인공지능프로그래밍

Lab2

DNN

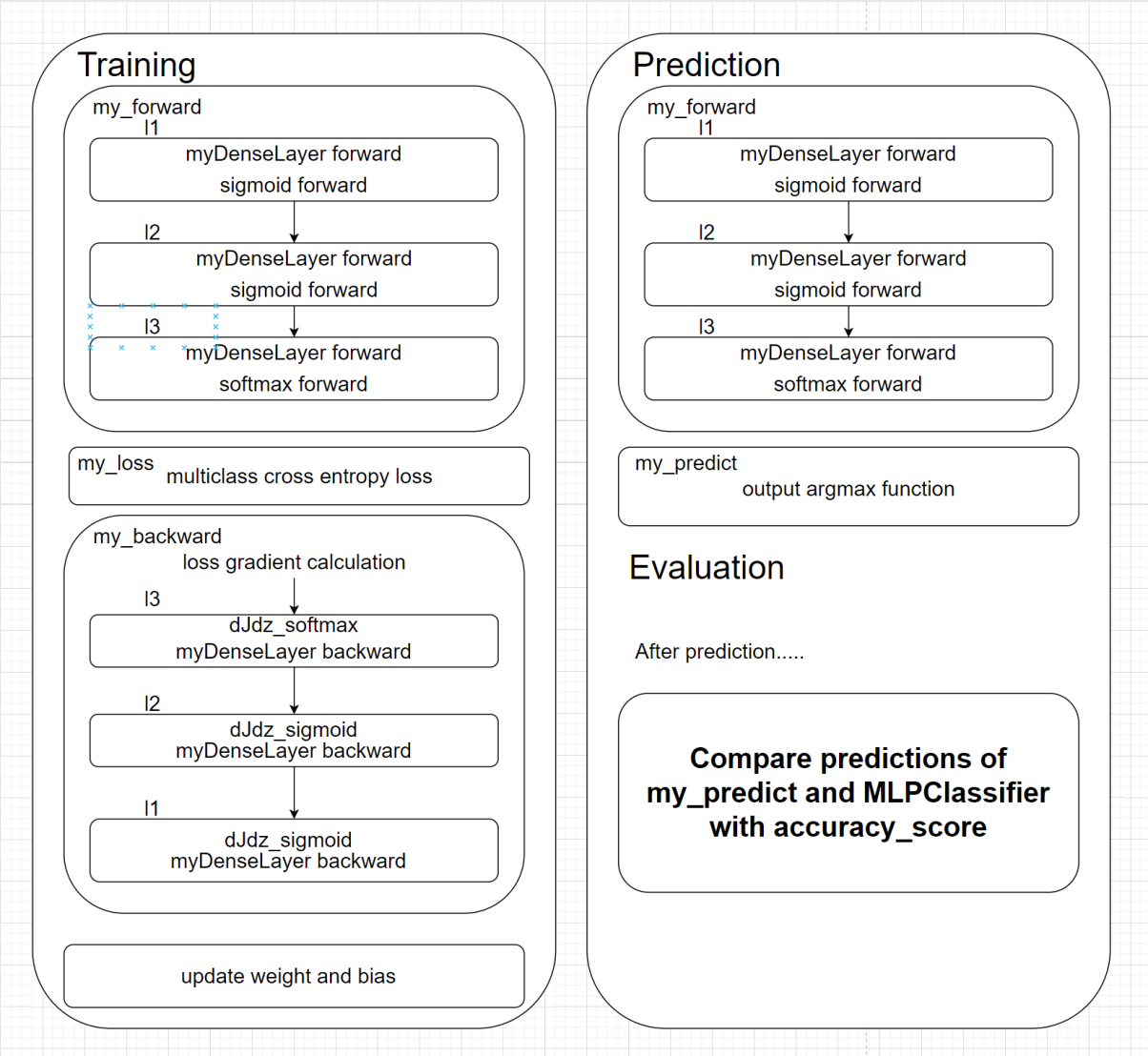
2024/09/30

2019202050 이강현

Lab Objective

MNIST Digit dataset을 이용하여 직접 구현한 DNN과 sklearn에서 제공하는 MLPClassifier와 classification 성능을 비교해본다. sigmoid과 softmax를 구현하면서 overflow와 같은 예외를 해결해보며 입력 데이터에 따른 수식의 변화를 이해한다. Forward와 Backward를 구현하면서 chain rule에 의해 모델이 학습되는 것을 이해하고 각 단계에서 propagation을 올바르게 구현하면서 DNN의 전체적인 프로세스에 대해 익힌다.

