인공지능프로그래밍
Lab 03: OPT_dist

학 번 : 2021202058

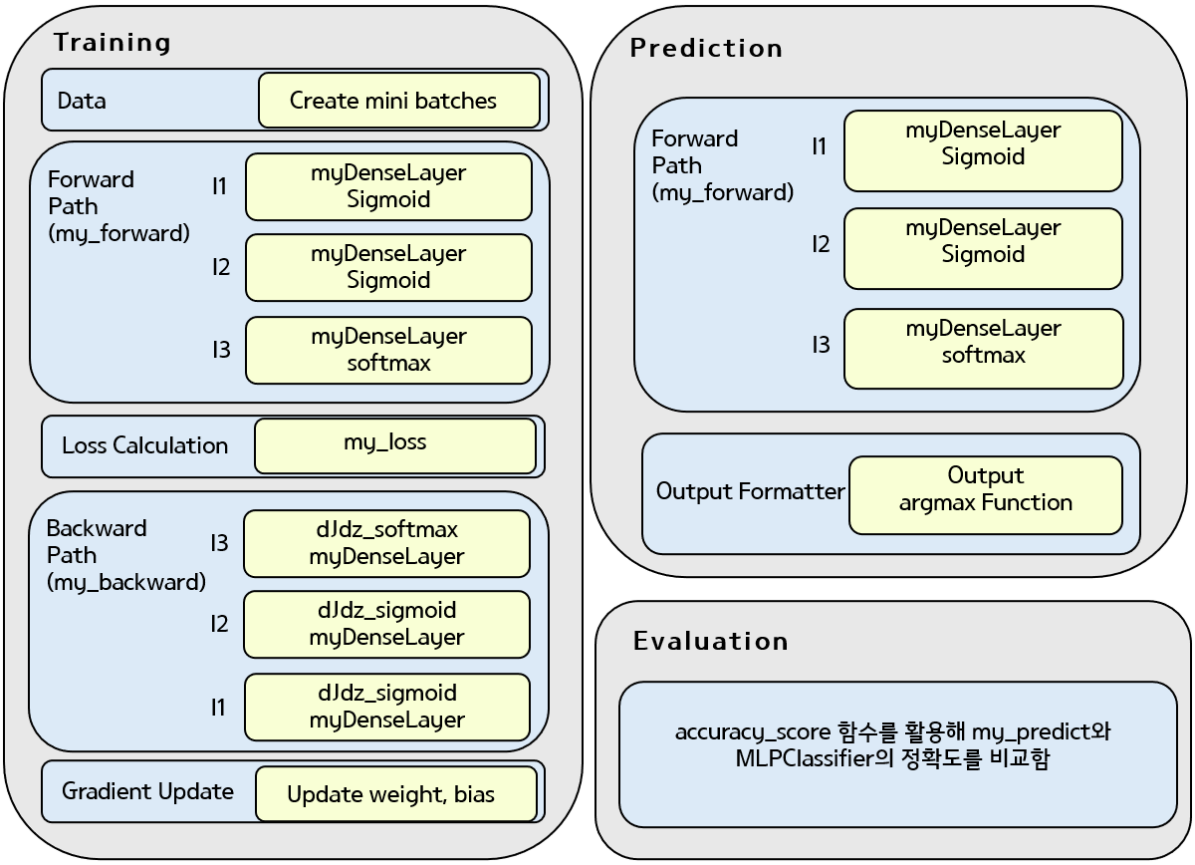
성 명 : 송채영

날 짜 : 2024.10.08

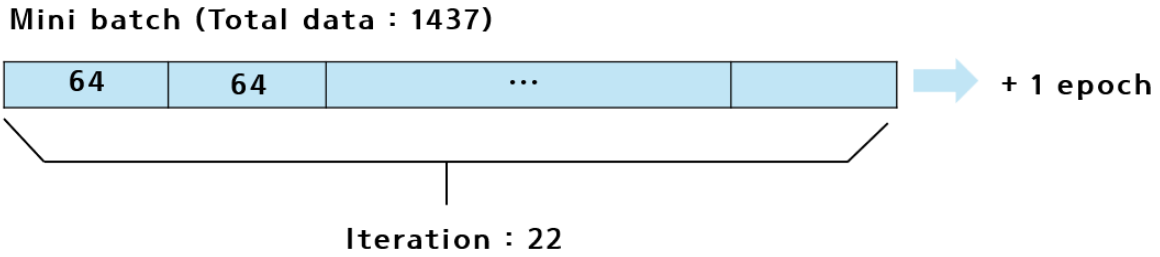
Lab Objective

이번 과제에서는 MNIST digit 데이터셋을 활용하여 Deep Neural Network를 구현하고, 숫자를 분류하는 모델을 구현해보는 것이다. 훈련 데이터를 나누기 위해 미니 배치를 구현하고 SGD, Momentum, Adagrad, RMSProp, Adam 등 다양한 optimizer를 구현해본 후, 마지막으로 구현한 DNN을 사용하여 학습시켜보고, 사이킷런의 MLPClassifier와 정확도를 비교해본다.

