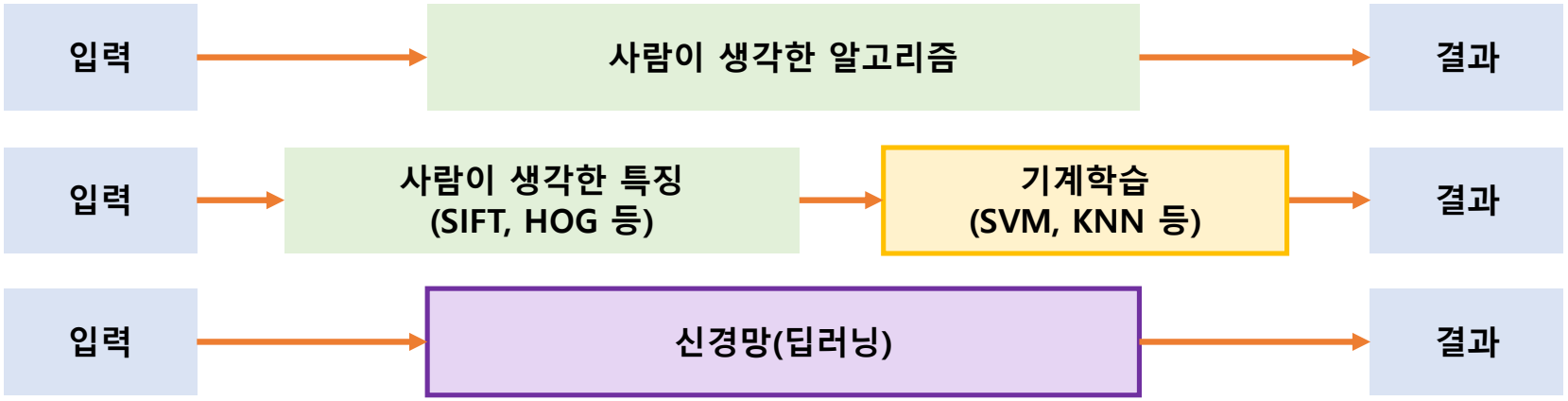
신경망 학습 :
Neural Network Learning

[Deep Learning from scratch] - 사이토 고키

중요 내용 요약 : 김진수

데이터 주도 학습



- 데이터(입력)에서 결과(출력)를 사람의 개입 없이 얻는다는 뜻으로, **종단간 기계학습 (End-to-end learning)**이라고도 한다.
- 기계학습에서는 **훈련 데이터(Training data)**와 **시험 데이터(Test data)**로 나누어 훈련 데이터로 학습하여 최적의 매개변수를 찾은 다음, 시험 데이터로 훈련된 모델의 실력을 평가하는 것이다.
- **범용 능력**(아직 못 본 데이터에서도 동작)을 제대로 평가하기 위해 시험 데이터를 사용하며, 훈련 데이터에만 지나치게 최적화된 상태를 **오버피팅(Overfitting)**이라 한다.

손실 함수(Loss function)

- 손실 함수는 신경망 학습에서 **최적의 매개변수 값을 탐색하는 지표**로, **신경망 성능의 오차 정도**를 나타낸다. 즉, 손실이 클수록 신경망 성능이 좋지 않다는 의미이다.
- '정확도'가 아닌 '손실 함수의 값'을 이용하는 이유?

  ⇒ 신경망 학습에서의 **'미분'의 역할**, '정확도'를 지표로 하면 가중치 매개변수의 미분 값이 대부분의 장소에서 0이 되기 때문이다.
- 신경망 학습에서는 최적의 매개변수(가중치와 편향)를 탐색할 때 **손실 함수의 값을 가능한 한 작게 하는 매개변수 값**을 찾는다 - 이때 **매개변수의 미분(기울기)를 계산**, 그 미분 값을 단서로 매개변수의 값을 서서히 갱신하는 과정을 반복한다.

손실 함수(Loss function)

- 자주 이용하는 손실 함수로는 **평균 제곱 오차(Mean squared error)**와 **교차 엔트로피 오차 (Cross entropy error)**가 있다.
- 평균 제곱 오차

  $MSE = \frac{1}{2}\sum_k^K (y_k - t_k)^2$ (y_k는 신경망의 출력, t_k는 정답 레이블, K는 출력 요소의 개수, k는 각 요소)
- 교차 엔트로피 오차

  $CE = -\sum_k^K t_k log y_k$, 원-핫 인코딩(One-hot encoding, 정답에 해당하는 인덱스 원소만 1이고 나머지는 0인 인코딩 방식)에서는 정답일 때의 출력이 전체 값을 정하게 된다.

