인공지능프로그래밍
Lab 02: DNN1_dist

학 번 : 2021202058

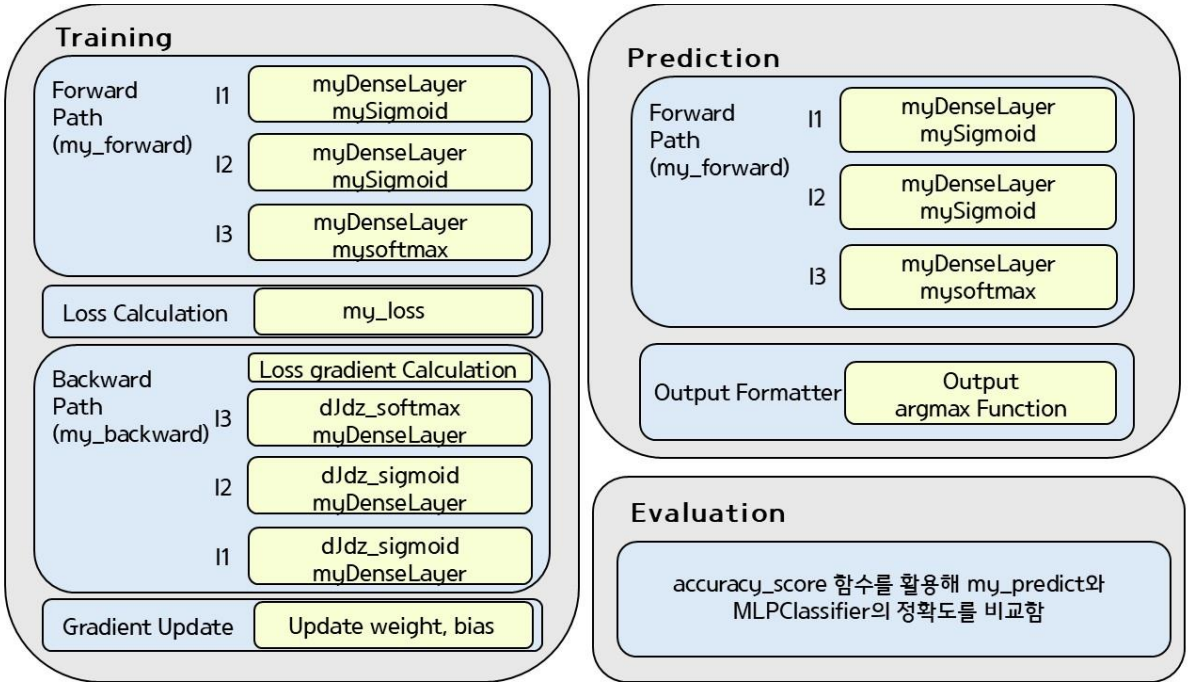
성 명 : 송채영

날 짜 : 2024.10.02

Lab Objective

이번 과제에서는 MNIST digit 데이터셋을 활용하여 Deep Neural Network를 구현하고, 숫자를 분류하는 모델을 구현해보는 것이다. 신경망의 각 층을 정의하고 sigmoid 및 softmax 활성화 함수를 수식을 바탕으로 구현해보고, Forward, Backward를 구현하며 동작 원리를 이해한다. 마지막으로 구현한 DNN을 사용하여 데이터를 분류하고, 사이킷런의 MLPClassifier와 정확도를 비교해본다.

