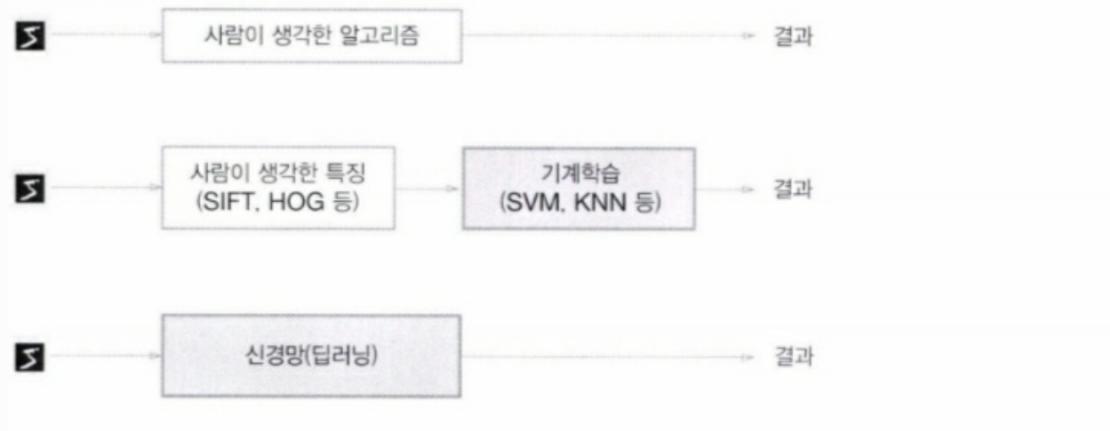


학습

- 훈련 데이터로부터 가중치 매개변수의 최적값을 자동으로 획득하는 것
- 손실함수의 결과값을 가장 작게 만드는 가중치 매개변수를 찾는 것이 학습의 목표

그림 4-2 규칙을 '사람'이 만드는 방식에서 '기계'가 데이터로부터 배우는 방식으로의 패러다임 전환 : 회색 블록은 사람이 개입하지 않음을 뜻한다.



신경망은 이미지를 있는 그대로 학습

이미지에 포함된 중요한 특징까지 기계가 스스로 학습

-> 딥러닝을 end-to-end machine learning이라고 함. 입력에서 출력을 사람의 개입 없이 얻는다는 뜻

Train set 과 Test set을 분리하는 이유

- 범용 능력을 제대로 평가하기 위해 나눈다!
- 범용 능력이란 아직 보지 못한 데이터로도 문제를 올바르게 풀어낼 수 있는 능력(Overfitting 방지)

손실함수(Loss Function)

'하나의 지표'를 기준으로 최적의 매개변수 값을 탐색

→ Loss function

일반적으로 오차제곱합과 CrossEntropy Function 사용

오차 제곱합(Sum of squares for error - SSE)

$$E = \frac{1}{2} \sum_k (y_k - t_k)^2$$

y_k : 신경망의 출력, t_k : 정답레이블 k : 데이터의 차원 수

>>> y = [0.1, 0.05, 0.6, 0.0, 0.05, 0.1, 0.0, 0.1, 0.0, 0.0] \Rightarrow 0~10 salt max output
>>> t = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0] \Rightarrow label (정답)

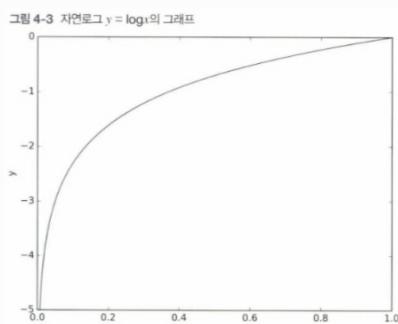
t 처럼 한 원소만 1로 하고 그 외는 0이라 표시하는 표기법은 **One Hot Encoding** 이라 함

CrossEntropy

$$E = - \sum_k t_k \log y_k \rightarrow \log = \ln = \log_e$$

$t_k \Rightarrow$ One Hot Encoding 형태의 정답값

정답인 때의 출력이 전체 값을 정하게 된다.



$$\Rightarrow x=1 \Rightarrow y=0, x=0 \Rightarrow y \downarrow$$

\therefore 정답에 해당하는 출력이 커질수록 0에 가까워지고, 출력이 1일 때 0이 된다.

