인공지능개론 정리노트#2

201904022 김상옥

202284026 안정빈

질문1. 미니 배치의 크기가 속도와 정확도에 어떤 영향을 미치며, 적절한 미니 배치의 수는 무엇일까?

김상옥 : 미니 배치 수가 크면 정확도가 더 높아지지 않을까?

안정빈 : 미니 배치 수가 크면 학습 속도가 빨라지니까 많은 데이터로 계산을 하면서 정확도를 높일 수 있을 것 같다는 생각이 들기는 하지만 과적합 위험도가 있을 것 같아.

김상옥 : 미니 배치가 범용성을 위한 학습이니까 너무 많은 크기로 학습하면 범용성이 떨어지는 게 맞는 거 같다. 그런데 학습 속도는 오히려 오래 걸리지 않을까? n번 반복하여 학습한다고 할 때 한번에 많은 양을 n번 처리하는 것 보다 적은 양을 n번 처리하는 게 더 빠르다고 생각해

안정빈 : 학습 속도는 미니배치 크기가 작을 경우가 더 빠른 게 맞는 것 같아! 그러면 미니배치의 크기가 클 때는 업데이트 당 속도가 느리고 많은 데이터가 있으면 메모리 부족의 위험도는 없을까?

김상옥 : 더 복잡하고 어려운 문제를 위해서는 신경망의 깊이나 가중치의 수가 엄청 많을 것이니 분명 메모리나 하드웨어에 부담이 많을 것 같아. 그래서 엔비디아같은 회사에서 AI를 위한 하드웨어를 개발하는 거라고 생각해. 아직은 공부를 위한 간단한 데이터로 활용했지만 이마저도 생각보다 오래걸린다고 느껴졌던 걸 생각하면 실제로 사용될 AI학습은 하드웨어에 영향을 받고 시간도 많이 걸릴 것 같아. 그렇기 때문에 속도도 느리지 않으면서 정확도(범용성)를 챙길 수 있는 미니 배치의 수가 중요하다고 생각해. 아까 미니 배치의 수가 크면 범용성이 떨어질 것이라 했는데 어떤 영향이 있을까?

안정빈 : 우선 학습 속도가 느려지고 미니 배치 수가 크면 모델이 학습 데이터의 특정 패턴에 지나치게 의존하게 되어 일반화 능력이 저하되고 다양한 패턴을 덜 학습하게 되는 것 같아. 그래서 범용성을 고려할 때 적절한 미니 배치 크기를 선택해야 되는 것 같아.

김상옥 : 직접 해보니 일정 수 이상으로는 큰 의미가 있을 정도로 정확도가 개선되지는 않고, 오히려 학습 속도에 영향을 주었는데 확실히 적절한 배치 크기를 설정하는게 중요한 거 같다. 또 학습할 신경망에 따라 반복수, 미니 배치 수 등 다양한 요소를 고려하여 최선의 학습을 구하는게 AI에 중요한 점인 거 같아. **(하단에 사진 첨부 하였습니다.)**

안정빈 : 우리 둘다 미니 배치 크기가 적절해야 한다고 생각하는 것 같아. 그러면 적절한 미니배치의 수가 무엇인지 생각해보자

결론 : 적절한 미니배치의 수는 학습할 신경망에 다양한 요인을 고려하여 선택하는 게 중요하다. 그렇기에 확실한 배치 수는 없고, 상황에 따라 선택을 해야합니다.