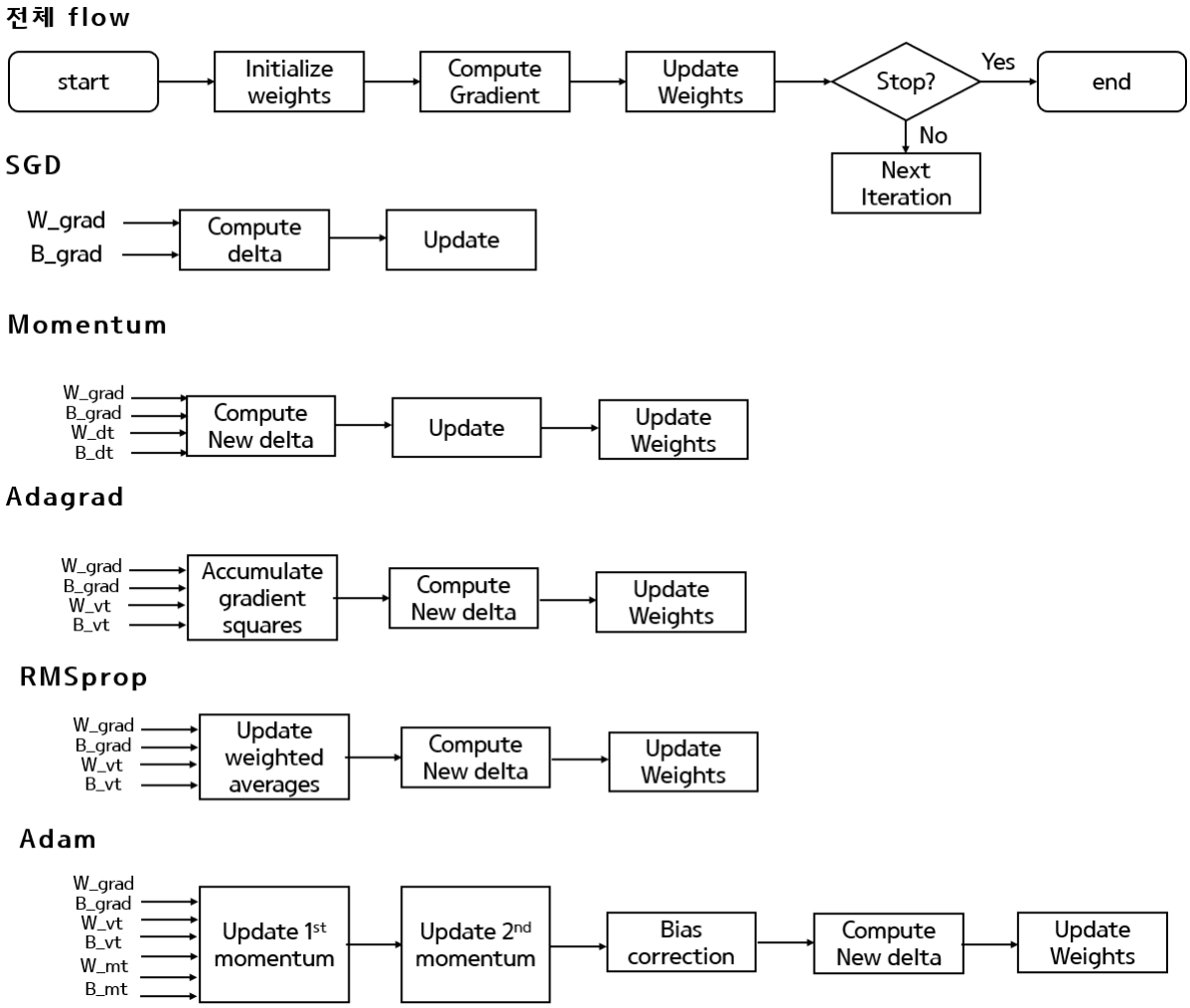
Program flow



전체적인 코드의 흐름이다. 강의자료를 바탕으로 그림을 나타내었고, Training, Prediction, Evaluation 이렇게 3가지 단계로 나뉜다. 지난 과제의 program flow와 동일하지만 달라진 점이 있다면 training을 할 때 mini batch로 데이터를 나누어서 진행한다는 것과 weights update를 할 때 기존에는 SGD만을 사용하였는데 이번 과제에서는 다양한 optimizer을 사용한다는 것이다. Prediction과 Evaluation 과정은 지난번과 동일하다.