Program flow



전체적인 코드의 흐름이다. 강의자료를 바탕으로 그림을 나타내었고, Training, Prediction, Evaluation 이렇게 3가지 단계로 나뉜다. 우선 Training 부분은 Forward path, Loss Calculation, Backward path, Gradient Update로 나눌 수 있다. 우선 Forward를 구현한 my_forward에서는 myDenseLayer와 mysigmoid 또는 mysoftmax 함수를 통해 입력이 신경망을 통과하고 출력을 계산한다. 이어서 Loss를 계산하는 my_loss 에서는 multiclass에 대한 cross entropy를 계산한다. 다음으로 Backward를 구현한 my_backward에서는 역전파를 이용해서 가중치와 편향에 대한 기울기를 계산한다. 마지막으로 weight와 bias를 업데이트를 한다. 다음으로 Prediction 부분이다. Forward path 와 output formatter로 나뉘는데, 먼저 my_forward 연산을 진행한 후 최종 레이어의 출력은 argmax 함수를 사용하여 최종 sofmax 출력에서 가장 높은 확률의 인덱스를 반환한다. 마지막으로 Evaluation부분으로, 모델 예측의 정확도를 평가한다. My_predict와 MLPClassifier, 두 모델을 평가를 accuracy_socre 함수를 통해 수행하며 평가를 진행한다.

Result

```
digits = load_digits()

# digits.data from sklearn contains 1797 images of 8x8 pixels
# Each image has a hand-written digit
digits_df = digits.images.reshape((len(digits.target), -1))
digits_tf = digits.target

# Splitting dataframe into train & test
X_train_org, X_test_org, y_train_num, y_test = train_test_split(digits_df, digits_tf, test_size= 0.20, random_state= 101)

# Digits data has range of [0,16], which often lead too big exponential values
# so make them normal distribution of [0,1] with the sklearn package, or you can just divide them by 16
sc = StandardScaler()
X_train = sc.fit_transform(X_train_org)
X_test = sc.transform(X_test_org)

n_classes = 10

# Transform Nx1 Y vector into Nx10 answer vector, so that we can perform one-to-all classification
y_train = np.zeros((y_train_num.shape[0],10))
for i in range(n_classes):
    y_train[:,i] = (y_train_num == i)
```

sklearn.datasets 모듈의 digits dataset을 사용했다. 손으로 쓴 숫자(0~9) 이미지를 모아둔 것이며 1797개의 이미지가 있고 각각은 8x8 픽셀이다. 픽셀 값의 범위는 0~16이며 0은 흰색, 16은 검은색을 의미한다. 코드를 살펴보면 원래 (1797, 8, 8)의 이미지를 (1797, 64)로 reshape 하였다. 즉 digits_df 변수는 데이터를 재구성 한 것이고, digits_tf는 각 이미지가 나타내는 숫자, 즉 라벨에 해당한다. 데이터는 80%를 train, 20%를 test로 사용하였고 X_train_org, X_test_org는 이미지 데이터, y_train_num, y_test는 해당 라벨이다. StandardScaler을 사용하여 데이터를 표준화하였고, one-hot encoding을 사용하여 라벨을 설정하였다.

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

    positive = (x >= 0) # Create a boolean array indicating which elements in x are positive
    x_p = x[positive]   # Extract the positive values from x
    x_n = x[~positive]  # Extract the negative values from x
    x[positive] = 1 / (1 + np.exp(-x_p))            # Apply the sigmoid function to the positive values
    x[~positive] = np.exp(x_n) / (1 + np.exp(x_n))  # Apply the sigmoid function to the negative values

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

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

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

$$sigmoid(x) = \frac{1}{1+e^{-x}} = \frac{e^{x}}{1+e^{x}} \cdots \text{수식}1$$

mysigmoid 함수는 수식1 처럼 정의되는 sigmoid 함수를 구현한 것이다. 우선 입력 x는 양수 값과 음수 값을 저장하는 numpy 배열이다. Positive는 boolean array로 x의 해당 요소가 양수인지(True), 음수인지(False)의 여부를 나타낸다. x_p는 x에서 양수값을 추출하고,

x_n은 x에서 음수 값을 추출한다. 이어서 sigmoid 함수를 적용하는 부분으로 양수 값인 경우와 음수 값인 경우를 수식에 맞게 구현하였다. 이를 통해 오버플로우 문제를 방지할 수 있다. 이어서 구현한 mySigmoid 함수를 호출하여 사용한 예시로 알맞게 구현이 된 것을 확인할 수 있다.

```
# define softmax. Assume (b, s)
def mySoftmax(x):
    ### START CODE HERE ###

    x = x - np.max(x, axis = -1, keepdims = True)        # Subtract the max value from each row for 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.])
```