(미니배치를 6000번 했을 경우)

```
[98]: iters_num = 10000
      train_size = x_train.shape[0]
      print(train_size)
      batch_size = 6000
      learning_rate = 0.1

      train_loss_list = []
      train_acc_list = []
      test_acc_list = []
      60000

[99]: iter_per_epoch = max(train_size // batch_size, 1)
      total = 0
      print(iter_per_epoch)
      10

[100]: for i in range(iters_num):
           # 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)
           grad = network.gradient(x_batch, t_batch)

           # 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)

           # 1에폭당 정확도 계산
           if i % iter_per_epoch == 0:
               total += 1
               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(str(total) + "번" + " ---- " + "train acc, test acc | " + str(train_acc) + ", " + str(test_acc))
      980번 ---- train acc, test acc | 0.9460666666666667, 0.9481
      981번 ---- train acc, test acc | 0.94665, 0.9482
      982번 ---- train acc, test acc | 0.9466333333333333, 0.9481
      983번 ---- train acc, test acc | 0.9466166666666667, 0.9487
      984번 ---- train acc, test acc | 0.94663, 0.9485
      985번 ---- train acc, test acc | 0.9466333333333333, 0.9483
      986번 ---- train acc, test acc | 0.9467333333333333, 0.9485
      987번 ---- train acc, test acc | 0.9468166666666666, 0.9481
      988번 ---- train acc, test acc | 0.94675, 0.9483
      989번 ---- train acc, test acc | 0.9467333333333333, 0.9485
      990번 ---- train acc, test acc | 0.94685, 0.9481
      991번 ---- train acc, test acc | 0.94685, 0.9482
      992번 ---- train acc, test acc | 0.9468333333333333, 0.9484
      993번 ---- train acc, test acc | 0.9469166666666666, 0.9484
      994번 ---- train acc, test acc | 0.9470333333333333, 0.9486
      995번 ---- train acc, test acc | 0.9470333333333333, 0.9481
      1000번 ---- train acc, test acc | 0.9485666666666666, 0.9491
```

(미니배치를 10번 했을 경우)

```
[161]: (x_train, t_train), (x_test, t_test) = load_mnist(normalize=True, one_hot_label=True)

[162]: network = TwoLayerNet(input_size=784, hidden_size=50, output_size=10)

[163]: iters_num = 10000
       train_size = x_train.shape[0]
       print(train_size)
       batch_size = 10
       learning_rate = 0.1

       train_loss_list = []
       train_acc_list = []
       test_acc_list = []
       60000

[164]: iter_per_epoch = max(train_size // batch_size, 1)
       total = 0
       print(iter_per_epoch)
       6000

[165]: for i in range(iters_num):
           # 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)
           grad = network.gradient(x_batch, t_batch)

           # 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)

           # 1에폭당 정확도 계산
           if i % iter_per_epoch == 0:
               total += 1
               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(str(total) + "번" + " ---- " + "train acc, test acc | " + str(train_acc) + ", " + str(test_acc))
      1번 ---- train acc, test acc | 0.09751666666666667, 0.0974
      2번 ---- train acc, test acc | 0.9276, 0.9277
```

(미니배치를 100번 했을 경우)

```
[150]: iters_num = 10000
       train_size = x_train.shape[0]
       print(train_size)
       batch_size = 100
       learning_rate = 0.1

       train_loss_list = []
       train_acc_list = []
       test_acc_list = []
       60000

[151]: iter_per_epoch = max(train_size // batch_size, 1)
       total = 0
       print(iter_per_epoch)
       600

[152]: for i in range(iters_num):
           # 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)
           grad = network.gradient(x_batch, t_batch)

           # 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)

           # 1에폭당 정확도 계산
           if i % iter_per_epoch == 0:
               total += 1
               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(str(total) + "번" + " ---- " + "train acc, test acc | " + str(train_acc) + ", " + str(test_acc))
      1번 ---- train acc, test acc | 0.10473333333333333, 0.1033
      2번 ---- train acc, test acc | 0.79435, 0.7976
      3번 ---- train acc, test acc | 0.8761833333333333, 0.8801
      4번 ---- train acc, test acc | 0.8881833333333333, 0.9025
      5번 ---- train acc, test acc | 0.9081166666666667, 0.9113
      6번 ---- train acc, test acc | 0.9139833333333334, 0.9168
      7번 ---- train acc, test acc | 0.9190333333333334, 0.9225
      8번 ---- train acc, test acc | 0.9225166666666667, 0.9251
      9번 ---- train acc, test acc | 0.9273333333333333, 0.9286
      10번 ---- train acc, test acc | 0.93075, 0.9322
      11번 ---- train acc, test acc | 0.9339666666666666, 0.9332
      12번 ---- train acc, test acc | 0.9365833333333333, 0.9358
      13번 ---- train acc, test acc | 0.9388166666666666, 0.9386
      14번 ---- train acc, test acc | 0.9409166666666666, 0.9418
      15번 ---- train acc, test acc | 0.94355, 0.9427
      16번 ---- train acc, test acc | 0.9454666666666667, 0.9448
      17번 ---- train acc, test acc | 0.9474, 0.9456
```

질문 2. 신경망의 깊이 vs 너비, 깊은 네트워크와 넓은 네트워크의 차이점은 무엇이며 어떤 문제에 어떤 구조가 적합할까?