Mini batch를 설명하는 그림이다. 총 데이터는 1797장인데 mini batch는 훈련할 때만 사용하기 때문에 total data는 1437장에 해당한다. 코드에서 batch size가 64로 설정되어 있으므로 batch size는 64이고 n_minibatches는 22에 해당한다.



Optimizer를 설명하는 그림이다. 우선 각 optimizer의 전반적인 흐름은 다음과 같다. 프로세스를 시작한 후 가중치를 초기화한다. 이후 backpropagation을 사용하여 가중치(w_grad)와 편향(B_grad)에 대한 기울기를 계산한다. 이후 가중치를 업데이트 하고 epoch수에 도달한 경우 종료, 아닌 경우 다음 iteration으로 이동한다. 다음으로 각 optimizer에 대해 설명하겠다. 우선 SGD는 W_grad와 B_grad를 입력으로 하고 delta를

계산한 후 weights와 bias를 업데이트한다. 다음으로 Momentum은 W_grad, B_grad, W_dt와 B_dt를 입력으로 받고 새로운 delta를 계산한 후 weights와 bias를 업데이트한다. Adagrad는 W_grad, B_grad, W_vt, B_vt를 입력으로 받고, 그라데이션의 제곱을 누적한다. 이후 누적된 기울기를 기반으로 새로운 delta를 계산하고 업데이트를 한다. RMSprop은 W_grad, B_grad, W_vt, B_vt를 입력으로 받고 제곱 기울기의 가중 평균을 업데이트한다. 이후 새로운 delta를 계산하고 업데이트를 한다. 마지막으로 Adam은 W_grad, B_grad, W_vt, B_vt, W_mt, B_mt를 입력으로 받고 첫 번째 momentum을 업데이트 하고 이어서 두 번쨰 momentum을 업데이트한다. 이후 bias correction을 적용한 후 새로운 delta를 계산하고 업데이트하는 식으로 동작한다.

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.

```python
from tensorflow.math import sigmoid as tf_sigmoid
from tensorflow.nn import softmax as tf_softmax

# Define the sigmoid function
def sigmoid(x):
    x = tf_sigmoid(x)
    return x.numpy()

# Define the softmax function
def softmax(x):
    x = tf_softmax(x)
    return x.numpy()
```

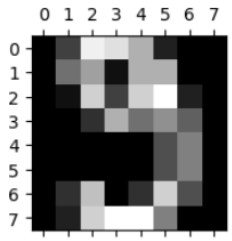
지난 과제에서는 sigmoid함수와 softmax 함수를 직접 구현하였지만 이번 과제에서는 Tensorflow에서 제공하는 sigmoid와 softmax를 사용하여 적용하였다.

```python
# 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 9
```

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.

```python
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 forward 전달 중 x_in 이 backward 전달에 사용된 입력과 동일함을 보장함을 알 수 있다. dw 는 가중치의 기울기로 backpropagation의 연쇄법칙을 사용하여 계산하였다. X.t는 출력에 대한 기울기고, x_in은 forward 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 backpropagation 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.)

```python
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
```

위 코드는 backpropagation에서 기울기를 계산하는 sigmoid, softmax 활성화 함수를 구현한 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.

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

    a_1 = sigmoid(layers[0].forward(X_in))              # first stage forward(sigmoid)
    a_2 = sigmoid(layers[1].forward(a_1))               # second stage forward(sigmoid)
    a_3 = softmax(layers[2].forward(a_2))               # third stage forward(softmax)

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

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

    dw_3, db_3, wdJdz_3 = layers[2].backward(dJdz_softmax(a_3, y_true), a_2)    # go through 3rd stage backward(softmax)
    dw_2, db_2, wdJdz_2 = layers[1].backward(dJdz_sigmoid(wdJdz_3, a_2), a_1)    # go through 2nd stage backward(sigmoid)
    dw_1, db_1, _       = layers[0].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(layers, X_in, y_true): # Calculate the loss
    l1, l2, l3 = layers
    ### START CODE HERE ###

    a_1,a_2,a_3 = my_forward(layers, 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(layers, X_in):      # Make predictions
    l1, l2, l3 = layers
    ### START CODE HERE ###

    a_1,a_2,a_3 = my_forward(layers, 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 backpropagation을 통해 각 층의 weight와 bias에 대한 그레디언트를 계산한다. 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)

layers = [l1, l2, l3]

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,)
```