손실 함수(Loss function) : Code

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

- 평균 제곱 오차(mean squared error)

$$MSE = \frac{1}{2}\sum_{k}^{K}(y_k - t_k)^2$$

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

- 교차 엔트로피 오차(cross entropy error)

$$CE = -\sum_{k}^{K} t_k log y_k$$

y_k의 값이 0일 경우에, 디지털에서는 값이 존재할 수 없으므로(-inf), 미세 값 delta(10^{-7})를 부여한다.

미니 배치 학습

- 전체 훈련 데이터에서 **무작위 데이터 일부를 추려** 전체 데이터 학습의 '**근사치**'로 접근하는 방식을 '**미니 배치 학습**'이라 한다.

- 미니 배치인 여러 개의 데이터에 대한 평균 교차 엔트로피 오차 함수 값

$$E = -\frac{1}{N}\sum_{n}^{N}\sum_{k}^{K} t_{nk} log y_{nk}$$

- y_{nk} : 신경망의 출력, t_{nk} : 정답 레이블, N : 미니 배치 데이터의 개수, n : 각 데이터, K : 1개 데이터 당 출력 요소의 개수, k : 각 요소

미니 배치 학습을 위한 Code

```
train_size = x_train.shape[0]
batch_size = 10
batch_mask = np.random.choice(train_size, batch_size)
x_batch = x_train[batch_mask]
t_batch = t_train[batch_mask]
print(batch_mask)
```

```
[49805 48050 24264 48077 20302 51335 26870 28894 27837 39491]
```

- $np.random.choice(x, y)$: $0 \sim x(int)$에서 $y(int)$개의 랜덤한 숫자를 골라낸 list를 return
- x_batch : x_train 중에서 10개($batch\_size$)의 $batch_mask$만 적용된 데이터 (t_batch도 t_train에서 동일)
- batch_mask list의 10개 요소 출력

```
def batch_cross_entropy_error(y, t):
    delta = 1e-7
    if y.ndim == 1:
        batch_size = 1
    else:
        batch_size = y.shape[0]
    return -np.sum(t * np.log(y + delta)) / batch_size
```

- $batch\_cross\_entropy\_error$: 여러 개의 데이터에 대한 평균 교차 엔트로피 오차를 구하기 위해 $batch_size$로 나누어준다.
- $batch_size$는 $y.shape[0]$로 받아올 수 있다.
- cross entropy error의 수식에 맞게 return

수치 미분

- 경사법에서는 기울기(경사) 값을 기준으로 나아갈 방향을 정한다.

  ⇒ 신경망 학습 시에는 **매개변수의 미분(기울기)을 기준으로 경사하강법**을 시행한다.

  ⇒ 경사하강법으로 **손실 함수의 값이 작아지도록**, 즉 정확도가 높아지도록 학습한다.

- 함수의 미분 : 함수의 아주 짧은 순간 변화량

$$\frac{d\,f(x)}{dx} = \lim_{h \to 0} \left(\frac{f(x+h) - f(x)}{h} \right)$$

- **중앙 차분** : 수치 미분에서 오차를 줄이기 위해 $\boldsymbol{x}$**를 기준으로 그 전후의 차분을 계산**

$$\frac{d\,f(x)}{dx} = \lim_{h \to 0} \left(\frac{f(x+h) - f(x-h)}{2h} \right)$$

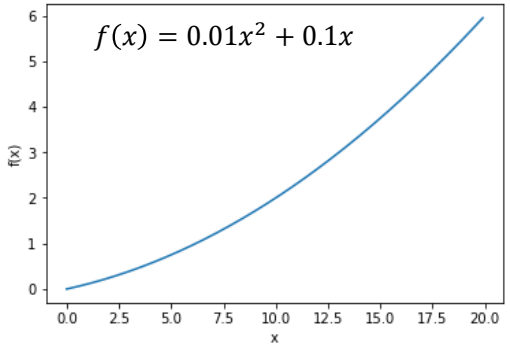
미분(중앙 차분) : Code

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

- $\frac{d\,f(x)}{dx} = \lim_{h \to 0} \left(\frac{f(x+h) - f(x-h)}{2h} \right)$을 구현한 함수이며, h는 반올림 오차 문제를 일으키지 않는 10^{-4} 정도의 작은 값을 설정한다. (너무 작을 시에는 float시 0.0이 되어버릴 수 있다.)

```
def function_1(x):
    return 0.01*(x**2) + 0.1*x
x = np.arange(0.0, 20.0, 0.1)
y = function_1(x)
plt.plot(x, y)
plt.xlabel('x')
plt.ylabel('f(x)')
plt.show()
```