미니 배치 학습

- 훈련 데이터에 대한 손실 함수의 값을 구하고 그 값을 최대한 줄여주는 매개변수를 찾아낸다. \rightarrow 모든 훈련 데이터를 대상으로 손실 함수 값을 구해야 됨.

$$E = -\frac{1}{N} \sum_n \sum_k t_{nk} \log y_{nk}$$

→ 모든 데이터에 대한
Cross Entropy 페널티

N : 데이터 개수, t_{nk} : n 번째 데이터의 k 번째 값(정답), y_{nk} : n 번째 데이터의 k 번째 값
(신경망의 출력)

N 을 나눈 이유 : N 을 나누어 정규화 \Rightarrow 평균 손실함수 도출

모든 데이터를 대상으로 손실 함수의 합을 구하기엔 시간이 오래 걸림.

- > 이를 극복하기 위해 데이터의 일부를 추려 근사치로 이용
- > 이 일부를 미니배치(mini batch)라고 함
- > ex. 데이터 6만장 중 100장만 추려 학습 = 미니배치 학습

왜 정확도가 아닌 손실함수를 설정하는가?

- '미분'의 역할에 주목, 최적의 매개변수 값을 탐색할 때 손실함수의 값을 가능한 한 작게 하는 매개변수 값을 탐색. 매개변수의 미분(기울기)를 계산하고 미분 값을 단서로 매개변수의 값을 서서히 갱신하는 과정 반복
- 손실함수의 미분 = 가중치 매개변수의 값을 아주 조금 변화시켰을 때, 손실 함수가 어떻게 변하나의 의미
- > if 미분값이 음수이면 가중치 매개변수를 양의 방향으로 변화시켜 손실 함수의 값을 줄일 수 있다.
- elif 미분값이 양수이면 가중치 매개변수를 음의 방향으로 변화시켜 손실 함수의 값을 줄일 수 있다.
- else 미분값이 0이면 가중치 매개변수를 어느쪽으로 움직여도 손실 함수의 값은 줄어들지 않는다. \rightarrow 매개변수의 갱신이 멈춤

따라서 신경망을 학습할 때 정확도를 지표로 삼아서는 안되는 이유는 매개 변수의 미분이 대부분의 장소에서 0이 되기 때문이다!

↳ why?? ↴
다음 장

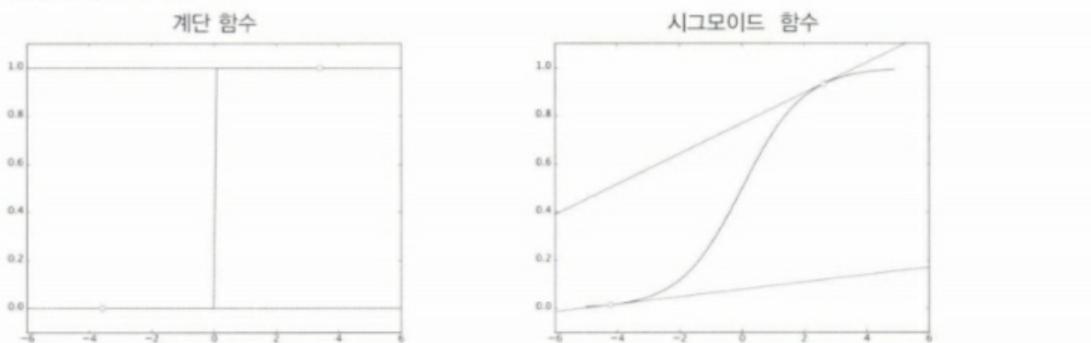
ex. 정확도가 지표였을 때 100장 중 32장을 바로 인식했을 때 정확도 32% 매개변수의 값을 약간만 조정했을 때에도 정확도가 개선되지 않고 일정하게 유지.(정확도 = 맞은 개수/전체 개수) <- 이 식을 생각한다면 이해가 됨) 만약 바뀐다 해도 32.054 같은 연속적인 값이 아니라 33,34와 같은 불연속적인 값으로 바뀜

반면 손실함수일 때는? 손실함수가 0.92543이라는 수치로 나오는데 이를 매개변수가 약간씩 변할 땐 이 값도 이에 맞춰 0.93432와 같은 연속적인 수치로 변화함!

따라서 정확도는 매개변수의 사소한 변화에는 거의 반응을 보이지 않고 있어도 불연속적!

이는 계단함수를 활성화 함수로 삼지 않은 이유와 동일하다.