Program Flow



위 그림은 전체 프로세스의 흐름을 도식화한 것이다.

Model training에서 forward와 backward 연산을 코드에서 구현한 함수의 이름인 my_forward, my_backward, my_loss들로 표현하였고 그 내부의 layer들에서 이루어지는 연산들을 표현했다.
prediction에서는 forward 연산 후 모델의 최종 레이어 softmax 계층에서 나온 값에서 가장 큰 값을 가진 인덱스를 뽑아내는 argmax함수를 이용해 라벨을 예측한다.

마지막으로 Evaluation에서는 prediction에서 예측한 값을 이용하여 sklearn에서 제공하는 MLPClassifier와 정확도를 비교하는 흐름으로 코드가 진행된다.

Result

코드의 주요 부분과 함께 결과를 살펴보자.

```
def mySigmoid(x):
    ### START CODE HERE ###

    positive = (x>=0)                           # boolean array of positive numbers
    x_p = x[positive]                           # array of positive x
    x_n = x[~positive]                          # array of negative x
    x[positive] = 1/(1+np.exp(-x_p))            # apply sigmoid function for x_p
    x[~positive] = np.exp(x_n)/(1+np.exp(x_n))  # apply sigmoid function for x_n

    ### END CODE HERE ###
    return x
```

```
mySigmoid(np.array([0.0, 1000.0, -1000.0]))
```

```
array([0.5, 1. , 0. ])
```

먼저 mySigmoid라는 함수이다.

Sigmoid의 원형은 x의 값이 음수이고 절댓값이 매우 클 때 분모가 너무 커져 overflow가 일어날 가능성이 높아진다. 따라서 분자와 분모에 exponent항을 곱해준 수식을 사용하여 x가 양수일 때, 음수일 때 따로 연산을 진행하게끔 함수를 구현한다. 결과를 보면 올바른 결과를 확인할 수 있다. 이 함수는 DNN의 1번 layer와 2번 layer의 activation function으로 사용된다.

```
# define softmax. Assume (b, s)
def mySoftmax(x):
    ### START CODE HERE ###
    x = x - np.max(x, axis=-1,keepdims=True)        # make x sufficiently small
    x = np.exp(x)                                   # execute exponential function
    x = x / np.sum(x, axis=-1,keepdims=True)        # calculate softmax

    ### END CODE HERE ###
    return x
```

```
mySoftmax(np.array([0.0, 1000.0, -1000.0]))
```

```
array([0., 1., 0.])
```

mySoftmax 함수이다.

softmax함수는 입력값이 매우 클때 exponent항이 커져 overflow를 야기할 수 있다. 따라서 max값을 데이터에서 뺀 후 수식을 구현하면 overflow을 막으면서 올바른 값을 구할 수 있다. Max와 Sum 연산을 할 때 마지막 축에서 연산을 하도록 구현하여 여러 샘플이 입력으로 들어왔을 때 샘플마다 max값을 얻어낼 수 있도록 하였다.

```
  def forward(self, x):       # (b, i)
      ### START CODE HERE ###

      self.saved_x = x     # keep it for backward
      x_lin = (self.wegt @ x.T).T + self.bias           # Linear Prediction

      ### END CODE HERE ###
      return x_lin

  def backward(self, x, x_in):  # x = dJ/dz (b, c)
      assert np.array_equal(self.saved_x, x_in), print('x_in does not equal to input X.')
      ### START CODE HERE ###

      dw = (x.T @ x_in)            # Gradients for weights
      db = np.sum(x,axis=0)        # Gradients for biases
      wdJdz = x @ self.wegt        # Propagation for lower layer

      ### END CODE HERE ###
      return dw, db, wdJdz
```

layer에서 사용하는 forward와 backward 함수이다.