안정빈 : 신경망의 깊이는 층의 수, 너비는 한 층의 뉴런의 수를 의미해. 깊은 네트워크는 우선 비선형이고, 구조는 단순해 보일 것 같지만 계산량은 많은 것 같아. 넓은 네트워크는 가중치가 늘어나니까 관계가 복잡할 것 같아.

김상옥 : 너비가 크다는 건 고려할 요인이 많다는 것 같아. 너비가 뉴런의 수이니 어떤 문제에 대해서 더 고려할 요인들(가중치)이 많으며, 그 말은 행렬의 크기가 크다는 의미이니 어떤 문제의 크기를 나타내는 게 아닐까. 그리고 깊이가 깊다는 건 그만큼 복잡하고 어려운 문제를 나타내는 것 같아. 각 활성화 함수를 통해 다음 층으로 넘어가는데 그 층이 많다는 건 그만큼 복잡하고 어려운 문제를 해결하기 위해 계산한다고 생각해.

안정빈 : 깊은 네트워크는 층이 많아지니까 복잡한 패턴을 학습하는데 사용 될 것 같아. 적은 매개변수로 복잡한 함수를 표현 할 것 같은데 예를들어 이미지, 오디오등 복잡한 처리 에 사용될 것 같고, 넓은 네트워크는 각 층에서 많은 뉴런을 가지고 많은 매개변수가 있어서 더 복잡한 함수를 표현 하는데 자연어처리, SNS 의 대규모 데이터 베이스 같은 대용량 처리를 다룰때에도 유용 할 것같아.

김상옥 : 그 말대로 문제에 따라 신경망의 너비와 깊이를 설정해야 좋은 신경망이겠다. 처리할 양이 많은 문제라면 너비를 늘리고, 복잡한 문제라면 깊이를 늘리는 식으로 신경망을 구성한다면 효율적인 신경망을 만들 수 있을 거 같아.

안정빈 : 나도 그렇게 생각해. 따라서 깊이와 너비는 모델의 복잡성을 결정하는 중요한 요소 같아, 적절한 깊이와 너비를 선택해서 모델이 원하는 작업을 수행할수록 있도록 하는 것이 중요하다고 생각해.

결론 : 깊이는 복잡한 것을 해결할 때 사용되고, 너비는 해결한 문제의 양이 많을 때 적합합니다. 문제에 따라서 깊이와 너비를 적절하게 설정하여 신경망을 설계하는게 좋다고 생각합니다.

4 장

(안정빈)

```python
import numpy as np
# y: 예측값, 추정값 t: 라벨링(정답, 참 값)
def mean_squared_error(y, t):
    return 0.5 * np.sum((y-t)**2)

#모델 변경하고, 가중치 값 조절하며 오차값을 줄이며 테스트함
```

```python
#정답은 '2'
t=[0, 0, 1, 0, 0, 0, 0, 0, 0, 0]

# 예 1 : '2' 일 확률이 가장 높다고 추정함(0.6)
# y 값이 합쳐서 1.0이 되어야함 (소프트맥스)
y= [0.1, 0.05, 0.6, 0.0, 0.05, 0.1, 0.0, 0.1, 0.0, 0.0]
mean_squared_error(np.array(y), np.array(t))
#-> 오차가 더 적어서 정답에 가까움
```

```
0.09750000000000003
```

```python
#예 2 : '7' 일 확률이 가장 높다고 추정함(0.6)
y= [0.1, 0.05, 0.1, 0.0, 0.05, 0.1, 0.0, 0.6, 0.0 , 0.0]
mean_squared_error(np.array(y), np.array(t))
```

```
0.5975
```

```python
import numpy as np
# y: 예측값, 추정값 t: 라벨링(정답, 참 값)

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