- $f(x) = 0.01x^2 + 0.1x$의 식을 기준으로 설명, x에 대한 $y = f(x)$식을 그래프로 그리면 오른쪽의 그림과 같다.



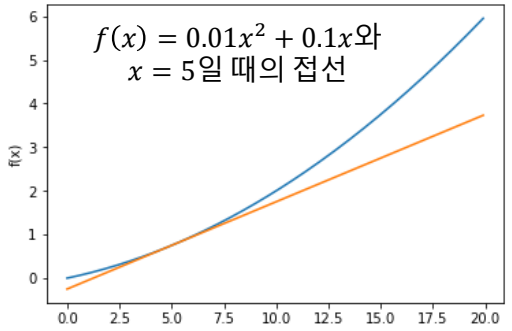

```
numerical_diff(function_1, 5)
```

0.1999999999990898

- $x = 5$일 때의 미분 값은 수치적으로 0.2와 거의 같은 값이라고 할 만큼 작은 오차이다.

```
def tangent_line(f, x):
    a = numerical_diff(f, x)
    b = f(x) - a*x
    return lambda t: a*t + b
y2 = tangent_line(function_1, 5)
plt.plot(x, y)
plt.plot(x, y2(x))
plt.xlabel('x')
plt.ylabel('f(x)')
plt.show()
```

- $f(x) = 0.01x^2 + 0.1x$에서, $x = 5$일 때의 접선을 구하는 함수($tangent\_line$)는 오른쪽 그림에서 주황색 선을 보이고, 함수 return 시에 $lambda$를 이용하여 $y2(x)$를 일괄적으로 plot하였다.



기울기 (Gradient)

- 편미분 : 변수가 여럿인 함수에서, 어느 한 편(변수)에 대한 미분인가?

  ex) $f(x_0, x_1) = {x_0}^2 + {x_1}^2$에서 $x_0 = 3,\ x_1 = 4$일 때, $\frac{df}{dx_0}$의 값은?

  x_0에 대한 편미분 $\frac{df}{dx_0} = 2x_0$이므로, $\frac{df}{dx_0} = 6$이 된다.

- 기울기 : 변수 별로 편미분을 따로 계산하는 것이 아닌, **모든 변수의 편미분을 벡터로 정리한 것**을 기울기라 한다.

  ex) $f(x_0, x_1) = {x_0}^2 + {x_1}^2$에서 기울기 벡터는 $(\frac{df}{dx_0}, \frac{df}{dx_1})$이라고 할 수 있다.

  $\frac{df}{dx_0} = 2x_0$, $\frac{df}{dx_1} = 2x_1$이므로 $x_0 = 3,\ x_1 = 4$일 때, 기울기 벡터는 $(6, 8)$이 된다.

- 기울기의 결과에서, 마이너스를 붙인 벡터가 가리키는 방향은 **함수의 출력 값을 가장 크게 줄이는 방향**이 된다. ⇒ 이후에 나올 **경사 하강법** 의미

기울기 (Gradient) : Code

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

    for idx in range(x.size):
        tmp_val = x[idx]
        x[idx] = tmp_val + h
        fxh1 = f(x)  # f(x+h)

        x[idx] = tmp_val - h
        fxh2 = f(x)  # f(x-h)