forward에서 weight와 입력값을 matrix multiplication한 후 bias를 더하는 수식이

구현되어 있다. 적절하게 transpose를 하여 다차원의 텐서가 들어왔을때도 적용이 가능하게 구현하였다. 또 backward에서 사용하기 위해 saved_x에 현재 입력값을 저장해둔다.

backward에는 먼저 forward시 입력값으로 들어온 saved_x와 backward시 weight 업데이트를 위해 넣어준 x_in이 같은지부터 검사를 진행한다. 이 두 값은 서로 동일해야 하지만 다르다면 올바른 weight 업데이트가 진행되지 않기에 넣어둔 예외처리라고 볼 수 있다. 각각 dJdw, dJdb, wdJdz를 구한다. 먼저 dJdw는 이전 레이어에서 전달된 dJdz(x)와 dzdw(x_in)를 내적하여 구한다. Chain rule에 의해 dJdw를 구할 수 있고 이는 dw에 저장된다.

다음은 dJdz(x)와 dzdb(1)를 내적하여 dJdb를 구하고 db에 저장한다.

마지막으로 다음 레이어의 backward에서 사용할 dJda를 구한다. 따라서 dJdz(x)와 dzda(self.wegt)를 내적하여 wdJdz에 곱한다. 2차원에서 @와 np.dot은 동일한 연산을 진행하므로 입력값이 (sample수 x 1차원으로 펼친 사진)임을 알기에 그림과 같이 구현하였다.

```
np.random.seed(0)

tmp = myDenseLayer(3,5)
tmp.wegt = np.random.randn(3,5)
tmp.bias = np.random.randn(3)

print(tmp.forward(np.random.randn(2,5,3)))
```

```
[[[ 3.23890168  3.05091188 -3.32627831]
  [ 0.388114    3.36724875  1.06158492]
  [ 3.10267869  1.87570497 -1.8326582 ]]

 [[-7.60581826  2.36703751 -1.16423539]
  [ 3.48035012  2.41940644 -0.13917734]
  [ 1.20541315  2.07585619 -1.5435161 ]]]
```

Expected Outputs

```
[[[ 3.23890168  3.05091188 -3.32627831]
  [ 0.388114    3.36724875  1.06158492]
  [ 3.10267869  1.87570497 -1.8326582 ]]

 [[-7.60581826  2.36703751 -1.16423539]
  [ 3.48035012  2.41940644 -0.13917734]
  [ 1.20541315  2.07585619 -1.5435161 ]]]
```

위 그림을 보면 예상 output과 동일한 결과를 얻고 forward가 잘 구현된 것을 확인한다. 또한 다차원의 입력에서도 forward 연산이 잘 진행됨을 확인한다.

```python
def dJdz_sigmoid(wdJdz_upper, az):
    ### START CODE HERE ###

    dJdz = wdJdz_upper * (1.0-az) * az            # backpropagation through activation function

    ### END CODE HERE ###
    return dJdz

def dJdz_softmax(y_hat, y):
    ### START CODE HERE ###

    dJdz = (y_hat - y)            # backpropagation through activation function

    ### END CODE HERE ###
    return dJdz
```

위 그림은 layer의 backward연산 후 activation function의 미분값을 곱해주기 위한 함수이다. wdJdz_upper는 이전 layer의 backward에서 구한 wdJdz로 dJda값이고 이것과 dadz(a′(z))를 이용해 dJdz를 얻어내는 것을 확인한다. 이때 activation function이 sigmoid일때와 softmax일때 두 수식에 차이가 발생할 뿐 모두 dJdz를 구한다.

```python
np.random.seed(0)

print(dJdz_sigmoid(np.random.randn(3),np.random.randn(3)))
print(dJdz_softmax(np.random.randn(3),np.random.randn(3)))
```

```
[-4.90531647 -0.64834065 -1.89126428]
[ 0.53948992 -0.29540078 -1.55749236]
```

Expected Outputs

```
[-4.90531647 -0.64834065 -1.89126428]
[ 0.53948992 -0.29540078 -1.55749236]
```

올바르게 구현됨을 위 그림으로 확인한다.

```
def my_forward(l1, l2, l3, X_in):
    ### START CODE HERE ###

    a_1 = mySigmoid(l1.forward(X_in))                    # first stage forward
    a_2 = mySigmoid(l2.forward(a_1))                    # second stage forward
    a_3 = mySoftmax(l3.forward(a_2))                    # third stage forward

    ### END CODE HERE ###
    return a_1, a_2, a_3
```