```python
#정답은 '2'
t=[0, 0, 1, 0, 0, 0, 0, 0, 0, 0]

# 예 1 : '2' 일 확률이 가장 높다고 추정함(0.6)
# y 값이 합쳐서 1.0이 되어야함 (소프트맥스)
y= [0.1, 0.05, 0.6, 0.0, 0.05, 0.1, 0.0, 0.1, 0.0, 0.0]
cross_entropy_err(np.array(y), np.array(t))
```

```
0.510825457099338
```

```python
#예 2 : '7' 일 확률이 가장 높다고 추정함(0.6)
y= [0.1, 0.05, 0.1, 0.0, 0.05, 0.1, 0.0, 0.6, 0.0 , 0.0]
cross_entropy_err(np.array(y), np.array(t))
```

```
2.302584092994546
```

```python
import sys, os
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)

print(x_train.shape)
print(t_train.shape)
```

```
(60000, 784)
(60000, 10)
```

```python
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(x_batch.shape)
print(t_batch.shape)
```

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

```python
np.random.choice(60000, 10)
```

```
array([16765, 55936, 42626, 27598, 32974, 43970, 58891, 52543, 59164,
        5001])
```

```python
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
```

```python
import numpy as np
import matplotlib.pylab as plt
```

미분을 파이썬으로 구현

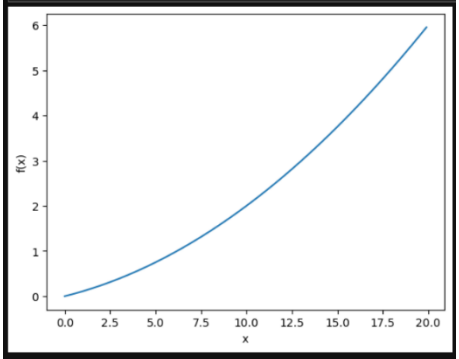
```python
def numerical_diff(f,x):
    h=10e-50
    return(f(x+h) - f(x))/ h
```

개선한 수치 미분 함수 (중심차분) , 오차 줄임

```python
def numerical_diff2(f,x):
    h = 1e-4
    return (f(x+h) - f(x-h)) / (2*h)
```

```python
def function_1(x):
    return 0.01*x**2 + 0.1*x
```

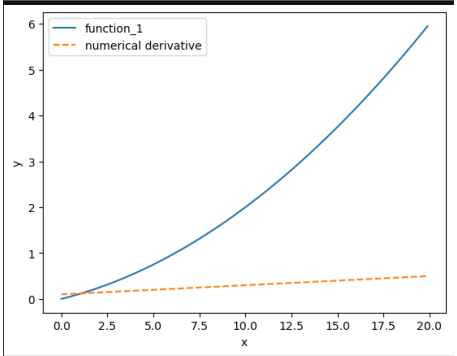
```python
x = np.arange(0.0, 20.0, 0.1)
y= function_1(x)
plt.xlabel("x")
plt.ylabel("f(x)")
plt.plot(x, y)
plt.show()
```




```python
numerical_diff2(function_1,5)
```

```
0.1999999999990898
```

```python
dy = numerical_diff2(function_1, x)

plt.figure()
plt.plot(x, y, label='function_1')
plt.plot(x, dy, label='numerical derivative', linestyle='--')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.show()
```




```python
numerical_diff2(function_1,10)
```

```
0.2999999999986347
```

편미분

```python
def partial_diff(x):
    return x[0]**2 + x[1]**2
```

```python
def numerical_gradient(f, x):
    h= 1e-4 # 0.0001
    grad = np.zeros_like(x) # x와 형상이 같은 배열을 생성

    for idx in range(x.size):
        tmp_val = x[idx]

        #f(x+h) 계산
        x[idx] = float(tmp_val) +h
        fxh1 = f(x)

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

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

    return grad
```

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

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

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

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

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

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

```python
#경사 하강법 구현
```

```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
```

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

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

