3장

@ 손글씨 숫자 인식

미리 학습된 매개변수를 가져오기 때문에, 학습은 생략.

추론과정만 구현하는데, 여기 도입되는 추론 과정을 순전파라고 함

MNIST 데이터셋: 28*28 흑백, 0~255

책에서 제공하는, 미리 학습된 매개변수를 가져오는 load_mnist 함수는
(훈련 이미지, 훈련 레이블), (시험 이미지, 시험 레이블)
위의 형태로 값을 반환함.
함수에 전달할 수 있는 인수는 normalize, flatten, one_hot_label 총 3개인데,
normalize: 픽셀값 0~255를 0.0~1.0으로 정규화 할 것인지 결정
flatten: 1차원 배열로 만든 후 처리할 것인지 결정
one_hot_label: 레이블을 원-핫 인코딩 형태로 저장할것인가를 결정

```python
def predict(network, x):
    W1, W2, W3 = network['W1'], network['W2'], network['W3']
    b1, b2, b3 = network['b1'], network['b2'], network['b3']

    a1 = np.dot(x, W1) + b1
    z1 = sigmoid(a1)

    a2 = np.dot(z1, W2) + b2
    z2 = sigmoid(a2)

    a3 = np.dot(z2, W3) + b3
    y = softmax(a3)

    return y
```

```python
4   def sigmoid(x):
5       return 1 / (1+np.exp(-x))
```

```
c:\Users\kimtaeyoon\Document
encountered in exp
  return 1 / (1+np.exp(-x))
Accuracy:0.9207
PS C:\Users\kimtaeyoon\Docum
```

```python
def sigmoid(x):
    return np.exp(-np.logaddexp(0, -x))
```

```
Accuracy:0.9207
PS C:\Users\kimtaeyoon\D
```

@ 배치 처리
위에서 구현한 구조는
784 -> 784*50 -> 50*100 -> 100*10 -> 10 형상인데

예를 들어서 100장의 이미지를 한번에 넣고 싶으면
100*784 -> 784*50 -> 50*100 -> 100*10 -> 100*10 형상으로 구현 가능

! 배치처리는 연산속도 향상에 장점이 있음. 1장당 처리 시간을 대폭 줄여줌. 먼저 라이브러리들이 큰 배열을 처리하는데 최적화되어있기 때문이다. 두 번째로 데이터 전송 자체가 병목으로 작용하는 경우도 있어 배치처리를 통해 버스에 가하는 부하를 줄여줄 수 있다.

@ 배치 처리의 구현

```
52  accuracy_cnt = 0
53  for i in range(len(x)):
54      y = predict(network, x[i])
55      p = np.argmax(y)
56      if p == t[i]:
57          accuracy_cnt += 1
```

```
Accuracy:0.9207
time:  0.4378316402435303
```

```
63  batch_size = 100
64  accuracy_cnt = 0
65  for i in range(0, len(x), batch_size):
66      x_batch = x[i:i+batch_size]
67      y_batch = predict(network, x_batch)
68      p = np.argmax(y_batch, axis = 1)
69      accuracy_cnt += np.sum(p == t[i:i+batch_size])
```

```
Accuracy:0.9207
time:  0.09075736999511719
```

@@ 3장 결론 -> 신경망에서 순전파 과정에 퍼셉트론의 스텝함수와 달리 비선형 시그모이드 함수를 도입했다.

4장 신경망 학습

손실 함수: 성능이 얼마나 나쁜지를 나타내는 지표.

@ 오차제곱합(가장 많이쓰는 손실함수)

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

여기서 y_k는 출력, t_k는 정답 레이블, k는 데이터의 차원 수

```
def sum_squares_error(y, t):
    return 0.5 * np.sum((y-t)**2)
```

```
t = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]

y1 = [0.1, 0.05, 0.6, 0.0, 0.05, 0.1, 0.0, 0.1, 0.0, 0.0]
print(sum_squares_error(np.array(y1), np.array(t)))


y2 = [0.1, 0.05, 0.1, 0.0, 0.05, 0.1, 0.0, 0.6, 0.0, 0.0]
print(sum_squares_error(np.array(y2), np.array(t)))
```

```
0.09750000000000003
0.5975
```

정답 레이블에 맞게 확률을 도출해낼 경우 오차제곱합이 더 작게 출력되는 것을 확인할 수 있다.