        grad[idx] = (fxh1 - fxh2) / (2*h)
        x[idx] = tmp_val

    return grad
```

```
# for numerical_ gradient
def function_2(x):
    return np.sum(x**2)
```

```
numerical_gradient(function_2, np.array([3.0, 4.0]))
```

```
array([6., 8.])
```

```
numerical_gradient(function_2, np.array([0.0, 2.0]))
```

```
array([0., 4.])
```

```
numerical_gradient(function_2, np.array([3.0, 0.0]))
```

```
array([6., 0.])
```

- 수치미분을 위한 h의 값으로 10^{-4} 값을 사용
- $grad$는 x와 size가 같고, 0의 값을 갖는 numpy array
- $x[idx]$ 값을 임시 저장해놓은 뒤,
  편미분을 위해 $x[idx]$에만 h값을 더하고 f에 대입하여
  $f(x+h)$를 의미하는 $fxh1$을 설정
  같은 방식으로 $x[idx]$에만 h값을 빼고 f에 대입하여
  $f(x-h)$를 의미하는 $fxh2$을 설정
  $grad[idx]$에 $\frac{f(x+h)-f(x-h)}{2h}$ 식을 대입하고, $x[idx]$ 값 복원
- $x_0 = 3,\ x_1 = 4$일 때, $f(x_0, x_1) = {x_0}^2 + {x_1}^2$ 식의 $numerical\_gradient$
  $\left(\frac{df}{dx_0}, \frac{df}{dx_1}\right) = (2x_0, 2x_1) = (6, 8)$이고, 같은 방식으로
  $x_0 = 0,\ x_1 = 2$일 때는 $(0, 4)$이고, $x_0 = 3, x_1 = 0$일 때는 $(6, 0)$이 된다.

신경망에서의 기울기

- 가중치 $\boldsymbol{W} = \begin{pmatrix} \boldsymbol{w_{11}} & \boldsymbol{w_{12}} & \boldsymbol{w_{13}} \\ \boldsymbol{w_{21}} & \boldsymbol{w_{22}} & \boldsymbol{w_{23}} \end{pmatrix}$일 때, 손실 함수가 L인 신경망이라면,

  경사(기울기) $\frac{\boldsymbol{dL}}{\boldsymbol{dW}} = \begin{pmatrix} \frac{\boldsymbol{dL}}{\boldsymbol{dw_{11}}} & \frac{\boldsymbol{dL}}{\boldsymbol{dw_{12}}} & \frac{\boldsymbol{dL}}{\boldsymbol{dw_{13}}} \\ \frac{\boldsymbol{dL}}{\boldsymbol{dw_{21}}} & \frac{\boldsymbol{dL}}{\boldsymbol{dw_{21}}} & \frac{\boldsymbol{dL}}{\boldsymbol{dw_{23}}} \end{pmatrix}$로 나타낼 수 있다.

- $\frac{\boldsymbol{dL}}{\boldsymbol{dW}}$의 각 원소($\frac{dL}{dw_{11}}$, $\frac{dL}{dw_{12}}$, …)**는 $\boldsymbol{W}$**의 각 원소(w_{11}, w_{12}, …)**에 대한 편미분**이며,

  예를 들어 $\frac{dL}{dw_{11}}$는 w_{11}가 조금 변경되었을 때의 **손실 함수 $\boldsymbol{L}$의 변화량**을 의미한다.

- 여기서, W의 각 원소에 대한 편미분 값이 $\frac{dL}{dW}$의 각 원소의 값이 되므로,

  $\boldsymbol{W}$와 $\frac{\boldsymbol{dL}}{\boldsymbol{dW}}$의 shape(여기선 2 X 3)**은 같아야 한다**.

경사 하강법(Gradient descent)

- **함수의 최솟값**을 찾기 위한 방법으로, 각 지점에서 **기울기를 지표로써 사용**하는 방법
  ⇒ 신경망에서는 **손실 함수의 최솟값**을 찾기 위해 사용된다.
- **기울기가 0인 지점**(극솟값 또는 최솟값)**의 방향으로 하강**하며 **매개변수를 갱신**한다.
  ⇒ 최적화 방법(Optimization)이라 하며 SGD, Momentum, RMSProp, Adam 등이 있다.
- $x_0 = x_0 - \eta \frac{df}{dx_0}$
  $x_1 = x_1 - \eta \frac{df}{dx_1}$
  여기서, $\boldsymbol{\eta}$**는 학습률(learning rate)**, (−)부호는 하강법을 의미하며, 1회에 해당하는 갱신이고 이 단계를 반복하게 된다.
- 학습률은 0.01, 0.001 등 특정 값으로 직접 정해두어야 하는데(hyper-parameter),
  신경망 학습에서는 **학습률 값을 변경하면서 올바르게 학습하고 있는지를 확인**해야한다.

경사 하강법 : Code