위 코드는 신경망을 정의하고 각 층의 weight와 bias를 초기화하며 입력 데이터와 신경망의 각 layer에 대한 정보를 출력한다.

```
# 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
def create_mini_batches(X, y, batch_size=64):
    mini_batches = [] # Initialize a list to hold the mini-batches

    n_minibatches = (X.shape[0] // batch_size) # Calculate the number of complete mini-batches
    n_variables = X.shape[1] # Get the number of features  in the input data
    ### START CODE HERE ###

    data = np.hstack((X,y)) # concatenate X and y with np.hstack
    np.random.shuffle(data) # shuffle the combined data randomly

    for i in range(n_minibatches):
        mini_batch = data[i * batch_size:(i + 1) * batch_size,:] # get a slice of mini-batch
        X_mini, y_mini = mini_batch[:,:-y.shape[1]], mini_batch[:, -y.shape[1]:] # split mini-batch into X & y
        mini_batches.append((X_mini, y_mini))

    # Check for any remaining data that didn't fit into a complete mini-batch
    if data.shape[0] % batch_size != 0:
        mini_batch = data[n_minibatches* batch_size:] # process the remaining data
        X_mini, y_mini = mini_batch[:,:-y.shape[1]], mini_batch[:, -y.shape[1]:] # split mini-batch into X & y

        mini_batches.append((X_mini, y_mini)) # Add the last mini-batch to the list

    ### END CODE HERE ###
    return mini_batches
```

위 코드는 mini batch를 구현한 코드로, 입력 데이터 x와 해당 라벨 y를 받아 batch_size인 64만큼 데이터를 나누고 랜덤하게 섞은 후 mini batch를 생성하여 반환한다. 주석의 요구사항에 맞게 입력 데이터 x와 출력 데이터 y를 np.hstack을 사용하여 하나의 배열로 결합하였다. 이후 결합된 데이터를 무작위로 섞어 학습이 편향되지 않도록 하였고 이어서 batch size인 64 단위로 데이터를 순차적으로 나누어 mini batch를 생성하였다. 슬라이싱을 통해 만들었고 특징 값 x_mini와 라벨 y_mini로 분리하였다. 이후 데이터의 크기가 batch size로 나누어떨어지지 않으면 남은 데이터를 마지막 mini batch로 처리한다. 마지막으로 리스트에 담은 후 반환한다.

```python
np.random.seed(1)

a = np.arange(20).reshape(10,2)
b = -np.arange(10,20).reshape(10,1)

c = create_mini_batches(a, b, 4)
for mini_X, mini_y in c:
    print(mini_X)
    print(mini_y, '\n')
```

```
[[ 4  5]
 [18 19]
 [12 13]
 [ 8  9]]
[[-12]
 [-19]
 [-16]
 [-14]]

[[ 0  1]
 [ 6  7]
 [ 2  3]
 [14 15]]
[[-10]
 [-13]
 [-11]
 [-17]]

[[16 17]
 [10 11]]
[[-18]
 [-15]]
```

expected outpu:

```
[[ 4  5]          [[ 0  1]          [[16 17]
 [18 19]           [ 6  7]           [10 11]]
 [12 13]           [ 2  3]          [[-18]
 [ 8  9]]          [14 15]]          [-15]]
[[-12]            [[-10]
 [-19]             [-13]
 [-16]             [-11]
 [-14]]            [-17]]
```