그림 4-4 계단 함수와 시그모이드 함수 : 계단 함수는 대부분의 장소에서 기울기가 0이지만, 시그모이드 함수의 기울기(접선)는 0이 아니다.



계단함수를 미분하였을 때 대부분의 장소에서 0이기 때문!

그러나 예를 들어 시그모이드 함수는 미분했을 때 곡선의 기울기도 연속적으로 변하기 때문에 괜찮다.(어느 장소에서 미분을 했을 때 0이 안됨)

미분

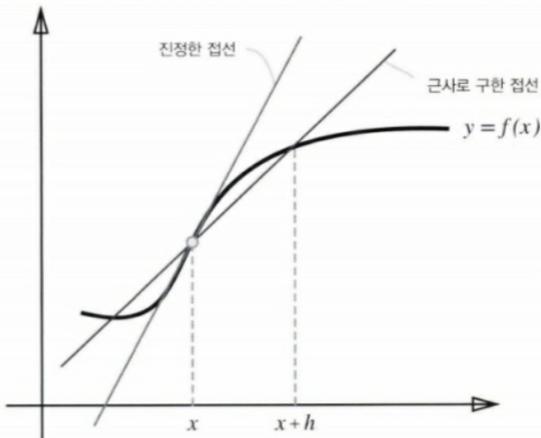
한순간의 변화량을 표시

$$\frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

→ Python으로 구현 시 $h = 10^{-4}$ 로 설정

미분을 코드로 구현했을 때 문제점 : 차분(임의의 두 점에서 함수 값들의 차이)

그림 4-5 진정한 미분(진정한 접선)과 수치 미분(근사로 구한 접선)의 값은 다르다.



수치 미분에는 오차가 포함. 이 오차를 줄이기 위해 f 의 차분을 계산
이러한 차분을 중심 차분 or 중앙 차분이라고 한다.

수치미분 예) $y = 0.01x^2 + 0.1x \Rightarrow 2 \times 0.01x + 0.1$

편미분

$$\textcircled{1} \quad f(x_0, x_1) = x_0^2 + x_1^2 \quad (x_0=3, x_1=4) \quad \frac{\partial f}{\partial x_0} ? \\ \Rightarrow \frac{\partial f}{\partial x_0} = 2x_0 = 6$$

$$\textcircled{2} \quad f(x_0, x_1) = x_0^2 + x_1^2 \quad (x_0=3, x_1=4) \quad \frac{\partial f}{\partial x_1} ?$$

$$\Rightarrow \frac{\partial f}{\partial x_1} = 2x_1 = 8$$

기울기

모든 변수의 편미분을 벡터로 정리한 것 $\left(\frac{\partial f}{\partial x_0}, \frac{\partial f}{\partial x_1} \right)$

기울기가 가리키는 쪽은 각 장소에서 함수의 출력 값을 가장 크게 줄이는 방법

경사하강법

기울기를 이용해 함수의 최솟값(손실함수의 최솟값)을 찾으려는 방법

-> 주의점 : 함수가 극솟값 최솟값 안정점이 되는 곳에선 기울기 0

기울기가 0인 곳을 찾으려 하지만 그 값이 최솟값이라는 보장 X

경사하강법의 flow : 현 위치에서 기울어진 방향으로 일정거리만큼 이동

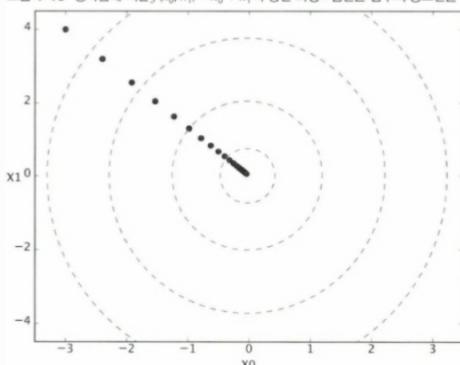
-> 다음 이동한 곳에서도 기울기를 구하고 그 방향으로 나아가는 것 반복

-> 이러한 방법으로 함수의 값을 줄이고 최적화

$$\begin{aligned} x_0 &= x_0 - \eta \frac{\partial f}{\partial x_0} \\ x_1 &= x_1 - \eta \frac{\partial f}{\partial x_1} \end{aligned}$$

η (에타) : 경신하는 양 (학습률) \rightarrow 한번의 학습으로 얼마나 많이 학습하는지, 매개변수 값을 얼마나 갱신하는지를 정함

그림 4-10 경사법에 의한 $f(x_0, x_1) = x_0^2 + x_1^2$ 의 경신 과정 : 점선은 함수의 등고선을 나타낸다.