```python
# for gradient descent
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
```

- $step_num$만큼 반복하면서 $numerical\_gradient$ 함수에 f, x를 넘겨주면 $grad$(기울기 벡터)를 return 해준다.
- 식 $x_0 = x_0 - \eta \frac{df}{dx_0}$, $x_1 = x_1 - \eta \frac{df}{dx_1}$ 구현

  (Code에서 $x[idx] \Rightarrow x_{idx}$, $\eta \Rightarrow lr$, $grad[idx] \Rightarrow \frac{df}{dx_{idx}}$ 를 의미)

```python
init_x = np.array([-3.0, 4.0])
gradient_descent(function_2, init_x=init_x, lr=0.1, step_num=100)
```

```
array([-6.11110793e-10,  8.14814391e-10])
```

```python
# too large learning rate
init_x = np.array([-3.0, 4.0])
gradient_descent(function_2, init_x=init_x, lr=10.0, step_num=100)
```

```
array([-2.58983747e+13, -1.29524862e+12])
```

```python
# too small learning rate
init_x = np.array([-3.0, 4.0])
gradient_descent(function_2, init_x=init_x, lr=1e-10, step_num=100)
```

```
array([-2.99999994,  3.99999992])
```

- Ex) $f(x_0, x_1) = {x_0}^2 + {x_1}^2$ (기울기 Code의 $function\_2(x)$)에서, 기울기 벡터를 이용하여 경사 하강법을 진행한다.
- lr(학습률)이 적당하면 경사 하강법이 적당하게 잘 진행되어 최저점인 (0,0)에 아주 가까운 지점으로 수렴
- lr(학습률)이 너무 크면, 최저점인 (0,0)을 한참 지나 아주 큰 값으로 발산할 수 있다.
- lr(학습률)이 너무 작으면 시작점인 $init_x$에서 거의 진행되지 않아서 최저점인 (0,0) 근처에도 가지 못한 채 학습이 끝나버림

신경망 학습 알고리즘 - MNIST

- 신경망의 매개변수(가중치와 편향)를 MNIST 훈련 데이터로 조정하는 **'신경망 학습'**을 진행
- 1단계 : 미니배치

  훈련 데이터의 **미니배치를 무작위로 선정**하는 **확률적 경사 하강법(Stochastic gradient descent : SGD)**를 이용하여 손실 함수 값을 줄이는 것이 목표

- 2단계 : 기울기 산출

  미니배치의 손실 함수 값을 줄이기 위해 **가중치 매개변수의 기울기**를 구하여, 손실 함수의 값을 가장 작게 하는 방향을 제시

- 3단계 : 매개변수 갱신

  가중치 매개변수를 **기울기 방향으로 아주 조금 갱신 (학습률에 비례)**

- 4단계 : 반복

  1~3 단계를 반복

수치미분을 적용한 신경망 구현 : Code

```
class TwoLayerNet:
    def __init__(self, input_size, hidden_size, output_size,
                 weight_init_std=0.01):
        self.params = {}
        self.params['W1'] = weight_init_std * \
                    np.random.randn(input_size, hidden_size)
        self.params['b1'] = np.zeros(hidden_size)
        self.params['W2'] = weight_init_std * \
                    np.random.randn(hidden_size, output_size)
        self.params['b2'] = np.zeros(output_size)

    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(a1, W2) + b2
        y = softmax(a2)

        return y
```

- 먼저, 신경망으로 사용될 $TwoLayerNet$이라는 class를 생성해준다.
- Initialize($\_\_init\_\_$) : 가중치 매개변수들의 무작위 초기값을 setting

  ⇒ $params$ 라는 $dictionary$에 $weight$와 $bias$의 변수를 생성하여

  표준 편차가 0.01인 정규분포의 무작위 값을 $weight$ 초깃값으로 대입하는데,

  이는 $weight$의 초깃값에 의해 학습 결과가 좌우되는 일을 줄이기 위함과

  가중치가 고르게 되지 않도록 무작위로 선정하는 데에 의미가 있다.
- $predict$ 함수에서는 x에 대한 forward processing($*\ weight\ +\ bias$)을 하고,

  활성화 함수로는 $sigmoid$, 출력층으로는 $softmax$의 결과를 return 해준다.

수치미분을 적용한 신경망 구현 : Code