위 코드는 구현한 mini batch 코드가 올바르게 동작되는지 확인하는 코드이다. Expected output과 동일한 결과가 나오는 것을 확인할 수 있다.

```
class myOptParam:
    def __init__(self, n_out, n_in):
        # Initialize previous delta values for the momentum optimizer
        self.W_dt = np.zeros((n_out, n_in)) # Weight delta
        self.B_dt = np.zeros(n_out) # Bias delta
         # Initialize variables for other optimizers
        self.W_mt = np.zeros((n_out, n_in)) # First moment for weights
        self.B_mt = np.zeros(n_out) # First moment for bias
        self.W_vt = np.zeros((n_out, n_in)) # Second moment for weights
        self.B_vt = np.zeros(n_out) # Second moment for bias
```

```
def my_optimizer(lyr, opt, W_grad, B_grad, solver='sgd', learning_rate=0.01, iter=1):
    epsilon = 1e-8  # arbitrary small number
    alpha = eta = learning_rate

    # Check if iteration starts from 1
    if iter==0:
        print('iteration should start from 1.')

    # optimizer routines
    if  solver=='sgd':
        W_dlt = alpha * W_grad # Update weights using gradient descent
        B_dlt = alpha * B_grad # Update biases using gradient descent

    elif solver=='momentum':
        gamma = 0.9 # Default momentum factor
        ### START CODE HERE ###
        W_dlt = gamma * opt.W_dt + alpha * W_grad # Calculate momentum update for weights
        B_dlt = gamma * opt.B_dt + alpha * B_grad # Calculate momentum update for biases
        opt.W_dt = W_dlt # keep weight delta for next iteration
        opt.B_dt = B_dlt # keep bias delta for next iteration
        ### END CODE HERE ###

    elif solver=='adagrad':
        ### START CODE HERE ###
        opt.W_vt = opt.W_vt + np.square(W_grad) # Accumulate squared gradients for weights
        opt.B_vt = opt.B_vt + np.square(B_grad) # Accumulater squared gradients for biases
        W_dlt = (alpha * W_grad) / np.sqrt(opt.W_vt + epsilon) # calculate new delta for weight
        B_dlt = (alpha * B_grad) / np.sqrt(opt.B_vt + epsilon) # and for bias
        ### END CODE HERE ###

    elif solver=='rmsprop':
        beta2 = 0.9 # default setting
        ### START CODE HERE ###
        opt.W_vt = beta2 * opt.W_vt + (1 - beta2) * np.square(W_grad) # blending with second momentum
        opt.B_vt = beta2 * opt.B_vt + (1 - beta2) * np.square(B_grad) # also doging samething for bias
        W_dlt = (alpha * W_grad ) / np.sqrt(opt.W_vt + epsilon) # calculate new delta for weight
        B_dlt = (alpha * B_grad ) / np.sqrt(opt.B_vt + epsilon) # and for bias
        ### END CODE HERE ###

    elif solver=='adam':
        beta1, beta2 = 0.9, 0.99  # default setting
        ### START CODE HERE ###
        opt.W_mt = beta1 * opt.W_mt + (1 - beta1) * W_grad # blending with first momentum
        opt.B_mt = beta1 * opt.B_mt + (1 - beta1) * B_grad # first momentum for bias
        opt.W_vt = beta2 * opt.W_vt + (1 - beta2) * np.square(W_grad) # blending with second momentum
        opt.B_vt = beta2 * opt.B_vt + (1 - beta2) * np.square(B_grad) # second momentum for bias

        W_mc = opt.W_mt / (1 - np.power(beta1, iter)) # bias correction of first momentum for weight
        B_mc = opt.B_mt / (1 - np.power(beta1, iter)) # and for bias term
        W_vc = opt.W_vt / (1 - np.power(beta2, iter)) # bias correction of second momentum for weight
        B_vc = opt.B_vt / (1 - np.power(beta2, iter)) # and for bias term
        W_dlt = alpha * W_mc / (np.sqrt(W_vc) + epsilon) # calculate new delat for weight
        B_dlt = alpha * B_mc / (np.sqrt(B_vc) + epsilon) # and for bias
        ### END CODE HERE ###

    else:
        print('optimizer error')

    # Update Layer weights and biases
    lyr.wegt = lyr.wegt - W_dlt # Adjust weights
    lyr.bias = lyr.bias - B_dlt # Adjust biases

    return
```

위 코드는 다양한 optimizer 함수를 구현한 코드이다. 먼저 myOptParam 클래스는 아래의 알고리즘에서 사용될 매개변수를 초기화한다. 이어서 my_optimizer 함수는 신경망 계층의 가중치와 편향을 업데이트하는 optimizer 함수이다.

- SGD

$\theta_t = \theta_{t-1} - \alpha \nabla_\theta J(\theta_{t-1})$

-Momentum

$\theta_t = \gamma \Delta\theta_{t-1} + \alpha \nabla_\theta J(\theta_{t-1})$

-AdaGrad

$v_t = v_{t-1} + \nabla_\theta J(\theta_{t-1})^2 \quad \Delta\theta_t = \frac{\eta \nabla_\theta J(\theta_{t-1})}{\sqrt{v_t + \epsilon}}$

-RMSProp

$v_t = \beta v_{t-1} + (1 - \mathrm{B}) \nabla_\theta J(\theta_{t-1})^2 \quad \Delta\theta_t = \frac{\eta \nabla_\theta J(\theta_{t-1})}{\sqrt{v_t + \epsilon}}$

-Adam

$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \nabla_\theta J(\theta_{t-1}) \quad v_t = \beta_2 v_{t-1} + (1 - \beta_2) \nabla_\theta J(\theta_{t-1})^2$

$m_c = \frac{m_t}{1 - \beta_1^t} \quad v_c = \frac{v}{1 - \beta_2^t} \quad \Delta\theta_t = \frac{\eta m_c}{\sqrt{v_t + \epsilon}}$

강의자료에 있는 위와 같은 각 optimizer 수식을 사용하여 구현하였다.