하이퍼파라미터 : 학습률과 같은 직접 설정해야하는 매개변수

<-> 가중치와 편향같은 '자동'으로 획득되는 매개변수와 다름

신경망에서의 기울기

$$W = \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{pmatrix} \quad \frac{\partial L}{\partial W} = \begin{pmatrix} \frac{\partial L}{\partial w_{11}} & \frac{\partial L}{\partial w_{12}} & \frac{\partial L}{\partial w_{13}} \\ \frac{\partial L}{\partial w_{21}} & \frac{\partial L}{\partial w_{22}} & \frac{\partial L}{\partial w_{23}} \end{pmatrix}$$

W = 가중치, L = 손실함수

ex) $\frac{\partial L}{\partial W}$ 의 $\frac{\partial L}{\partial w_{11}}$ ≈ 0.2 \Rightarrow w_{11} 을 1 만큼 늘리면 loss function 0.2h ↑

학습 알고리즘 구현

1. 미니배치 - 훈련 데이터 중 일부를 무작위로 가져옴
 - 이렇게 선별한 데이터를 미니배치라 하며 이 미니 배치의 손실 값을 줄이는 것이 목표
2. 기울기 산출 - 손실 함수 값을 줄이기 위해 각 가중치 매개변수의 기울기를 구함. 손실 함수의 값이 가장 작은 방향을 제시
3. 매개변수 갱신 - 가중치 매개변수를 기울기 방향으로 조금씩 갱신
4. 반복

\Rightarrow 이러한 반복 과정을 확률적 경사하강법 (SGD)라고 함.

```

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):
        # 가중치 초기화
        self.params = {} ↗ 인스턴스 변수
        self.params['W1'] = weight_init_std * \
            np.random.randn(input_size, hidden_size) ↗ 첫 번째 층의 가중치
        self.params['b1'] = np.zeros(hidden_size) ↗ 첫 번째 층의 편향
        self.params['W2'] = weight_init_std * \
            np.random.randn(hidden_size, output_size) ↗ 두 번째 층의 가중치
        self.params['b2'] = np.zeros(output_size) ↗ 두 번째 층의 편향

    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) ⇒ Cross Entropy Function

    def accuracy(self, x, t):
        y = self.predict(x)
        y = np.argmax(y, axis=1)
        t = np.argmax(t, axis=1)

        accuracy = np.sum(y == t) / float(x.shape[0])
        return accuracy

    # x : 입력 데이터, t : 정답 레이블
    def numerical_gradient(self, x, t): ↗ 테스트
        loss_W = lambda W: self.loss(x, t)

        grads = {} ↗ 기울기 보관하는 변수
        grads['W1'] = numerical_gradient(loss_W, self.params['W1']) ↗ 가중치 기울기
        grads['b1'] = numerical_gradient(loss_W, self.params['b1']) ↗ 편향 기울기
        grads['W2'] = numerical_gradient(loss_W, self.params['W2'])
        grads['b2'] = numerical_gradient(loss_W, self.params['b2'])

        return grads

```

```
net = TwoLayerNet(input_size=784, hidden_size=100, output_size=10)
net.params['W1'].shape # (784, 100)
net.params['b1'].shape # (100,)
net.params['W2'].shape # (100, 10)
net.params['b2'].shape # (10,)
```

784 \Rightarrow 100 \Rightarrow 10

미니 배치 학습 구현

```
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_num = 10000 # 반복 횟수 = epoch
train_size = x_train.shape[0]
batch_size = 100 # 미니배치 크기
learning_rate = 0.1

network = TwoLayerNet(input_size=784, hidden_size=50, output_size=10)

for i in range(iters_num):
    # 미니배치 획득
    batch_mask = np.random.choice(train_size, batch_size)  $\sim$  랜덤 미니배치
    x_batch = x_train[batch_mask]
    t_batch = t_train[batch_mask]

    # 기울기 계산
    grad = network.numerical_gradient(x_batch, t_batch)
    # grad = network.gradient(x_batch, t_batch) # 성능 개선판!

    # 매개변수 갱신
    for key in ('W1', 'b1', 'W2', 'b2'):
        network.params[key] -= learning_rate * grad[key]

    # 학습 경과 기록
    loss = network.loss(x_batch, t_batch)
    train_loss_list.append(loss)
```