```python
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
```

```python
net = simpleNet()
print(net.W)
```

```
[[-0.99410894  0.82344361 -0.63989023]
 [ 1.57474657 -0.45575843  0.00442616]]
```

```python
x = np.array([0.6, 0.9])
P= net.predict(x)
print(p)
```

```
[-0.02916385  1.24607407 -0.99265148]
```

```python
np.argmax(p) #최댓값의 인덱스
```

```
1
```

```python
t= np.array([0,0,1]) # 정답레이블
net.loss(x,t)
```

```
1.777117368406503
```

```python
def f(W):
    return net.loss(x,t)
```

```python
dW = numerical_gradient(f, net.W)
print(dW)
```

```
[[ 0.33716359  0.16136119 -0.49852478]
 [ 0.50574539  0.24204179 -0.74778717]]
```

```python
f = lambda w: net.loss(x, t)
dW = numerical_gradient(f, net.W)
print(dW)
```

```
[[ 0.33716359  0.16136119 -0.49852478]
 [ 0.50574539  0.24204179 -0.74778717]]
```

```python
import sys, os
sys.path.append(os.pardir)
import numpy as np
import matplotlib.pyplot as plt
from dataset.mnist import load_mnist
from two_layer_net import TwoLayerNet
```

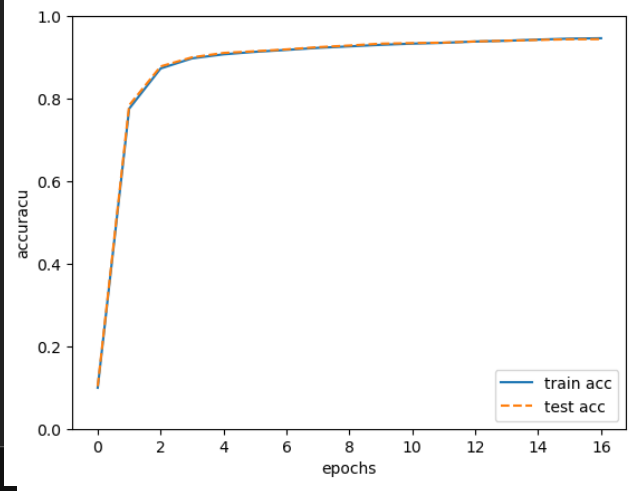
```python
(x_train, t_train), (x_test, t_test) = load_mnist(normalize=True, one_hot_label=True)

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

```python
iters_num = 10000
train_size = x_train.shape[0]
batch_size = 100
learning_rate = 0.1
```

```python
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.numerical_gradient(x_batch, t_batch)
    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(i,"번째 acc |", str(train_acc) , ", " , str(test_acc))
```

```
0 번째 acc | 0.0993 ,  0.1032
600 번째 acc | 0.7756833333333333 ,  0.7839
1200 번째 acc | 0.8731666666666666 ,  0.8782
1800 번째 acc | 0.8977166666666667 ,  0.9002
2400 번째 acc | 0.9074166666666666 ,  0.9108
3000 번째 acc | 0.9134 ,  0.9149
3600 번째 acc | 0.9181 ,  0.9194
4200 번째 acc | 0.9230666666666667 ,  0.9248
4800 번째 acc | 0.9266666666666666 ,  0.9288
5400 번째 acc | 0.9304333333333333 ,  0.9333
6000 번째 acc | 0.9333166666666667 ,  0.9346
6600 번째 acc | 0.9353 ,  0.9358
7200 번째 acc | 0.9384666666666667 ,  0.9386
7800 번째 acc | 0.94035 ,  0.9404
8400 번째 acc | 0.9431166666666667 ,  0.9423
9000 번째 acc | 0.94545 ,  0.9439
9600 번째 acc | 0.9464333333333333 ,  0.9442

오버 피팅 발생 x
```

```python
#그래프 그리기
markers ={'train':'0', '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("accuracu")
plt.ylim(0, 1.0)
plt.legend(loc='lower right')
plt.show()
```



(김상옥)