위에서 구현한 함수들을 종합하여 3개 layer에 대한 forward연산을 구현한 함수이다. 이전 레이어의 출력을 이후 레이어의 입력으로 넣어주는 것을 확인할 수 있고 마지막 레이어는 네트워크가 classification을 진행할 수 있게 decision function인 softmax를 사용하여 구현한다.

```
def my_backward(l1, l2, l3, a_1, a_2, a_3, X_in, y_true):
    ### START CODE HERE ###

    dw_3, db_3, wdJdz_3 = l3.backward(dJdz_softmax(a_3,y_true),a_2)
    dw_2, db_2, wdJdz_2 = l2.backward(dJdz_sigmoid(wdJdz_3,a_2),a_1)
    dw_1, db_1, _       = l1.backward(dJdz_sigmoid(wdJdz_2,a_1),X_in)

    ### END CODE HERE ###

    d_1 = [dw_1, db_1]
    d_2 = [dw_2, db_2]
    d_3 = [dw_3, db_3]

    return d_1, d_2, d_3
```

backward는 반대로 마지막 레이어부터 미분값을 전달하는 것을 확인할 수 있다. 각 레이어의 backward 출력값을 저장한다. 이는 가중치를 업데이트하기 위함이다.

```python
def my_loss(l1, l2, l3, X_in, y_true):
    ### START CODE HERE ###

    _,_,a_3 = my_forward(l1,l2,l3,X_in)  # only last layer's forward output is necessary
    loss = -(np.sum(y_true*np.log(a_3))) / X_in.shape[0]                # calculate loss

    ### END CODE HERE ###
    return loss

def my_predict(l1, l2, l3, X_in):
    ### START CODE HERE ###

    _,_,a_3 = my_forward(l1,l2,l3,X_in)           # only last layer's forward output is necessary
    pred = np.argmax(a_3,axis=1)                  # make prediction

    ### END CODE HERE ###
    return pred
```

loss와 prediction을 맡은 두 함수이다.

먼저 forward연산을 진행하고 loss는 multiclass cross entropy 수식을 구현하여 전체 loss를 구하고 이는 모든 샘플의 loss가 합쳐진 값이므로 입력값의 샘플 수로 나누어준다. prediction은 (샘플 수 x 예측할 class 수)의 형태인 a_3 변수에서 class축을 기준으로 가장 큰 값을 가진 인덱스를 추출하여 각 샘플당 이 샘플이 어떤 클래스인지 예측한 값을 반환한다.

```python
# alpha: learning rate, lamda: regularization factor
alpha = 0.01
n_epochs = 5000

for epoch in range(n_epochs):
    ### START CODE HERE ###

    # Forward Path
    a_1, a_2, a_3 = my_forward(l1,l2,l3,X_train)   # forward path

    # Backward Path
    d_1, d_2, d_3 = my_backward(l1,l2,l3,a_1,a_2,a_3,X_train,y_train)   # backward path

    ### END CODE HERE ###
    #save weight for update
    dw_1, db_1 = d_1
    dw_2, db_2 = d_2
    dw_3, db_3 = d_3

    # Update weights and biases
    ### START CODE HERE ###
    #each layer's weight and bias are updated with backpropagation
    l3.wegt = l3.wegt - (alpha * dw_3)
    l3.bias = l3.bias - (alpha * db_3)
    l2.wegt = l2.wegt - (alpha * dw_2)
    l2.bias = l2.bias - (alpha * db_2)
    l1.wegt = l1.wegt - (alpha * dw_1)
    l1.bias = l1.bias - (alpha * db_1)

    ### END CODE HERE ###

    # Print loss
    if ((epoch+1)%500==0):
        loss_J = my_loss(l1, l2, l3, X_train, y_train)
        print('Epoch: %4d,  loss: %10.8f' % (epoch+1, loss_J))
```