@ 교차 엔트로피 오차

$$E = -\sum_k t_k \log y_k$$

-> 실질적으로 정답 레이블 에서의 $-\log y_k$이다.

```python
def cross_entropy_error(y, t):
    delta = 1e-7 # -inf 발산 대비
    return -np.sum(t*np.log(y+delta))
```

```python
t = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]

y1 = [0.1, 0.05, 0.6, 0.0, 0.05, 0.1, 0.0, 0.1, 0.0, 0.0]
print(cross_entropy_error(np.array(y1), np.array(t)))

y2 = [0.1, 0.05, 0.1, 0.0, 0.05, 0.1, 0.0, 0.6, 0.0, 0.0]
print(cross_entropy_error(np.array(y2), np.array(t)))
```

```
0.510825457099338
2.302584092994546
```

정답 레이블에 맞게 확률을 도출해낼 경우 교차 엔트로피 오차가 더 작게 출력되는 것을 확인할 수 있다.

@ 미니 배치 학습

-> 위에서 나왔던 배치학습에서 랜덤으로 값을 꺼내 배치 학습을 돌리는 것으로, 생략.

@ 왜 손실함수를 설정하는가? (상세)

-> 왜 '정확도'가 아닌 손실함수라는 우회적인 지표를 도입하느냐?

신경망 학습에서는 손실함수의 값을 가장 작게하는 매개변수(weight와 bias)를 탐색하는데, 매개변수의 미분을 계산하고 그 미분값을 기반으로 값을 튜닝하는 과정을 반복한다.

매개변수의 값을 약간 변화시키고 손실함수의 미분값을 조사하고 이를 줄여주는 방향으로 매개변수를 튜닝하는데, 이때 0이 되면 튜닝을 반복을 멈추는 형식으로 학습이 진행된다.

하지만 '정확도'는 대부분의 구역에서 미분값이 0이 되어 매개변수를 튜닝하는데 적합하지 않다.

왜 대부분의 구역에서 미분이 0이 되는가?

100장중 32장을 구분해내는 형태에서, 정확도는 32%이며 매개변수의 값이 약간 달라져도 변하지 않거나 33%, 31% 등 불연속적으로 값이 변한다. 따라서 사실상 모든 구역에서 미분값이 0이다.

@ 파이썬을 통한 미분 구현

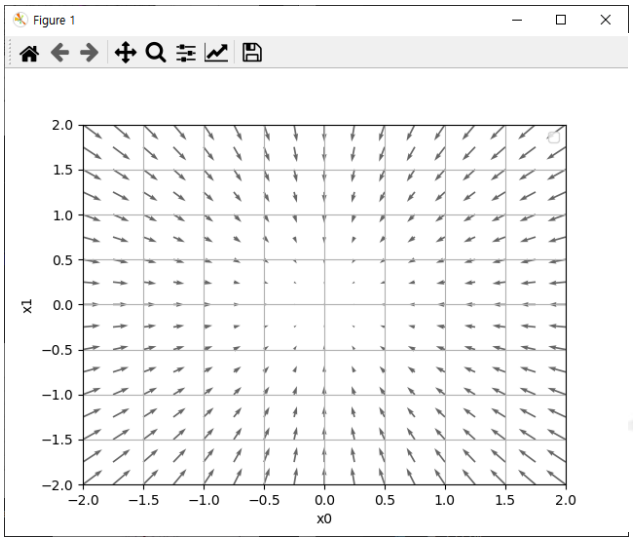
$$f'(x) \simeq \lim_{h \to 0} \frac{f(x+h) - f(x-h)}{2h}$$

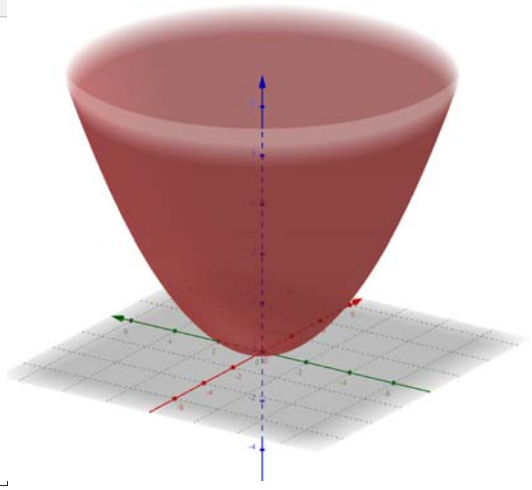
```python
def numerical_diff(f, x):
    h = 1e-4
    return (f(x+h) - f(x-h))/(2*h)
```

@ 편미분을 응용한 gradient 계산

$$f(x_0, x_1) = x_0^2 + x_1^2$$

```python
def numerical_gradient(f, x):
    h = 1e-4
    grad = np.zeros_like(x)

    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
```





$$f(x_0, x_1) = x_0^2 + x_1^2$$

```python
print(numerical_gradient(function_2, np.array([3.0, 4.0])))
```
-> `[6. 8.]`

출력되는 gradient가 의미하는 것

-> ! ! ! 각 좌표에서 함수의 출력 값을 가장 크게 줄이는 방향 ! ! !