```
def loss(self, x, t):
    y = self.predict(x)
    return batch_cross_entropy_error(y, t)

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

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
```

- $loss$ 함수에서는 x에 대한 $predict$의 return 값과 t를 b$atch_cross_entropy_error$를 대입하여, 손실 함수로 사용한다.
- $accuracy$ 함수에서는 x의 $predict$ 결과인 y와 t를 비교한다.

  $\Rightarrow np.argmax$를 통해 가장 큰 값을 갖는 index를 출력하는데,

  $y.shape$: $(batch\_size, 10)$ 에서 $argmax$할 $axis$ 기준을 선택하여

  y와 t가 같은지를 비교, True(1) / False(0)로 합산한 다음 전체 개수로 나누어 계산한 $accuracy$를 return

  $\underset{axis=0}{} \quad \underset{axis=1}{}$
- $numerical\_gradient$ 함수에서는 $loss$에서 구한 손실 함수 값을 이용, 각 가중치 매개변수마다의 $loss$ 값에 대한 경사 하강법을 실행하여, $grads$라는 $dictionary$에 저장하여 그 변수를 return

신경망 학습 알고리즘 : Code

```
(x_train, t_train), (x_test, t_test) = ₩
        load_mnist(normalize=True, one_hot_label=True)
```

```
iters_num = 1000  # 실제 코드 : 10000
train_size = x_train.shape[0]
batch_size = 500  # 실제 코드 : 100
learning_rate = 0.1

train_loss_list = []
train_acc_list = []
test_acc_list = []

iter_per_epoch = train_size // batch_size
print('iter_per_epoch =', iter_per_epoch)
print('epochs =', iters_num // iter_per_epoch
      if iters_num % iter_per_epoch == 0 else iters_num // iter_per_epoch + 1)
```

```
iter_per_epoch = 120
epochs = 9
```

```
net = TwoLayerNet(input_size=784, hidden_size=50, output_size=10)
print(net.params['W1'].shape)
print(net.params['b1'].shape)
print(net.params['W2'].shape)
print(net.params['b2'].shape)
```

```
(784, 50)
(50,)
(50, 10)
(10,)
```

- *load_mnist* 함수로 0~1의 값으로 정규화와 one-hot 인코딩이 되어 있는 (정답 1, 나머지 0인) MNIST 데이터를 불러온다.
- 여러 하이퍼 파라미터들(학습 전 조절 값들)을 설정해준다.
  => 간단한 결과를 보기 위해 반복 횟수를 줄이고 배치 사이즈를 늘렸다.
  PC의 CPU 성능이 좋고, RAM의 메모리가 커서 배치 사이즈를 넉넉히 설정했다.
- *iter_per_epoch*(훈련용 데이터 크기 / 배치 사이즈)을 반복 횟수로 나누어준 값, 즉, 훈련 데이터 전체에 대한 1회 학습 반복을 epoch이라 하며, 정확히 떨어지지 않을 경우 마지막 반복(+1)을 추가했다.
- 앞서 정의해 두었던 *TwoLayerNet*(*net*이라 명명)을 불러온다.

	X		W1		W2	→	Y
Shape:	(500, 784)	x	**(784, 50)**	x	**(50, 10)**		(500, 10)
			+B1		+B2		
			(50,)		**(10,)**		

신경망 학습 알고리즘 : Code

```
# train
start_time = time.time()
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 = net.numerical_gradient(x_batch, t_batch)

    for key in ('W1', 'b1', 'W2', 'b1'):
        net.params[key] -= learning_rate * grad[key]