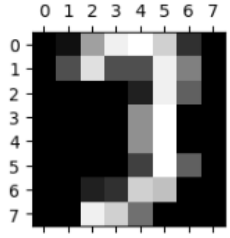
$$softmax(x_i) = \frac{e^{x_i}}{\sum_{k=1}^{n} e^{x_k}} \cdots \text{수식2}$$

mysoftmax 함수는 수식2처럼 정의되는 softmax 함수를 구현한 것이다. 수치적으로 안정성을 향상시키기 위해 각 행에서 최대값을 빼주었다. softmax에서 지수함수가 포함되어 값이 큰 경우 np.exp()로 인해 오버플로우가 발생할 수 있기 때문이다. 이어서 지수함수를 적용해주었고 마지막으로 softmax 출력의 합이 1이 되도록 정규화를 해주었다. 변환된 각 값을 모든 변환된 값의 합으로 나누어주었다. 이렇게 했을 때 mysoftmax 함수를 호출하여 확인해보면 알맞게 나오는 것을 확인할 수 있다.

```
# Print the shape of the digits DataFrame and the training data
print(digits_df.shape)
print(X_train.shape)
print(y_train.shape)
# Print the first sample from the original training data
print(X_train_org[0])

idx = np.random.randint(X_train.shape[0]) # Randomly select an index from the training data
dimage = X_train_org[idx].reshape((8,8))  # Reshape the selected sample
# Plot the image
plt.figure(figsize=(2, 2))
plt.gray()
plt.matshow(dimage, fignum=1)
plt.show()
print('The number is', y_train_num[idx])
```

```
(1797, 64)
(1437, 64)
(1437, 10)
[ 0.  0.  0.  9. 16.  6.  0.  0.  0.  0.  4. 15.  6. 15.  0.  0.  0.  0.
  8. 11.  9. 11.  0.  0.  0.  0.  8. 16. 14.  2.  0.  0.  0.  0. 11. 16.
 13.  0.  0.  0.  0.  6. 14.  2. 12.  9.  0.  0.  0.  5. 16. 11.  5. 13.
  4.  0.  0.  0.  3.  8. 13. 16.  9.  0.]
```




```
The number is 3
```

해당 코드는 digits dataset에서 데이터를 load 한 후 print로 shape을 확인해보고 시각화하는 코드이다. Dataframe 및 train data의 shape을 print해서 확인해보고 X_trian_org의 첫 번째 sample을 print해서 확인해보고 train sample을 무작위로 선택한 후 plot해서 확인해보는 코드이다.

```
class myDenseLayer:
    def __init__(self, n_out, n_in):
        # Initialize the weights and biases
        self.wegt = np.empty((n_out, n_in))
        self.bias = np.zeros((n_out))
        self.saved_x = None     # store x to use while backpropagation

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

        self.saved_x = x                                        # Save x for use in backpropagation
        x_lin = (np.matmul(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)
        # Check if input during forward pass matches the one during backpropagation
        assert np.array_equal(self.saved_x, x_in), print('x_in does not equal to input X.')
        ### START CODE HERE ###

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

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

신경망에서 fully connected layer을 나타내는 myDenseLayer 클래스를 정의한 코드이다. Init 메소드는 클래스를 초기화한다. 여기서 n_out은 layer의 뉴런 수(출력 크기), n_in은 이전 layer의 입력 특징(입력 크기)에 해당하고 self.wegt는 가중치를 저장하는 행렬, self.bias는 bias에 해당한다. 이어서 forward 부분이다. X는 일반적으로 (b,i) layer에 대한 입력이고 여기서 b는 batch, i는 input 크기에 해당한다. 입력 x는 나중에 기울기 계산을 위한 backpropagation에서 사용하기 위해 저장한다. 이어서 np.matmul(self.wegt, x.T)는 가중치 행렬과 입력의 내적을 계산한다. 이어서 .T를 다시 적용하고 bias를 더해주었다. Backward 부분이다. assert np.array_equal(self.saved_x, x_in)을 통해 정방향 전달 중 x_in이 역방향 전달에 사용된 입력과 동일함을 보장함을 알 수 있다. dw 는 가중치의 기울기로 역전파의 연쇄법칙을 사용하여 계산하였다. X.t는 출력에 대한 기울기고, x_in은 순방향 전달 중 입력에 해당한다. 둘을 내적하여 가중치 행렬의 기울기를 구했다. db는 bias에 대한 기울기로 배치의 모든 데이터 포인트에서 공유되므로 axis=0에서 기울기 x를 합산한다. wdJdz는 역전파 중 이전 레이어로 전달된 레이어의 입력에 대한 손실 기울기를 계산한다. 출력에 대한 그래디언트와 가중치 행렬 사이의 내적은 하위 레이어로 전달될 그래디언트를 제공한다. (모든 코드는 강의자료의 수식을 바탕으로 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 ]]]
```