@ 경사하강법

경사하강법을 수식으로 나타내면,

$$x_0 = x_0 - \eta \frac{\partial f}{\partial x_0}, \quad x_1 = x_1 - \eta \frac{\partial f}{\partial x_1}$$

η는 갱신하는 양을 나타내며, 학습률(learing rate)이라고 한다.

η의 값으로는 0.1, 0.001 등 특정값으로 정해주어야 하며, 너무 크거나 작으면 적절한 값을 찾지 못한다.

```python
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
```

그림 22 : 경사하강 학습의 구현

```python
init_x = np.array([-3., 4.])
print(gradient_descent(function_2, init_x = init_x, lr=0.1, step_num = 100))
```

```
[-6.11110793e-10  8.14814391e-10]
```

위와 같이 사실상 0이라는 수를 찾아낸 것을 확인할 수 있다.

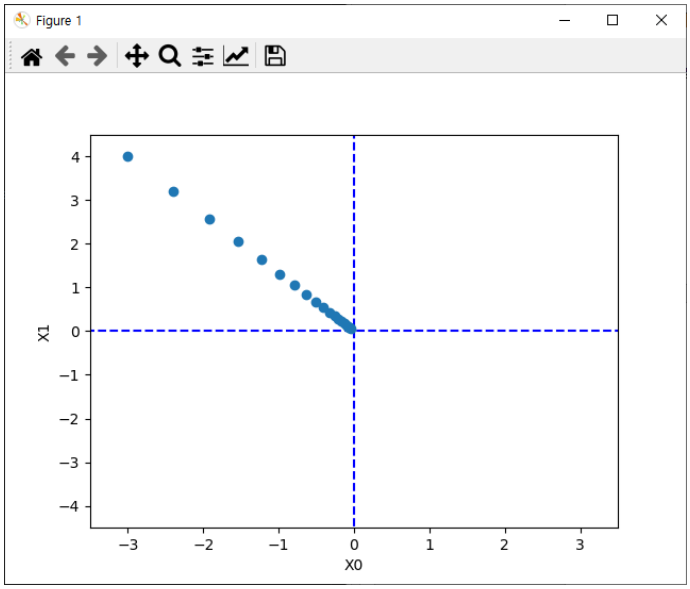


그림 25 : 학습 과정을 나타낸 그림 (lr = 0.1)

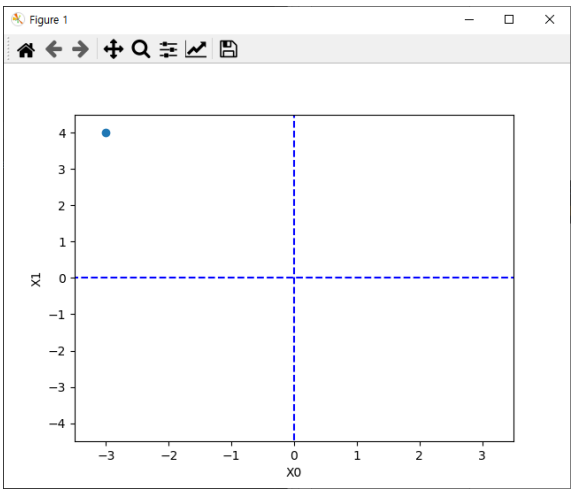


그림 26 : lr = 10.0

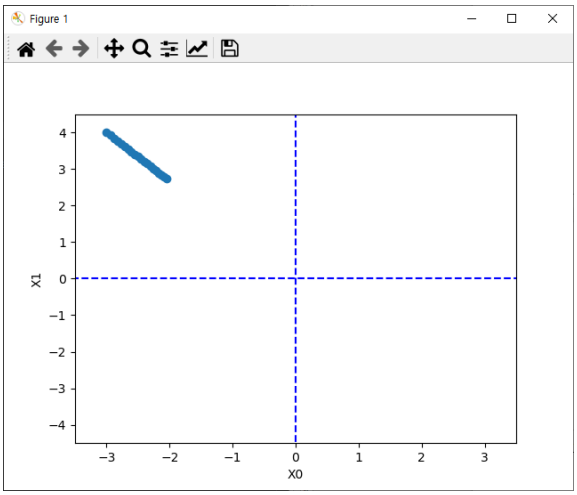


그림 27 : lr = 0.01

learning rate(학습률)을 조정하는 것은 매우 중요하다. weight는 자동으로 획득할 수 있는 반면, 학습률과 같은 매개변수는 사람이 직접 설정해야하므로 hyper parameter라고 한다.

@ 학습 알고리즘 구현하기

먼저 2층 신경망(은닉층 1개)을 하나의 클래스로 구성.

1 에폭이란? 훈련데이터를 모두 소진하는 횟수. ex) 10000개의 훈련데이터에서 100개 미니배치 학습 -> 경사하강 100회 시행 시 1 에폭.

784 -> 50 -> 10 형태