인공지능개론 정리노트 #2

ICT융합공학부 202204010 공성택

4장

Q. 손실 함수는 데이터에 대한 '예측과 결과의 차이'라는 하나의 목적을 나타내는 지표인데, 굳이 손실 함수의 종류가 여러 개인 이유는 무엇인가?

A. 손실 함수의 종류가 여러 개인 이유는 신경망에게 주어지는 문제의 유형이 다르기 때문이다. 단순히 "데이터에 대한 예측과 결과의 차이"라는 정의 때문에 나는 이 개념에 대해서 "예측과 결과의 차이라는 목적이 같으면 하나의 식으로도 되지 않나?"라는 오해를 했지만, 주어지는 문제의 유형이 다르기 때문에 상황에 맞는 여러 손실 함수식을 사용해야 한다. 기계학습, 딥러닝에서 주어지는 대표적인 문제 유형은 크게 회귀 문제와 분류 문제로 나뉜다. 회귀 문제는 연속적인 값을 예측하는 문제이다. 예를 들면, 주식 예측 등이 있다. 이런 문제에서는 주로 평균 제곱 오차를 사용한다. 반면에 분류 문제는 주어진 데이터가 어디에 포함되는지 예측하는 문제이다. 예를 들면, 강의 자료에 있는 것처럼 손글씨를 인식하고, 아니면 사진을 보고 해당 사진이 무엇인지 인식하는 문제 등이 있다. 분류 문제에서는 회귀 문제와 달리 교차 엔트로피 오차를 사용한다. 이처럼 주어지는 문제 유형이 다르기 때문에 "예측과 결과의 차이를 도출한다"는 목표는 같더라도 활용되는 손실 함수의 종류가 나뉘는 것이다.

Q. 무작위로 미니배치를 뽑을 때, 추려진 미니배치가 전체 데이터셋에 대해서 골고루 나타난 것이 아닌 편향된 데이터셋일 가능성이 있다. 그러면 미니배치를 사용한 학습에 대해 신뢰할 수 있는가? 그렇다면 미니배치를 통한 학습을 사용하는 것이 옳은 것인가?

A. 미니배치를 사용하는 이유는 전체 데이터를 사용했을 때보다 소요 시간이 매우 줄어들고, 하드웨어의 성능 등 제한된 환경에서 실행해야 하기 때문에 최대한 효율적으로 데이터를 학습하는 타협점이기 때문이다. 실제로 미니 배치를 사용하는 것이 효율적인 부분이 많다. 그러나 데이터셋에서 랜덤으로 미니배치를 뽑았을 때, 미니배치가 전체 데이터셋의 특성, 분포 등을 완전히 대표하지 못할 수 있는 것은 사실이다. 극단적으로 생각한다면, 6만개의 데이터 중 100개를 뽑을 때, 한쪽에 치중된 데이터가 100개가 나올 수도 있기 때문이다. 하지만 조사한 결과, 그럴 확률은 극히 낮고, 그런 문제들을 완화하기 위해서 임의로 훈련 데이터를 조정하거나, 미니배치를 뽑을 때 완전 무작위가 아닌 골고루 선택하여 뽑는 방법을 사용할 수 있다고 한다. 결국 하드웨어의 부담과 학습 소요 시간의 효율과 같은 현실적 문제를 타협하기 위해, 위와 같은 여러 방면을 고려해 미니배치 학습을 사용한다면 더 효율적으로 기계학습을 할 수 있다고 결론지었다.

4장-1 수업시간 실습(수업 시간에 직접 실습했습니다.)

[25]:
```
import numpy as np

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

[26]:
```
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
```

[27]:
```
def numerical_diff(f, x):
    h = 1e-4
    return (f(x+h) - f(x-h)) / (2*h)
```

[28]:
```
def function_1(x):
    return 0.01*x**2 + 0.1*x
```

[29]:
```
numerical_diff(function_1, 5)
```

[29]:
```
0.1999999999990898
```

[30]:
```
numerical_diff(function_1, 10)
```

[30]:
```
0.2999999999986347
```

[31]:
```
def partial_diff(x):
    return x[0]**2 + x[1]**2
```

[35]:
```
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] = float(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
```

[36]:
```
numerical_gradient(partial_diff, np.array([3.0, 4.0]))
```

[36]:
```
array([6., 8.])
```

[2]:
```
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)
```