학습 알고리즘 구현

- 미니배치
- 신경망 학습
- 학습 데이터 평가

전체적인 학습 과정

1. 학습에 맞게 신경망을 구성하며, 가중치는 랜덤으로, 편향은 0으로 설정한다.
2. 학습 데이터를 랜덤하게 선별하여 미니배치를 준비한다 -> **미니배치**
3. 미니배치 데이터를 신경망에 넣어 계산한다. -> **sigmoid, softmax**
4. 출력값의 손실 함수를 구한다 -> **평균 제곱 오차, 교차 엔트로피 오차**
5. 손실 함수를 바탕으로 수치 미분 or 오차역전파를 수행한다 -> **중앙 차분, 편미분**
6. 기울기를 구하고 가중치를 설정한다 -> **경사하강법, 학습률, 기울기**
7. 2번~6번의 결과를 반복이 끝날 때 까지 수행한다.

필요 라이브러리, 데이터셋 import

[64]:
```
import numpy as np
import matplotlib.pylab as plt
import sys, os
sys.path.append(os.pardir)

from dataset.mnist import load_mnist
```

--사용 함수--

활성화 함수

- 시그모이드
- 소프트맥스 (크기에 맞게 변경)

손실 함수

- 교차 엔트로피 오차

기울기

- 편미분, 수치 미분 --> 기울기

[65]:
```
def sigmoid(x):
    y = 1 / (1 + np.exp(-x))
    return y
```

[66]:
```
def softmax(x):
    if x.ndim == 2:
        x = x.T
        x = x - np.max(x, axis=0)
        y = np.exp(x) / np.sum(np.exp(x), axis=0)
        return y.T

    x = x - np.max(x)
    return np.exp(x) / np.sum(np.exp(x))
```

[67]:
```
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)

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

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

    return grad
```

[68]:
```
def cross_entropy_error(y, t):
    if y.ndim == 1:
        t = t.reshape(1, t.size)
        y = y.reshape(1, y.size)

    if t.size == y.size:
        t = t.argmax(axis=1)

    batch_size = y.shape[0]
    return -np.sum(np.log(y[np.arange(batch_size), t] + 1e-7)) / batch_size
```

학습에 사용될 클래스

[69]:
```
class TwoLayerNet:

    # 가중치 초기화
    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)


    # 가중치와 편향으로 계산 신경망 계산
    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


    # 교차 엔트로피 오차 손실 함수를 구한다
    def loss(self, x, t):
        # x : 입력 데이터, t : 정답 레이블
        y = self.predict(x)
        return cross_entropy_error(y, t)


    # 정확도 구하기
    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


    # 수치 미분으로 기울기 구하기
    def numerical_gradient(self, x, t):
        # x : 입력 데이터, t : 정답 레이블
        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
```

미니배치 학습

```
[70]: (x_train, t_train), (x_test, t_test) = load_mnist(normalize=True, one_hot_label=True)
```

```
[71]: network = TwoLayerNet(input_size=784, hidden_size=50, output_size=10)
```

```
[72]: iters_num = 10000
train_size = x_train.shape[0]
print(train_size)
batch_size = 100
learning_rate = 0.1

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

```
60000
```

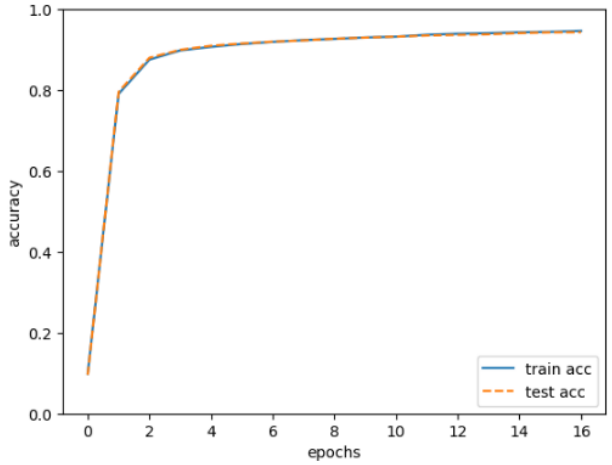
```
[73]: iter_per_epoch = max(train_size // batch_size, 1)
print(iter_per_epoch)
```

```
600
```

```
[74]: for i in range(iters_num):
    # 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)
    grad = network.gradient(x_batch, t_batch)