구현한 my_forward, my_backward 함수를 사용하고 매 epoch마다 가중치를 업데이트하게끔 구현하였다. 이때 alpha(learning rate)를 이용하여 조금씩 가중치가 업데이트되게끔 한다.

```
Epoch:  500,  loss: 0.03429470
Epoch: 1000,  loss: 0.01820917
Epoch: 1500,  loss: 0.01581421
Epoch: 2000,  loss: 0.01374552
Epoch: 2500,  loss: 0.01224504
Epoch: 3000,  loss: 0.00977645
Epoch: 3500,  loss: 0.00750443
Epoch: 4000,  loss: 0.00527477
Epoch: 4500,  loss: 0.00162356
Epoch: 5000,  loss: 0.00092371
```

loss가 점점 떨어지면서 모델이 train data를 학습하는 것을 확인한다.

```
from sklearn.metrics import accuracy_score

y_pred = my_predict(l1, l2, l3, X_test) #prediction

accuracy_score(y_pred, y_test)
```

```
0.9472222222222222
```

Neural Network from scikit-learn

```
from sklearn.neural_network import MLPClassifier

mlp = MLPClassifier(hidden_layer_sizes=(80, 70, ), activation='logistic', solver='sgd', \
                    alpha=0.01, learning_rate_init=0.01, max_iter=1000)

# Training/Fitting the Model
mlp.fit(X_train, y_train_num)

# Making Predictions
s_pred = mlp.predict(X_test)
accuracy_score(s_pred, y_test)
```

```
0.9666666666666667
```

구현한 모델과 sklearn에서 제공하는 MLPClassifier와 성능 비교이다. 구현한 모델이 성능이 약간 낮지만 준수한 정확도를 보여준다.

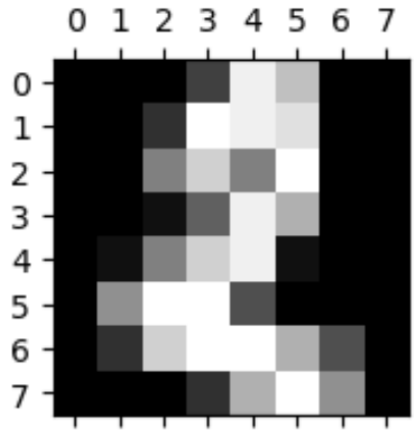
```
idx = np.random.randint(X_test.shape[0]) #display random sample
dimage = X_test_org[idx].reshape((8,8)) #1D -> 2D
plt.figure(figsize=(2, 2))
plt.gray()
plt.matshow(dimage, fignum=1)
plt.show()

X_input = np.expand_dims(X_test[idx], 0)

y_pred = my_predict(l1, l2, l3, X_input)

s_pred = mlp.predict(X_input)

print('My prediction is ' + str(y_pred[0]))
print('sk prediction is ' + str(s_pred[0]))
print('Actual number is ' + str(y_test[idx]))
```




```
My prediction is 2
sk prediction is 2
Actual number is 2
```

마지막으로 무작위로 샘플을 추출하여 같은 data에 대해서 두 모델이 올바르게 예측하는지 보여주는 코드이다. 두 모델 모두 2라고 예측하며 정확히 예측함을 눈으로 확인할 수 있었다.

Discussion

DNN을 직접 구현해보면서 forward와 backward연산이 어떻게 진행되는지 확인할 수 있었다. 네트워크가 데이터의 어떤 특징을 학습하는지는 딥러닝의 특성상 눈으로 확인하기 어려웠지만 많은 파라미터를 기반으로 학습된다는 것은 알 수 있었다. 지난 과제에서 배운 행렬곱이 실제 데이터에서 어떻게 이루어지는지 과제를 통해 확인할 수 있어서 유익했고 추가로 activation function을 구현하면서 효율적이고 올바른 연산을 위해 수식을 적절하게 수정하는 것에서 추후 알고리즘을 개발할 때 아키텍처를 잘 이해하고 코드를 구현하는 것이 중요할 것이다라는 생각을 하게 되었다. 행렬 연산에서 특히 행렬의 크기가 잘 맞지 않아 오류가 많이 발생했는데 프로그래밍을 할때 shape을 출력해보거나 중간중간 전체적인 모델의 구조를 확인해보면서 코드를 구현해야 오류를 줄이고 시간을 절약할 수 있겠다고 생각하였다.