    loss = net.loss(x_batch, t_batch)
    train_loss_list.append(loss)
    if i % iter_per_epoch == 0 or i == iters_num-1:
        train_acc = net.accuracy(x_train, t_train)
        test_acc = net.accuracy(x_test, t_test)
        train_acc_list.append(train_acc)
        test_acc_list.append(test_acc)
        print('%d iter train acc, test acc : %.4f, %.4f'
              % (i, train_acc, test_acc))
        print('%d iteration time = %.2f sec' % (i, time.time() - start_time))
```

- $batch\_mask$: $0 \sim train_size - 1$까지의 정수 중에, $batch\_size$ 개수의 무작위 값을 $np.random.choice$가 선택해준 numpy array

  ⇒ x_train과 x_batch에 적용하여 x_batch와 t_batch를 생성

- $net.numerical_gradient$를 통해 기울기 벡터($grad$)를 계산 후, 각 매개변수들에 대해 $learning\_rate$ 만큼의 경사하강법을 진행한다.

- x_batch와 t_batch를 $net.loss$ 함수로 넘겨주면, $loss$를 return

  ⇒ $net.loss$ 내부에선 x_batch의 predict 결과와 t_batch 사이의 cross entropy error를 계산하여 $loss$로 return한다.

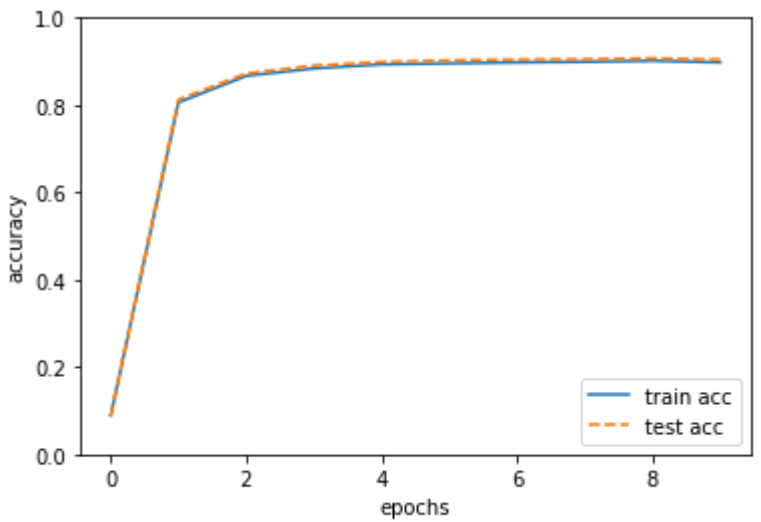
```
0 iter train acc, test acc : 0.0889, 0.0865
0 iteration time = 152.28 sec
120 iter train acc, test acc : 0.8062, 0.8123
120 iteration time = 18189.62 sec
240 iter train acc, test acc : 0.8670, 0.8725
240 iteration time = 36347.82 sec
360 iter train acc, test acc : 0.8840, 0.8906
360 iteration time = 55320.22 sec
480 iter train acc, test acc : 0.8932, 0.8988
480 iteration time = 73661.63 sec
600 iter train acc, test acc : 0.8956, 0.9025
600 iteration time = 92374.45 sec
720 iter train acc, test acc : 0.8979, 0.9042
720 iteration time = 110415.52 sec
840 iter train acc, test acc : 0.8992, 0.9053
840 iteration time = 128502.12 sec
960 iter train acc, test acc : 0.9015, 0.9072
960 iteration time = 146844.47 sec
999 iter train acc, test acc : 0.8979, 0.9044
999 iteration time = 152915.06 sec
```

- 학습이 끝난 후에 epoch 마다 $loss$를 확인할 수 있도록 하기 위해 $train_loss_list$에 append 한다. (추후에 사용하진 않음)

- epoch 마다 $net.accuracy$를 통해 훈련용 데이터와 테스트 데이터를 추정한 결과의 정확도, 걸린 시간을 print 해주며, list에 저장

신경망 학습 알고리즘 : Code

```
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()
```



- $train_acc_list$를 plot한 그래프 위에 $test_acc_list$의 선 스타일이 다르도록 plot 해주고, 축 명과 y축 범위, 범례를 설정한다.
- 매 epoch에 따라 훈련용 데이터를 이용한 추정 결과 ($train\_acc\_list$) 값과 테스트 데이터를 이용한 추정 결과($test\_acc\_list$) 값이 높아짐을 그래프로 알 수 있으며, 학습이 진행 되고 있음을 알 수 있다.
- $train\_acc\_list$ 값과 $test\_acc\_list$ 값이 거의 동일한 양상을 보이므로, 훈련용 데이터 과적합(overfitting)이 일어나지 않았음을 확인할 수 있다.
  ⇒ 훈련용 데이터의 추정 결과는 좋지만 테스트 데이터의 추정 결과가 좋지 않다면, overfitting을 의심할 수 있다. 예방법으로는 추후에 살펴볼 Dropout 방식과 가중치 감소(weight decay)등이 있다.