    # 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)

    # 1에폭당 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))
```

```
train acc, test acc | 0.09863333333333334, 0.0958
train acc, test acc | 0.7907166666666666, 0.7958
train acc, test acc | 0.8755666666666667, 0.8801
train acc, test acc | 0.8982833333333333, 0.8998
train acc, test acc | 0.9070666666666667, 0.9099
train acc, test acc | 0.9141666666666667, 0.9153
train acc, test acc | 0.9194166666666667, 0.9199
train acc, test acc | 0.9236333333333333, 0.923
train acc, test acc | 0.9268333333333333, 0.9273
train acc, test acc | 0.9303166666666667, 0.93
train acc, test acc | 0.9322333333333334, 0.9324
train acc, test acc | 0.93715, 0.9352
train acc, test acc | 0.9395333333333333, 0.9367
train acc, test acc | 0.94095, 0.9381
train acc, test acc | 0.94315, 0.9412
train acc, test acc | 0.94425, 0.943
```

```
[75]: 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()
```



신경망 학습

- 손실 함수 (위에 작성)
- 미니 배치 학습

손실 함수

- 오차제곱합 (평균 제곱 오차)
- 교차 엔트로피 오차

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

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

미니 배치 학습

```
[37]: import numpy as np
import matplotlib.pylab as plt
import sys, os, pickle
from PIL import Image
sys.path.append(os.pardir)

from dataset.mnist import load_mnist
(x_train, t_train), (x_text, t_test) = load_mnist(normalize=True, one_hot_label=True)
```

```
[38]: train_size = x_train.shape[0]
batch_size = 10
batch_mask = np.random.choice(train_size, batch_size) # np.random.choice(number, amount) number에서 amount만큼 랜덤 뽑기
x_batch = x_train[batch_mask]
t_batch = t_train[batch_mask]
print(x_batch)
print("----------------------------------------------------------------------------------------")
print(t_batch)
```

```
[[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]
----------------------------------------------------------------------------------------
[[1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]]
```

```
[39]: def cross_entropy_error2(y, t):
    if y.ndim == 1:
        t = t.reshape(1, t.size)
        y = y.reshape(1, y.size)

    delta = 1e-7
    batch_size = y.shape[0]
    return -np.sum(t * np.log(y + delta)) / batch_size
```

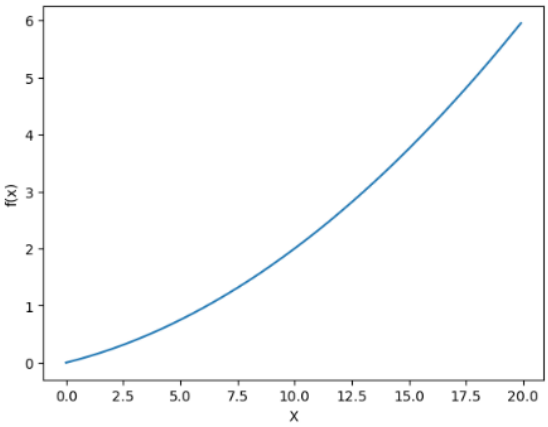
수치 미분

```
[140]: import numpy as np
import matplotlib.pylab as plt
```

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

```
[142]: def function_1(x):
    return 0.01*x**2 + 0.1*x
```

```
[143]: x = np.arange(0., 20, 0.1)
y = function_1(x)
plt.xlabel("X")
plt.ylabel("f(x)")
plt.plot(x, y)
plt.show()
```




```
[144]: print(numerical_diff(function_1, -10))
```

```
-0.0999999999995449
```

```
[145]: print(numerical_diff(function_1, 10))
```

```
0.2999999999986347
```

편미분

+ 6 cells hidden

기울기

기울기를 통해 올바른 학습 방향을 설정한다

[171]:
```
def function_2(x):
    return x[0]**2 + x[1]**2
```

[172]:
```
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)