위에서 구현한 myDenseLayer를 초기화 하고 test 해보는 코드이다. Expected output과 동일한 결과가 나오므로 잘 구현했음을 알 수 있다.

```
def dJdz_sigmoid(wdJdz_upper, az): # Backpropagation through the sigmoid activation function
    ### START CODE HERE ###

    dJdz = wdJdz_upper * (az * (1 - az))  # backpropagation through activation function

    ### END CODE HERE ###
    return dJdz

def dJdz_softmax(y_hat, y):       # Backpropagation through the softmax activation function
    ### START CODE HERE ###

    dJdz =  y_hat - y                    # backpropagation through activation function

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

```
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]
```

위 코드는 backpropagation에서 기울기를 계산하는 sigmoid, softmax 활성화 함수를 구

현한 것이다. 우선 dJdz_sigmoid는 시그모이드 활성화 함수에 대한 전 입력 z에 대한 기울기를 계산한다. wdJdz_upper는 다음 layer의 기울기, az는 현재 layer의 활성화 값에 해당한다. 이어서 dHdz_softmax는 소프트맥스 활성화 함수에 대한 전 입력 z에 대한 손실 기울기를 계산한다. Y_hat은 예측 확률에 해당하고 y는 실제 레이블에 해당한다. 이어서 구현한 함수를 호출하여 확인했을 때 Expected Outputs과 결과가 일치함을 알 수 있다.

```
def my_forward(l1, l2, l3, X_in): # Forward propagation through three layers
    ### START CODE HERE ###

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

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

def my_backward(l1, l2, l3, a_1, a_2, a_3, X_in, y_true): # Backward propagation through three layers
    ### START CODE HERE ###

    dw_3, db_3, wdJdz_3 = l3.backward(dJdz_softmax(a_3, y_true), a_2)     # go through 3rd stage backward(softmax)
    dw_2, db_2, wdJdz_2 = l2.backward(dJdz_sigmoid(wdJdz_3, a_2), a_1)    # go through 2nd stage backward(sigmoid)
    dw_1, db_1, _       = l1.backward(dJdz_sigmoid(wdJdz_2, a_1), X_in)   # go through 1st stage backward(sigmoid)

    ### END CODE HERE ###

    # Return gradients for all layers
    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

def my_loss(l1, l2, l3, X_in, y_true): # Calculate the loss
    ### START CODE HERE ###

    a_1,a_2,a_3 = my_forward(l1, l2, l3, X_in)                  # Forward pass to get the predictions
    loss = -(np.sum(y_true * np.log(a_3))) / X_in.shape[0]      # calculate cross-entropy loss

    ### END CODE HERE ###
    return loss

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

    a_1,a_2,a_3 = my_forward(l1, l2, l3, X_in)      # Forward pass to get the softmax probabilities
    pred = np.argmax(a_3, axis=1)                   # make prediction