다음으로 아래의 코드는 무작위로 생성된 값들을 바탕으로 위에서 구현한 optimizer 수식이 올바르게 동작하는지 확인하는 코드이다. Expected output과 동일한 결과가 나오는 것을 확인할 수 있다.

```
np.random.seed(101)

lyr = myDenseLayer(2,3)
opt = myOptParam(2,3)

# Initialize weights, biases, and optimizer states
lyr.wegt = np.random.randn(2,3)
lyr.bias = np.random.randn(2)
opt.W_dt = np.random.randn(2,3)
opt.B_dt = np.random.randn(2)
opt.W_mt = np.random.randn(2,3)
opt.B_mt = np.random.randn(2)
opt.W_vt = np.abs(np.random.randn(2,3))
opt.B_vt = np.abs(np.random.randn(2))
# Random gradients for weights and biases
W_grad = np.random.randn(2,3)
B_grad = np.random.randn(2)

# optimizer settings are: 'sgd', 'momentum', 'adagrad', 'rmsprop', 'adam'
opts = ['sgd', 'momentum', 'adagrad', 'rmsprop', 'adam']
expt = [[ 7.67789007,  8.16882972, 10.34203348, -3.22934657],
        [14.46528172,  15.04341688, 19.30016537, -4.77070266],
        [22.50872929,  22.74302212, 28.47667875, -7.62607443],
        [30.69802889,  30.60433129, 37.72651766, -10.62235939],
        [29.41774022,  19.27573813, 23.68071186,  1.52919472]]
test_passed = True

# Loop through each optimizer
for i, sol in enumerate(opts):
    my_optimizer(lyr, opt, W_grad, B_grad, sol, 10, 3)
    print('For '+sol+':')
    res = np.concatenate((lyr.wegt[0], lyr.bias[0:1]), axis=0)
    print(res)
    if not np.allclose(res, expt[i]):
        print(sol+' failed.')
        test_passed = False
if test_passed: print('test passed.')
else: print('test failed.')
```

```
For sgd:
[ 7.67789007  8.16882972 10.34203348 -3.22934657]
For momentum:
[14.46528172 15.04341688 19.30016537 -4.77070266]
For adagrad:
[22.50872929 22.74302212 28.47667875 -7.62607443]
For rmsprop:
[ 30.69802889  30.60433129  37.72651766 -10.62235939]
For adam:
[29.41774022 19.27573813 23.68071186  1.52919472]
test passed.
```

Expected Outputs

For SGD:

```
[ 7.67789007  8.16882972 10.34203348 -3.22934657]
```

For Momentum:

```
[14.46528172 15.04341688 19.30016537 -4.77070266]
```

Fpr Adagrad:

```
[22.50872929 22.74302212 28.47667875 -7.62607443]
```

For RMSProp:

```
[ 30.69802889  30.60433129  37.72651766 -10.62235939]
```

For Adam:

```
[29.41774022 19.27573813 23.68071186  1.52919472]
```

```
o1 = myOptParam(n_hidden1, n_inputs)
o2 = myOptParam(n_hidden2, n_hidden1)
o3 = myOptParam(n_classes, n_hidden2)
```

위 코드는 세개의 layer에 대한 optimizer 매개변수를 초기화하는 코드이다.

```
# optimizer settings are: 'sgd', 'momentum', 'adagrad', 'rmsprop', 'adam'
# alpha is learning rate
optimizer ='adam'
alpha = 0.01
n_epochs = 1000

for epoch in range(n_epochs): # Training loop

    batches = create_mini_batches(X_train, y_train, batch_size=64)
    for one_batch in batches:
        X_mini, y_mini = one_batch
        batch_len = X_mini.shape[0]  # last batch might have different length

        # Forward Path
        a_1, a_2, a_3 = my_forward(layers, X_mini)  # Forward pass through the network

        # Backward Path
        d_1, d_2, d_3 = my_backward(layers, a_1, a_2, a_3, X_mini, y_mini) # Backward pass to compute gradients for all layers

        # 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
        my_optimizer(l1, o1, dw_1, db_1, solver=optimizer, learning_rate=alpha, iter=epoch+1)
        my_optimizer(l2, o2, dw_2, db_2, solver=optimizer, learning_rate=alpha, iter=epoch+1)
        my_optimizer(l3, o3, dw_3, db_3, solver=optimizer, learning_rate=alpha, iter=epoch+1)

    # Print loss
    if ((epoch+1)%100==0):
        loss_J = my_loss(layers, X_train, y_train)
        print('Epoch: %4d,  loss: %10.8f' % (epoch+1, loss_J))
```