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

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

    return grad
```

[173]:
```
numerical_gradient(function_2, np.array([3.0, 4.0]))
```

[173]:
```
array([6., 8.])
```

[174]:
```
numerical_gradient(function_2, np.array([0.0, 2.0]))
```

[174]:
```
array([0., 4.])
```

[175]:
```
numerical_gradient(function_2, np.array([3.0, 0.0]))
```

[175]:
```
array([6., 0.])
```

[176]:
```
numerical_gradient(function_2, np.array([0.0, 100.0]))
```

[176]:
```
array([  0.        , 200.00000001])
```

경사 하강법 ▽

[179]:
```
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
```

미분

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

수치 미분

중앙 차분으로 기존 미분보다 정확한 결과를 보여준다.

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

편미분

연산에 미지수가 2개 이상일 때 하나의 x값에 초점을 두어 미분을 구한다. 나머지 값은 고정하고, 초점을 둔 x의 변화 따른 변화량만 구하는 작업이다.

기울기

편미분을 벡터로 정리한 것이 기울기이다.
각 x에 대해 편미분을 구한 값이 기울기다. 기울기가 의미하는 것은 기울기의 방향이 출력값을 가장 크게 줄이는 방향을 의미한다. 그러나 기울기가 늘 최소값을 가리키는 건 아니나 조금이라도 작은 수를 가리킨다.

경사법

경사법은 기울기를 통해 학습을 하는 방법이다. 현재 기울기의 어느정도 앞에가고, 그 위치에서 또 기울기를 구하는걸 반복하며 함수의 값을 줄인다.

경사 하강법

경사법으로 최소값을 찾는 과정이다. 반대로 상승법은 최대값을 찾는다.

전체적인 학습 과정

1. 학습에 맞게 신경망을 구성하며, 가중치는 랜덤으로, 편향은 0으로 설정한다.
2. 학습 데이터를 랜덤하게 선별하여 미니배치를 준비한다 -> **미니배치**
3. 미니배치 데이터를 신경망에 넣어 계산한다. -> **sigmoid, softmax**
4. 출력값의 손실 함수를 구한다 -> **평균 제곱 오차, 교차 엔트로피 오차**
5. 손실 함수를 바탕으로 수치 미분 or 오차역전파를 수행한다 -> **중앙 차분, 편미분**
6. 기울기를 구하고 가중치를 설정한다 -> **경사하강법, 학습률, 기울기**
7. 2번~6번의 결과를 반복이 끝날 때 까지 수행한다.

4장 신경망 학습

미니배치

미니배치는 전체적인 학습 데이터에서 작은 그룹을 뽑아 그 반복적으로 학습을 하는 과정이다. 학습 데이터에서 골고루 뽑아 범용성있는 성능을 올리는 것이 중요하다. 한쪽만 치우쳐진 학습 결과는 편향적으로 결과를 내기에 다양하게 학습 데이터를 골라내는 것이 중요하다.

미니배치와 손실 함수

일부 골라낸 데이터의 손실 함수를 구하고, 손실 함수의 합을 통해 평균을 구하면 하나의 지표가 된다.

훈련 데이터와 시험 데이터

이 두 데이터를 나누는 이유는 범용성을 위해서다. 범용성이란 학습한 데이터가 아닌 새로운 데이터에서도 올바른 추론을 하는 능력으로 학습 이후 시험 데이터를 통해 범용성을 테스트한다.

오버피팅

학습이 편향되어 범용성이 떨어진 학습을 의미한다

손실 함수

손실 함수는 학습이 올바른지 검사하는 지표이다. 기본적으로 0에 가까우면 올바른 추론이며, 반대로 멀어진다면 오답에 가까운 추론이다.

손실 함수를 지표로 되는 이유

정확도를 지표로 삼으면 대부분의 구간에서 미분이 0이 되기에 학습이 안된다.

오차제곱합

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

교차 엔트로피 오차

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

1e-7을 y에 더하는 이유는 0이 되어버리면 미분을 할 수 없기 때문이다.

미분의 기능

학습에 있어 미분은 상당히 중요한 역할을 한다.
미분이란 한 순간의 변화량을 나타내는데 이는 학습에 결과가 어떤 방향으로 나아갈 지 정하는 수치가 된다. 손실 함수에 따라 올바른 학습의 기울기를 정하는데 미분이 사용된다.