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

위 코드는 3개의 신경망 층을 사용하여 Forward Propagation, Backward Propagation, Loss Claculation, Prediction을 구현한 것이다. 각각 my_forward, my_Backward, my_loss, my_predict에 해당한다. 먼저 my_forward는 입력 데이터를 신경망의 각 층을 통과시키면서 순차적으로 처리한다. X_in은 입력 데이터이며, l1, l2, l3은 각각 layer에 해당한다. 먼저 l1을 통과시킨 후 mySigmoid를 적용하여 a1을 계산하고 l2를 통과시킨 후 mysigmoid를 적용해 a_를 계산하고 마지막으로 l3을 통과시킨 후 softmax를 적용하여 a3을 계산한다. 이어서 my_backward는 역전파를 통해 각 층의 weight와 bias에 대한 그레디언트를 계산

한다. 먼저 softmax와 crossentropy loss 함수의 기울기를 계산한 후 l3 layer에서의 기울기를 구한다. 이어서 l2에서는 sigmoid 함수의 기울기를 계산하고 그 값을 이용해 backward를 수행하여 기울기를 계산한다. L1에서도 같은 방식으로 기울기를 계산하고 backward해 기울기를 구한다. My_loss에서는 주어진 입력과 실제 값 y_true에 대해 cross-entropy loss를 계산한다. My_predict에서는 입력 데이터를 통해 예측을 수행한다. Forward를 통해 예측 활률을 계산하고 소프트맥스 출력인 a3에서 가장 높은 확률을 가지는 클래스 인덱스를 np.argmax로 선택해 예측값을 반환하도록 한다.

```
# Define the number of inputs, hidden units, and output classes
n_inputs  = 64
n_hidden1 = 80
n_hidden2 = 70
n_classes = 10

# Initialize three layers of the neural network
l1 = myDenseLayer(n_hidden1, n_inputs)
l2 = myDenseLayer(n_hidden2, n_hidden1)
l3 = myDenseLayer(n_classes, n_hidden2)

print(X_train.shape, y_train.shape)
print(l1.wegt.shape, l1.bias.shape)
print(l2.wegt.shape, l2.bias.shape)
print(l3.wegt.shape, l3.bias.shape)
```

```
(1437, 64) (1437, 10)
(80, 64) (80,)
(70, 80) (70,)
(10, 70) (10,)
```

Expected Outputs

```
(1437, 64) (1437, 10)
(80, 64) (80,)
(70, 80) (70,)
(10, 70) (10,)
```

위 코드는 신경망을 정의하고 각 층의 weight와 bias를 초기화하며 입력 데이터와 신경망의 각 layer에 대한 정보를 출력한다. 출력한 값이 Expected Outputs과 일치하는 것을 확인할 수 있다.

```
# Weights are initialized to...
l1.wegt = np.random.randn(n_hidden1, n_inputs)
l2.wegt = np.random.randn(n_hidden2, n_hidden1)
l3.wegt = np.random.randn(n_classes, n_hidden2)
```

위 코드는 신경망의 각 층에 대한 가중치를 무작위로 초기화하는 코드이다.

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

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

    # Forward Path
    a_1, a_2, a_3 = my_forward(l1, l2, l3, X_train)                          # Forward pass through the network

    # Backward Path
    d_1, d_2, d_3 =  my_backward(l1, l2, l3, a_1, a_2, a_3, X_train, y_train)   # Backward pass to compute gradients for all layers

    ### END CODE HERE ###
    # Extract the gradients for weights and biases from each layer
    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 ###
    l3.wegt -= alpha * dw_3
    l3.bias -= alpha * db_3
    l2.wegt -= alpha * dw_2
    l2.bias -= alpha * db_2
    l1.wegt -= alpha * dw_1
    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))
```

