의 역전파는 (y1 – t1, y2 – t2, y3 – t3)라는 결과를 내놓는데 이는 Softmax 계층의 출력과 정답 레이블의 차분입니다. 역전파에서 이 오차가 앞 계층에 전해지며 이는 신경망 학습의 중요한 성질입니다. 신경망 학습의 목적이 신경망의 출력이 정답 레이블에 가까워지도록 가중치 매개변수를 조정하는 것인데, 역전파의 결과는 이를 그대로 드러냅니다. 그럼 Softmax-with-Loss 계층을 구현해보도록 합시다.

class SoftmaxWithLoss:  
 def \_\_init\_\_(self):  
 self.loss = None # 손실  
 self.y = None # softmax의 출력  
 self.t = None # 정답 레이블  
  
 def forward(self, x, t):  
 self.t = t  
 self.y = softmax(x)  
 self.loss = cross\_entropy\_error(self.y, self.t)  
 return self.loss  
  
 def backward(self, dout=1):  
 batch\_size = self.t.shape[0]  
 dx = (self.y - self.t) / batch\_size # 데이터 1개당 오차  
   
 return dx

앞에서 만들었던 softmax와 cross\_entropy\_error를 활용합니다.

5.7 오차역전파법 구현하기

앞에서 구현한 계층들을 조립하면 신경망을 구축할 수 있습니다. 이번 절에서 한번 해보도록 하겠습니다.

5.7.1 신경망 학습의 전체 그림

전제

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

1단계 – 미니배치

훈련 데이터 중 일부를 무작위로 가져옵니다. 이렇게 선별한 데이터를 미니배치라 하면, 그 미니배치의 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.

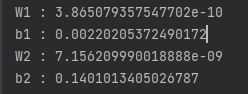
저번에 만든 신경망에서는 수치 미분을 사용해 계산이 매우 오래 걸렸지만 이번에는 오차역전파법을 통해 기울기 계산을 효율적으로 해봅시다.

import sys, os  
sys.path.append(os.pardir)  
import numpy as np  
from common\_trial.layers\_trial import \*  
from common\_trial.gradient\_trial import numerical\_gradient  
from collections import OrderedDict  
  
  
class TwoLayerNet:  
 def \_\_init\_\_(self, input\_size, hidden\_size, output\_size, weight\_init\_std=0.01):  
 # 가중치 초기화  
 self.params = {}  
 self.params['W1'] = weight\_init\_std \* np.random.randn(input\_size, hidden\_size)  
 self.params['b1'] = np.zeros\_like(hidden\_size)  
 self.params['W2'] = weight\_init\_std \* np.random.randn(hidden\_size, output\_size)  
 self.params['b2'] = np.zeros\_like(output\_size)  
  
 # 계층 생성  
 self.layers = OrderedDict()  
 self.layers['Affine1'] = Affine(self.params['W1'], self.params['b1'])  
 self.layers['Relu1'] = Relu()  
 self.layers['Affine2'] = Affine(self.params['W2'], self.params['b2'])  
  
 self.lastLayer = SoftmaxWithLoss()  
  
 def predict(self, x):  
 for layer in self.layers.values():  
 x = layer.forward(x)  
  
 return x  
  
 def loss(self, x, t):  
 y = self.predict(x)  
 return self.lastLayer.forward(y, t)  
  
 def accuracy(self, x, t):  
 y = self.predict(x)  
 y = np.argmax(y, axis=1)  
 if t.ndim != 1 : t = np.argmax(t, axis=1)  
  
 accuracy = np.sum(y==t) / float(x.shape[0])  
 return accuracy  
  
 def numerical\_gradient(self, x, t):  
 loss\_W = lambda W: self.loss(x, t)  
  
 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  
  
 def gradient(self, x, t):  
 self.loss(x, t)  
  
 dout = 1  
 dout = self.lastLayer.backward(dout)  
  
 layers = list(self.layers.values())  
 layers.reverse()  
 for layer in layers:  
 layer.backward(dout)  
  
 grads = {}  
 grads['W1'] = self.layers['Affine1'].dW  
 grads['b1'] = self.layers['Affine1'].db  
 grads['W2'] = self.layers['Affine2'].dW  
 grads['b2'] = self.layers['Affine2'].db  
  
 return grads

5.7.3 오차역전파법으로 구한 기울기 검증하기

오차역전파법은 구현하기 어렵기 때문에 종종 실수를 할 수 있습니다. 그 때문에 구현이 쉬운 수치 미분과 결과를 비교하는 기울기 확인(gradient check)라는 작업을 해 오차역전파법을 제대로 구현했는지 검증하곤 합니다. 구현 및 결과는 다음과 같습니다.

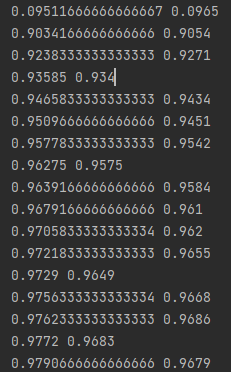
import sys, os  
sys.path.append(os.pardir)  
import numpy as np  
from dataset.mnist import load\_mnist  
from two\_layer\_net import TwoLayerNet  
  
  
(x\_train, t\_train), (x\_test, t\_test) = load\_mnist(normalize=True, one\_hot\_label=True)  
  
network = TwoLayerNet(input\_size=784, hidden\_size=50, output\_size=10)  
  
x\_batch = x\_train[:3]  
t\_batch = t\_train[:3]  
  
grad\_numerical = network.numerical\_gradient(x\_batch, t\_batch)  
grad\_backprop = network.gradient(x\_batch, t\_batch)  
  
for key in grad\_numerical.keys():  
 diff = np.average(np.abs(grad\_backprop[key] - grad\_numerical[key]))  
 print(key + " : " + str(diff))



결과를 보면 차이가 매우 작기 때문에 오차역전파법으로 구한 기울기에 신뢰성을 더할 수 있습니다.

5.7.4 오차역전파법을 사용한 학습 구현하기

import sys, os  
sys.path.append(os.pardir)  
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)  
  
network = TwoLayerNet(input\_size=784, hidden\_size=50, output\_size=10)  
  
iters\_num = 10000  
train\_size = x\_train.shape[0]  
batch\_size = 100  
learning\_rate = 0.1  
  
train\_loss\_list = []  
train\_acc\_list = []  
test\_acc\_list = []  
  
iter\_per\_epoch = max(train\_size / batch\_size, 1)  
  
for i in range(iters\_num):  
 batch\_mask = np.random.choice(train\_size, batch\_size)  
 x\_batch = x\_train[batch\_mask]  
 t\_batch = t\_train[batch\_mask]  
  
 grad = network.gradient(x\_batch, t\_batch)  
  
 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)  
  
 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)



이로써 계산 그래프를 통해 신경망의 동작과 오차역전파법을 설명하고, 계층이라는 단위로 구현했습니다. 모든 계층에서 forward, backward라는 메서드를 구현해 가중치 매개변수의 기울기를 효율적으로 구할 수 있었습니다. 동작을 계층으로 모듈화 한 덕분에 신경망의 계층을 자유롭게 조합하여 신경망을 만들 수 있었습니다.