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.

```
from sklearn.metrics import accuracy_score

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

accuracy_score(y_pred, y_test)
```

```
0.9805555555555555
```

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)
```

첫 번째 코드는 직접 구현한 모델에 대한 정확도이다. accuracy_score을 사용하여 확인해 본 결과를 얻은 것을 확인할 수 있고, 두 번째 코드는 Scikit-learn의 MLPClassifier를 사용했을 때의 결과를 확인할 수 있다.

```
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(layers, 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]))
```

마지막으로 테스트 데이터 중 하나의 이미지를 random으로 골라 plot 하고 구현한 모델과 MLPClassifier로 해당 이미지에 대한 예측을 수행하는 코드이다.

Optimizer 별 정확도를 비교해보겠다.

-SGD

```
Epoch:  100,  loss: 0.00455763
Epoch:  200,  loss: 0.00208210
Epoch:  300,  loss: 0.00132452
Epoch:  400,  loss: 0.00096241
Epoch:  500,  loss: 0.00075172
Epoch:  600,  loss: 0.00061451
Epoch:  700,  loss: 0.00051833
Epoch:  800,  loss: 0.00044732
Epoch:  900,  loss: 0.00039283
Epoch: 1000,  loss: 0.00034975
```

```python
from sklearn.metrics import accuracy_score

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

accuracy_score(y_pred, y_test)
```

```
0.95
```

Neural Network from scikit-learn

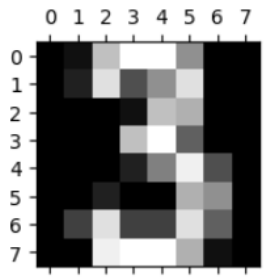
```python
from sklearn.neural_network import MLPClassifier

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)
```

```
0.9777777777777777
```




```
My prediction is 3
sk prediction is 3
Actual number is 3
```

-Momentum

```
Epoch:  100,  loss: 0.00016269
Epoch:  200,  loss: 0.00010406
Epoch:  300,  loss: 0.00007591
Epoch:  400,  loss: 0.00005950
Epoch:  500,  loss: 0.00004880
Epoch:  600,  loss: 0.00004129
Epoch:  700,  loss: 0.00003574
Epoch:  800,  loss: 0.00003148
Epoch:  900,  loss: 0.00002811
Epoch: 1000,  loss: 0.00002537
```

```
from sklearn.metrics import accuracy_score

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

accuracy_score(y_pred, y_test)
```

```
0.9583333333333334
```

Neural Network from scikit-learn

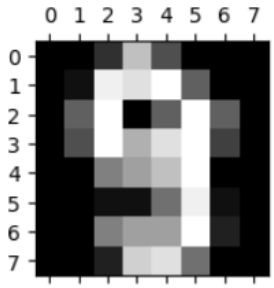
```
from sklearn.neural_network import MLPClassifier

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)
```

```
0.9805555555555555
```




```
My prediction is 9
sk prediction is 9
Actual number is 9
```

-Adagrad

```
Epoch:  100,  loss: 0.00000164
Epoch:  200,  loss: 0.00000087
Epoch:  300,  loss: 0.00000059
Epoch:  400,  loss: 0.00000044
Epoch:  500,  loss: 0.00000036
Epoch:  600,  loss: 0.00000030
Epoch:  700,  loss: 0.00000025
Epoch:  800,  loss: 0.00000022
Epoch:  900,  loss: 0.00000020
Epoch: 1000,  loss: 0.00000018
```

```
from sklearn.metrics import accuracy_score

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

accuracy_score(y_pred, y_test)
```

```
0.9666666666666667
```

Neural Network from scikit-learn

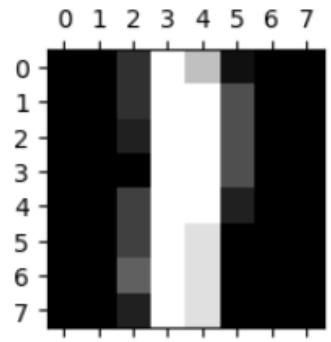
```
from sklearn.neural_network import MLPClassifier