```
Downloading train-images-idx3-ubyte.gz ...
Done
Downloading train-labels-idx1-ubyte.gz ...
Done
Downloading t10k-images-idx3-ubyte.gz ...
Done
Downloading t10k-labels-idx1-ubyte.gz ...
Done
Converting train-images-idx3-ubyte.gz to NumPy Array ...
Done
Converting train-labels-idx1-ubyte.gz to NumPy Array ...
Done
Converting t10k-images-idx3-ubyte.gz to NumPy Array ...
Done
Converting t10k-labels-idx1-ubyte.gz to NumPy Array ...
Done
Creating pickle file ...
Done!
(60000, 784)
(60000, 10)
```

[3]:
```
np.random.choice(60000, 10)
```

[3]:
```
array([34833, 15107,  6991, 49282, 33330, 40915, 37120, 36337, 28750,
       20241])
```

[4]:
```
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)
```

[19]:
```
t = [0,0,1,0,0,0,0,0,0,0]
y = [0.1, 0.05, 0.6, 0.0, 0.05, 0.1, 0.0, 0.1, 0.0, 0.0]
```

[20]:
```
mean_squared_error(np.array(y), np.array(t))
```

[20]:
```
0.09750000000000003
```

[21]:
```
cross_entropy_error(np.array(y), np.array(t))
```

[21]:
```
0.510825457099338
```

[22]:
```
y1 = [0.1, 0.05, 0.0, 0.0, 0.05, 0.1, 0.6, 0.1, 0.0, 0.0]
```

[23]:
```
mean_squared_error(np.array(y1), np.array(t))
```

[23]:
```
0.6975
```

[24]:
```
cross_entropy_error(np.array(y1), np.array(t))
```

[24]:
```
16.11809565095832
```

[ ]:

[19]:
```
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
```

[20]:
```
init_x = np.array([-3.0, 4.0])
gradient_descent(partial_diff, init_x=init_x, lr=0.1, step_num=100)
```

[20]:
```
array([-6.11110793e-10,  8.14814391e-10])
```

[23]:
```
import sys, os
sys.path.append(os.pardir)
import numpy as np
from common.functions import softmax, cross_entropy_error
from common.gradient import numerical_gradient


class simpleNet:
    def __init__(self):
        self.W = np.random.randn(2,3)

    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

net = simpleNet()
x = np.array([0.6, 0.9])
p = net.predict(x)
np.argmax(p)
t = np.array([0, 0, 1])
net.loss(x,t)

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

dW = numerical_gradient(f, net.W)

print(dW)
```

```
[[ 0.13850412  0.36338124 -0.50188536]
 [ 0.20775618  0.54507186 -0.75282804]]
```

4장-2 학습 알고리즘 구현(동영상 강의와 교재 소스 참고하며 실습했습니다.)

```python
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):
        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(z1, W2) + b2
        y = softmax(a2)

        return y

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

        return 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
```

```
    def gradient(self, x, t):
        W1, W2 = self.params['W1'], self.params['W2']
        b1, b2 = self.params['b1'], self.params['b2']
        grads = {}

        batch_num = x.shape[0]

        a1 = np.dot(x, W1) + b1
        z1 = sigmoid(a1)
        a2 = np.dot(z1, W2) + b2
        y = softmax(a2)

        dy = (y - t) / batch_num
        grads['W2'] = np.dot(z1.T, dy)
        grads['b2'] = np.sum(dy, axis=0)

        da1 = np.dot(dy, W2.T)
        dz1 = sigmoid_grad(a1) * da1
        grads['W1'] = np.dot(x.T, dz1)
        grads['b1'] = np.sum(dz1, axis=0)

        return grads
```

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

(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 | " + str(train_acc) + ", " + str(test_acc))

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()
```

```
train acc, test acc | 0.09863333333333334, 0.0958
train acc, test acc | 0.7873, 0.7905
train acc, test acc | 0.8744666666666666, 0.8778
train acc, test acc | 0.8983833333333333, 0.9019
train acc, test acc | 0.9067666666666667, 0.9099
train acc, test acc | 0.9138166666666667, 0.9163
train acc, test acc | 0.9195333333333333, 0.9202
train acc, test acc | 0.9244666666666667, 0.9248
train acc, test acc | 0.9273666666666667, 0.9291
train acc, test acc | 0.9306333333333333, 0.9305
train acc, test acc | 0.9339833333333334, 0.934
train acc, test acc | 0.9368, 0.9361
train acc, test acc | 0.9394333333333333, 0.938
train acc, test acc | 0.9411833333333334, 0.9408
train acc, test acc | 0.94435, 0.9434
train acc, test acc | 0.9460833333333334, 0.9455
train acc, test acc | 0.9479833333333333, 0.9455
```