```
Epoch:  500,  loss: 0.01596297
Epoch: 1000,  loss: 0.00289776
Epoch: 1500,  loss: 0.00176497
Epoch: 2000,  loss: 0.00104026
Epoch: 2500,  loss: 0.00081159
Epoch: 3000,  loss: 0.00058899
Epoch: 3500,  loss: 0.00046510
Epoch: 4000,  loss: 0.00033567
Epoch: 4500,  loss: 0.00027970
Epoch: 5000,  loss: 0.00021400
```


신경망을 훈련시키기 위한 코드로 구현한 forward와 backward를 통해 weight와 bias를 업데이트 하고 loss를 출력한다. 먼저 forward Path 부분은 입력 데이터 x_train을 사용해 신경망을 통과시키고 각 층의 출력인 a1, a2, a3을 계산하였다. 이어서 backward path 부분은 forward 부분에서 계산된 출력과 실제 레이블 y_train을 사용해 기울기를 계산한다. 이 과정에서 각 층의 weight와 bias를 구한다. 이어서 각 층의 weight인 dw와 bias인 db 기울기를 추출하고 추출한 기울기를 사용하여 weight와 bias를 업데이트 한다. 이때 학습률을 곱해서 얼마나 이동할지를 조정해주었다. 500번째 에폭마다 현재 loss 값을 계산하고 출력해주었다. Loss를 확인해보면 학습이 잘 진행되는 것을 알 수 있다.

```
from sklearn.metrics import accuracy_score

# Make predictions on the test set using the trained model
y_pred = my_predict(11, 12, 13, X_test)

accuracy_score(y_pred, y_test)
```

```
0.9444444444444444
```

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
```

첫 번째 코드는 직접 구현한 모델에 대한 정확도이다. accuracy_score을 사용하여 확인해 본 결과 약 94% 정도의 정확도를 얻은 것을 확인할 수 있다. 이어서 두 번째 코드는 Scikit-learn의 MLPClassifier를 사용했을 때로 약 96% 정도의 정확도를 얻은 것을 확인할 수 있다. 직접 구현한 모델의 정확도가 조금 더 낮지만, 비슷한 결과를 얻은 것을 확인했다.

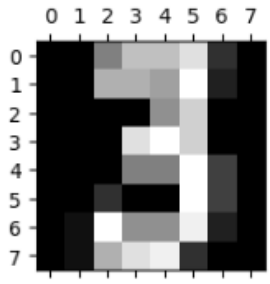
```
idx = np.random.randint(X_test.shape[0])
dimage = X_test_org[idx].reshape((8,8))
# Plot the image
plt.figure(figsize=(2, 2))
plt.gray()
plt.matshow(dimage, fignum=1)
plt.show()

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

y_pred = my_predict(11, 12, 13, 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 3
sk prediction is 3
Actual number is 3
```

마지막으로 테스트 데이터 중 하나의 이미지를 random으로 골라 plot 하고 구현한 모델과 MLPClassifier로 해당 이미지에 대한 예측을 수행하는 코드이다. 구현한 모델과 MLPClassifier 모두 3이라고 알맞게 예측하는 것을 확인했다.

Discussion

Depp Neural Network를 수업시간에 개념적으로, 수식적으로만 배웠을 때는 이해가 잘 안 가고 어떤 식으로 동작하는지 확 와닿지 않았는데 DNN을 직접 구현해보고 program flow를 그려보고 여러 번 실행해보고 시행착오를 겪으며 이해가 조금씩 됐던 것 같다. 또한 코드를 구현하면서 주석만으로는 어떻게 구현해야 할 지 이해가 잘 가지 않아 구현하는 데 있어 많은 시간이 걸렸다. 또한 Denselayer 부분을 구현한 후 expected output의 결과와도 일치했지만 실제 학습을 진행할 때 행렬의 크기가 안 맞는 경우도 생겨 어려움이 많았던 것 같다. 결론적으로는 강의 자료의 수식을 바탕으로 구현하니 완성할 수 있었지만 크기를 맞추는 것이 가장 어려운 것 같다. 이번 과제를 하면서 지난 과제에서 배웠던 부분도 써먹었고 다른 딥러닝 과제에서 이해가 안 됐던 것들이 이번에는 이해가 되기도 하며 많은 것을 배우고 느낄 수 있는 과제였던 것 같다.