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)
```

```
0.9777777777777777
```




```
My prediction is 1
sk prediction is 1
Actual number is 1
```

-RMSprop

```
Epoch:  100,  loss: 0.00000009
Epoch:  200,  loss: 0.00000006
Epoch:  300,  loss: 0.00000005
Epoch:  400,  loss: 0.00000004
Epoch:  500,  loss: 0.00000003
Epoch:  600,  loss: 0.00000003
Epoch:  700,  loss: 0.00000002
Epoch:  800,  loss: 0.00000002
Epoch:  900,  loss: 0.00000002
Epoch: 1000,  loss: 0.00000002
```

```
from sklearn.metrics import accuracy_score

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

accuracy_score(y_pred, y_test)
```

```
0.9722222222222222
```

Neural Network from scikit-learn

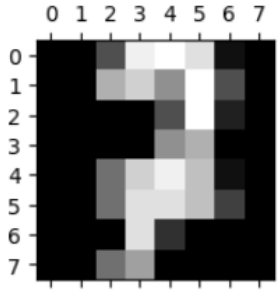
```
from sklearn.neural_network import MLPClassifier

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)
```

```
0.9611111111111111
```




```
My prediction is 7
sk prediction is 7
Actual number is 7
```

-Adam

```
Epoch:  100,  loss: 0.00000000
Epoch:  200,  loss: 0.00000000
Epoch:  300,  loss: 0.00000000
Epoch:  400,  loss: 0.00000000
Epoch:  500,  loss: 0.00000000
Epoch:  600,  loss: 0.00000000
Epoch:  700,  loss: 0.00000000
Epoch:  800,  loss: 0.00000000
Epoch:  900,  loss: 0.00000000
Epoch: 1000,  loss: 0.00000000
```

```
from sklearn.metrics import accuracy_score

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

accuracy_score(y_pred, y_test)
```

```
0.9805555555555555
```

Neural Network from scikit-learn

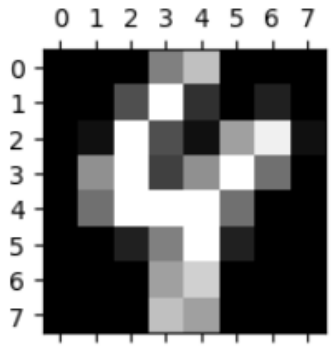
```
from sklearn.neural_network import MLPClassifier

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)
```

```
0.9777777777777777
```




```
My prediction is 4
sk prediction is 4
Actual number is 4
```

모든 결과는 한 번씩 실행해 보았기 때문에 일반적인 결과는 아니지만 5개의 결과는 SGD는 약 95%, Momentum은 약 95.8%, Adagrad는 약 96.6% RMSProp은 97.2%, 마지막으로 Adam은 약 98%를 얻었다. 5개의 optimizer 중 Adam의 정확도가 가장 높았다. 또한 Adam의 경우 loss값이 가장 낮게 나왔다. Adam optimizer의 경우 기울기의 첫 번째 momentum과 두 번째 momentum의 추정치를 모두 사용하기 때문에 다른 optimizer들과 비교했을 때 더 효과적인 결과가 나왔다고 생각이든다.

Discussion

지난 과제에 이어 Depp Neural Network를 구현해보았는데, 지난 과제를 보지 않고 다시

forward, backward 등을 구현해보니 복습도 되고 좋았다. 또한 minibatch 부분이 수업시간에 배웠던 것은 단순한데 구현하려고 하니 이해가 잘 가지 않았던 것 같다. 특히 x mini와 y mini를 split 하는 과정에서 어려움이 많았던 것 같다. 처음에는 y.shape으로 split 하지 않고 밑에 예시를 시도해보는 것에만 동작이 되게끔 구현을 했다 보니 코드를 학습시킬 때 크기가 안 맞는 문제가 생겼다. 다시 살펴보니 크기에 맞게 split을 했어야 했는데 그러지 않아 생겼던 오류였고, 알맞게 고쳐서 해결해주었다. Optimizer의 경우 수식 그대로 구현하면 되는 것이니 큰 어려움이 없었지만, 하나하나 직접 구현해보며 동작 방식도 더 잘 이해가 되는 것 같다. 강의로는 이해가 어려운 부분들을 직접 코드로 구현해보니 이해의 면에서 도움이 많이 되는 것 같아 좋다.